Virtual Human Anatomy and Surgery System

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Yanling Liu
Master of Science
Southwest Jiaotong University, China, 2001
Bachelor of Science
Southwest Jiaotong University, China, 1998

Director: Jim X. Chen, Professor
Department of Computer Science

Fall Semester 2008
George Mason University
Fairfax, VA

Dedication

To my wife Qian.

Acknowledgments

I would like to thank Dr. Jim X. Chen for his time, guidance, advice, and clear thinking. Without his generosity, this thesis would not have been written. I would also like to thank Dr. Daniel Carr, Dr. Gheorghe Tecuci, and Dr. Harry Wechsler for providing a solid grounding in computer science and image processing and for sharing their love of science. I would also like to thank Dr. Hassan Gomaa for his guidance in finishing this thesis.

Table of Contents

List of Tables

List of Figures

Abstract

VIRTUAL HUMAN ANATOMY AND SURGERY SYSTEM
Yanling Liu, PhD
George Mason University, 2008
Dissertation Director: Jim X. Chen

Historically, medical students have practiced on cadavers to learn human anatomy, as have physicians wanting to brush up on their knowledge. However, because of storage cost and limited availability of cadavers, practice on cadavers has proven problematic. As computers become more powerful, medical professors have dreamed of a day when they will be able to dissect bodies with the assistance of virtual reality. We have developed the Virtual Human Anatomy and Surgery System (VHASS) as a potential solution. VHASS uses cryosection images (natural-color images generated by slicing a frozen cadaver) to reconstruct computerized three-dimensional cadavers. VHASS enhances human anatomy education by creating three-dimensional volume models that include details of human organs, giving medical students and physicians unlimited access to realistic virtual cadavers. Major components in VHASS include three-dimensional virtual humans, direct volume rendering of virtual humans, surface models of segmented human parts, and real-time manipulation on virtual humans.

Direct volume rendering on un-segmented cryosection images is still an open research topic. Different from traditional volume rendering, which uses transfer functions to map scalar values to colors and opacity, direct volume rendering on cryosection images needs efficient transfer functions mapping vectors to opacity, which is complicated by the non-linearity of color space. We have created a series of new transfer functions for volume rendering on un-segmented cryosection images.

To create human part surface models, we separate human tissues within cryosection images, dissect all human organs according to their anatomic structures, and reconstruct a three-dimensional volume model for each part. VHASS renders each part as a high-resolution, natural-appearance three-dimensional model and labels it properly to facilitate learning. This enables users to group different parts to better understand human anatomy.

VHASS allows real-time interactions, such as drilling, scanning and slicing on human parts. We re-generate human part surface models at run-time for deforming interactions. We have analyzed the limitation of the well-known Marching Cubes algorithm and modified the algorithm to work with our data. We also have developed a new neighbor-based surface reconstruction algorithm, which has the same performance as the Marching Cubes algorithm but without the limitation of the Marching Cubes method. For better performance, the new algorithm has been ported onto the new graphics hardware using the geometry shader. Our implementation on the geometry shader serves as an example of exploiting the new GPU parallel processing hardware.

VHASS supports stereo rendering, haptic interaction, tracking and three-dimensional content production. Using the Sharp three-dimensional display on a laptop, VHASS provides low-cost, portable stereo rendering of human parts without the requirement of special glasses. Integrating with large size stereo projector and ultrasonic trackers, VHASS allows people to manipulate human parts in the immersive stereo environment. By integrating SensAble Onmi haptic device, VHASS enables people to feel the touch on human parts. VHASS integrates three-dimensional content creation by allowing students to print out physical models of human parts.

Chapter 1: Introduction

1.1 Motivation

The necessity of digitalized virtual human has been proven in many literatures [1][2][3][4][5]. Over the past few decades, many researchers have focused on creating a three-dimensional atlas of human anatomy for learning and teaching purposes. The earlier versions of such electrical human anatomy atlas were created from CT and MRI images without cadaver tissue colors. With the results of the Virtual Human Project, many atlas have been ported to support cadaver tissue colors using cryosection images (photo images generated by slicing and scanning a frozen cadaver). Comprehensive databases have been created to organize three-dimensional views of virtual human parts as well as two-dimensional images, audio/video files, illustration texts, and reference materials. The effectiveness of such three-dimensional atlas in human anatomy learning has been evaluated by various projects.

However, existing systems are not yet mature. There are several weakness and limitation of existing systems. First, existing systems has no direct volume rendering of unsegmented cryosection images. Compared to rendering segmented human parts, direct volume rendering of unsegmented cryosection images renders human parts more naturally by revealing transitions between different human parts.

Secondly, existing systems are unable to provide free manipulation of virtual human parts. The interactions in the existing three-dimensional atlas are usually limited to rotation, transparency rendering, and pre-generated effects, such as QuickTime VR movies. These atlas are often used as auxiliary materials in lecturing. Students are able to browse the atlas to pick and highlight interested human parts. However, students are not able to cut, slice, and drill on segmented human parts freely.

Missing accurate human parts colors and textures on surface models is another common weakness in existing systems. Usually these surface models are colored and textured by animator and artist to have vivid but inaccurate appearance. Moreover, these surface models are hollow inside so that interactions revealing inner part of human organs are not supported.

We were motivated to create a new human anatomy learning system to overcome above problems.

1.2 Problem Statements

In summary, features and functions missing in existing systems are:

- direct volume rendering on un-segmented cryosection images;
- accurate colors and textures on human part surface models;
- internal structure information in human part surface models;
- real-time deformation and interaction on virtual human parts.

To apply direct volume rendering on un-segmented cryosection images, a critical step is to assign appropriate opacity values to voxels so that people can distinguish different human parts. However, volume rendering on color volumes is not well understood yet and capturing boundary information on color images is still an open research area. We have focused on creating effective opacity mapping function to support direct volume rendering on cryosection images.

We also have created new 3D models for human parts. Instead of using hollow polygon meshes, we use both segmentation results and cryosection volumes. We use segmentation results to store implicit boundary surface information as well as internal structure information. Cryosection volumes are used to ensure correct colors and textures on human parts.

To support real-time manipulation, we have developed new surface reconstruction algorithms. Our new neighbor-based surface reconstruction algorithm is superior than other algorithms in terms of linear performance, linear memory requirement and close surface generation. We have modified the original Marching Cubes method to work on our data.

Finally, we have developed a series interaction and visualization tools based on our new opacity mapping functions and new 3D human part models. These tools help people to explore and interact with virtual humans.

1.3 Research Approach

In our system, we have implemented both direct volume rendering and 3D human part models. For volume rendering, we have improved existing work on designing effective opacity mapping functions on color volumes. We also have created new opacity estimation methods based on 3D edge detection as well as edge preserving filtering. Due to the non-linearity of color spaces, no opacity estimation method can dominate other in all situations. Our method is to find out all possible ways and let people to choose based on their needs.

We also have created new 3D models for human parts. Since both volume and surface have their own advantages as well as disadvantages, we combine volume and surface together for advantages from both. We visualize human parts as polygon meshes and use cryosection volume in behind to support real colors and textures. We use real-time surface reconstruction to regenerate surface when interaction happens on human part models.

To help human anatomy learning, we developed a series of unique tools for exploration and simulation on human parts. We also integrate virtual reality devices, such as haptic devices, stereo display, and tracking device to allow human anatomy education in a virtual reality environment.

1.4 Contributions

Our major contributions are:

- **New opacity transfer functions for direct volume rendering on unsegmented cryosection images**. Opacity transfer functions map voxel colors to opacity values so that human part boundaries are highlighted to be distinguishable in cryosection volumes. Compared to existing transfer functions, our transfer functions are more efficient in capturing boundary information and producing better visualization results.

- **New three-dimensional models of segmented human parts**. Our new models combine volume data and human parts segmentation together to provide accurate appearance of human parts in real-time deforming interaction such as cutting and drilling.
- **New algorithms for real-time surface reconstruction on segmented human parts**. Our new algorithms overcome limitations of the well-known Marching Cubes method while maintaining the same performance with the original Marching Cubes method.
- **New human anatomy learning platform**. Our new platform features accurate cadaver appearance and free manipulation on segmented anatomical structures.
- **Various simulation and visualization for human anatomy learning**.

In summary, we have developed a new platform, VHASS (Virtual Human Anatomy and Surgery System), for human anatomy knowledge learning. Starting from cryosection images (photo images generated by slicing and scanning a frozen cadaver), we have focused on accurate virtual human visualization as well as free real-time manipulation on segmented human parts.

1.5 Related Works

1.5.1 Visible Human Project

The Virtual Human Project was established in 1989 to build a digital image of volumetric data representing complete, normal adult male and female anatomy [6][1][2]. The long-term goal of the Visible Human Project is to produce a system of knowledge structures, which will transparently link visual knowledge forms to symbolic knowledge formats [1].

Early in 1989, under the direction of the National Library of Medicine's Board of Regents, an ad hoc planning panel was convened to provide the Library with in-depth guidance as to its proper role in the rapidly changing field of digital imaging. The panel recognized that much of understanding of complicated health and disease processes actually lay in images not text [1]. Over the centuries people have been using texts to describe our view of body systems, organs, and molecules because of the difficulty and expense of creating and distributing the images. With this as a background, the Panel recommended that the National Library of Medicine undertake a first project: building a digital image library of volumetric data representing a complete normal adult human and female.

It took approximately two and half a year to identify the male and female cadaver. The male data set consists of MR, CT, and color anatomical images. Axial MR images are taken at 4 mm intervals for the head and neck and longitudinal sections of the remainder of the body. The resolution of the MR images is 256 pixels by 256 pixels. Each pixel has 12 bits of gray tone.

The CT data consists of axial CT scans of the entire body taken at 1 mm intervals at a resolution of 512 pixels by 512 pixels where each pixel is made up of 12 bits of gray tone. The axial anatomical images are 2,048 pixels by 1,216 pixels where each pixel is defined by 24 bits of color, each image consisting of about 7.5 megabytes of data. The anatomical cross-sections are also at 1 mm intervals and coincide with the CT axial images. There are 1,871 cross-sections for CT and anatomy images obtained from the male cadaver.

The data set from the female cadaver has similar characteristics to the male cadaver. The axial anatomical images were obtained at 0.33 mm intervals instead of 1.0 mm intervals resulting in over 5,000 anatomical images. The female data set is about 40 gigabytes in size. To enable developers who are interested in three-dimensional reconstructions to work with cubic voxels, spacing in the "Z" direction was reduced to 0.33 mm in order to match the pixel spacing in the "XY" plane.

The Visible Human Project has been the focus of many researchers creating three-dimensional virtual humans for human anatomy learning. The data sets are available to researchers from academia and industry who are interested in using the digital image data produced by this project through a non-financial licensing agreement with the National Library of Medicine. This approach makes it possible for interested parties to help the National Library of Medicine reach a consensus on the collection, maintenance, and distribution processes for digital image libraries which will be mutually beneficial [1].

1.5.2 VOXEL-MAN

VOXEL-MAN is a framework for generating volume based three-dimensional interactive atlas from cross-sectional images[3][7][8][9]. These atlases are based on a two-layer model of image volumes linked to a semantic network containing descriptive knowledge. For extraction of the model's contents, VOXEL-MAN provides a large set of visualization, exploration, and simulation tools.

The Visible Human in the VOXEL-MAN Framework

The primary pictorial bases of VOXEL-MAN atlas are radiologic cross-sectional images. While this has the advantage that anatomy can be linked to radiological imaging, anatomical detail is subject to improvement. The high resolution data sets of the Visible Human Project are an ideal basis for improving anatomical details. Based on direct volume rendering, Pommert et al. have presented a method of segmentation and visualization which provides high quality volume models [10].

Model Creation First, Pommert et al. transfered cryosection images of the male Visible Human into an image volume congruent with the photographic images. Then the image volume is segmented interactively using an "electronic sculpturing tool". On a cross-section, an expert paints a typical region of the anatomical structure under consideration. All voxels in the volume with similar RGB-tuples are then collected to form a mask. This mask is then refined by repeating the procedure in order to include the target structure completely. The resulting voxel cluster in RGB space has an ellipsoid shape, due to the slight correlation of the colors [10]. This cluster is then converted into the mathematical form of an ellipsoid, which facilitates subsequent computations. If other regions with the same characterization present in the volume, three-dimensional interactive cutting tools are used to isolate connected regions.

Some anatomical structures, such as nerves and small vessels, are too small to be segmented adequately. For these, Pommert et al. developed a modeling tool to create tube-like structures. An expert imposes ball-shaped markers of variable diameters onto the landmarks visible on the cross-sections or on the three-dimensional image. These markers are automatically connected to form tubes of varying diameters. Unlike the segmented objects, which are represented as sets of voxels, the modeled structures are represented as composed of small triangles.

Rendering Human Parts Segmented objects are rendered using modified volume rendering method. As in traditional volume rendering, viewing rays are casted from an image plane onto the image volume in viewing direction. However, the rays stop at the first encountered object. Only boundaries of human parts are rendered. Surface texture and surface inclination (important for proper computation of light reflection) are calculated from the RGB-tuples at the segmented border line. The decisive quality improvement is achieved by determining the object surfaces with a spatial resolution higher than the resolution of the original voxels.

Results The final result of Pommert's method is pre-computed "Intelligent QuickTime VR Movies" [11], which supports limited interactive browsing and exploration.

1.5.3 The Vesalius Project

The Vesalius Project at Columbia University is named after Andreas Vesalius, a sixteenth century anatomist whose work laid the foundations for all subsequent anatomical research. The Columbia University Health Sciences Library owns several first editions of his work.

The goal of the Vesalius Project is interactive electronic curriculum based on the Visible Human data. A comprehensive knowledge base is created to organize pre-computed three-dimensional anatomical images and associated video/audio material, two-dimensional images, explanatory text and reference materials.

1.5.4 Virtual Surgery Patient

Virtual Surgery Patient is designed to teach medical students and residents the principles and practices of surgery [12]. The Virtual Surgery Patient consists of a three-dimensional data set of handcrafted models of human anatomy. Animation of organs and structures are used to demonstrate surgical and physical exam maneuvers, allowing students to visualize anatomy in an in-depth manner.

Methods

Under the direction of surgeons and anatomists, two experienced medical illustrators used Maya animation software to create three-dimensional models with flexible polygon counts of organs in the human abdomen and thorax. The fundamental data reference used to resurrect the three-dimensional anatomy were the digitized photographic images of the Visible Human Project acquired from the National Library Medicine. The axial view of the cryosectioned cadaver allowed illustrators to correctly define the shape of a particular organ by tracing its contours on each photograph in a selected stack.

Three-dimensional surface models in the Virtual Surgery Patient are created by using Maya software. In the first step, the point cloud data of human organ boundaries from the Visible Human Project is imported into the Maya workspace. Then, surface curves are used to trace the structure of the point cloud. The curves act as guidelines for the creation of a low-polygon-count surface model. Finally, surface models are created using Maya tools.

Artificially created textures are applied onto the new low-polygon-count surface model for expressive animation effect. Within Maya software, animation handles and deformers are added onto the surface models to simulate surgery operations. As the Virtual Surgery

Patient continues to evolve, new animation handles are added to create an extensible tool kit for simulating a diverse range of surgical maneuvers.

Results

The Virtual Surgery Patient results in animations generated by Maya for educational purposes. The system has focused on providing animated surgery operation illustrations instead of real-time manipulation and surgery simulation on three-dimensional virtual humans.

1.5.5 Commercial Virtual Human Products

A.D.A.M. Interactive Anatomy

A.D.A.M. Interactive Anatomy (AIA) is an interactive learning DVD-ROM that enhances the study of human anatomy and related topics. Like existing systems, AIA includes a comprehensive database including two-dimensional images, illustration text and pre-rendered materials of three-dimensional models. The major task of the system is to teach anatomy based on detailed two-dimensional anatomy images. However, direct manipulation on virtual human parts is not available.

Visible Body

The Visible Body is a web-delivered application to facilitate understanding of human anatomy. The Visible Body includes three-dimensional models of over 1,700 anatomical structures, including all major organs and systems of the human body. With the Visible Body, people can:

- search and locate anatomical structures by name;
- interact on human part models such as move, rotate, and zoom;
- adjust transparency of human parts.

The three-dimensional models in the Visible Body are manually textured to achieve vivid colors. These colors do not, however, represent accurate live body or cadaver appearance.

1.6 Organization

This dissertation is organized as follows. In the second chapter, we introduce our data sets, preprocessing steps and segmentation on cryosection images. The third chapter focuses on visualizing human parts by direct volume rendering and new opacity mapping functions. New 3D models and new surface reconstruction algorithms are outlined in the fourth chapter. In the fifth chapter, we discuss our tools for virtual human learning and exploration. The sixth chapter summarizes future work on our research. Finally, summary and conclusions are discussed in the seventh chapter.

Chapter 2: Data Preparation and Acquisition

This chapter is organized as follows. In the first section, we introduce our data sets. The second section focuses on preprocessing steps. Human parts segmentation and our segmentation method are outlined in the third section.

2.1 Data Source

We have two different data sets. The first data set includes 2,100 cryosection images scanned from a frozen Asian male cadaver. Each image has a $3,072 \times 2,048$ pixel resolution with physical resolution 0.166×0.166 mm. Roughly, each uncompressed image requires 18 megabytes of storage space. Overall, uncompressed images require 37.8 gigabytes of storage space. Four different thicknesses are used: 0.1mm, 0.2mm, 0.5mm and 1.0mm. Thinner thickness is used when the human body shape exhibits rapid change on sagittal or coronal plane.

The second data set is the Visible Female from the Visible Human Project (VHP) at the National Library of Medicine [6]. The Visible Female data set includes 5,190 cryosection images. Each image has a $2,048 \times 1,216$ pixel resolution with physical resolution 0.33×0.33 mm. Roughly, each uncompressed image requires 7 megabytes of storage space. Each image has a physical thickness of 0.33mm.

2.2 Preprocessing Step

Cryosection images have a blue background since the frozen human body was surrounded by blue color ice. We remove this blue background by filling black color on pixels having more blue component than red component. We can do this because human tissue has more red component than blue component. The results are human body cross-sections on a clean black background.

During the slicing and scanning process, it was inevitable for human error such as incorrect alignment between sequential cryosection images. Fortunately the Visible Female data sets are properly aligned images so that we don't need to perform alignment by ourselves. For the Asian male data set, we have implemented auxiliary applications for alignment using C++ and a light-weight cross platform GUI SDK, Fox-Toolkit.

2.3 Human Parts Segmentation

2.3.1 Image Segmentation

Image segmentation is the first step in image analysis and pattern recognition. A critical and essential component of image analysis and/or pattern recognition, it is one of the most difficult tasks in image processing, and determines the quality of the final result of analysis. Image segmentation is a process of dividing an image into different regions such that each region is, but the union of any two adjacent regions is not, homogeneous [13].

While many papers and several surveys have focused on monochrome image segmentation techniques, color images segmentation also has attracted more and more attention. Quite often the objects cannot be extracted using gray scale but can be extracted using color information. Compared to gray scale, color provides information in addition to intensity. Color is useful, even necessary, for pattern recognition and computer vision. Also the acquisition and processing hardwares for color images have become more available and accessible to deal with the computational complexity caused by the high-dimensional color space. Hence, color image processing has become increasingly more practical [13].

The literature on color image segmentation is at present not as extensively as that of monochrome image segmentation. Most published results of color image segmentation are based on gray level image segmentation approaches with different color representations [13]. Most gray level image segmentation techniques can be extended to color images, such as thresholding, clustering, region growing, edge detection, fuzzy approaches and neural networks. Gray level segmentation methods can be directly applied to each component of a color space, and the results can be then combined in some way to obtain a final segmentation result [14]. However, one of the problems is how to employ the color information as a whole for each pixel. When the color is projected onto three components, the color information is so scattered that the color image becomes simply a multispectral image and the color information that humans can perceive is lost. Another problem is how to choose the color representation for segmentation. Commonly used color representations include RGB, YIQ, YUV, HIS, CIE$L^*u^*v^*$ and CIE$L^*a^*b^*$. Each color representation has its advantages and disadvantages. No single color representation can surpass others for segmenting all kinds of color images [13].

2.3.2 Common Image Segmentation Methods

Image Segmentation Based on Histogram Thresholding

Histogram thresholding is widely used technique for gray level image segmentation. The assumption behind histogram-threshold based image segmentation techniques is that the histogram of images could be separated into a number of peaks, each corresponding to one region in images, and a threshold value corresponds to the valley between two adjacent peaks.

Typically, a threshold interval is defined by an average value a and a tolerance e [15]. All pixel intensity values $I(p)$ that lie within the resulting interval $[a-e, a+e]$ are selected. The result is a binary image. In the next step, a connected component analysis may be performed on the binary image to detect connecting regions and finally generate a segmentation of the target structure. A connected component analysis initializes a first component with the first pixel. The algorithm recursively looks for adjacent pixels in the binary image and adds them to this component. If no more connected pixels are found, and there are still pixels that have not been visited, a new component is initialized. This process terminates when all pixels are processed and assigned to one region. Usually these regions are conveyed to the user by assigning individual colors [15].

Threshold based segmentation can be extended to using multiple intensity intervals

$D_1, D_2, \ldots D_n$ [15]. On color images, multiple-histogram based thresholding divides color space by thresholding each component histogram.

While manual adjustment of thresholds is common, in most cases it should be avoided if possible. In addition to taking time and being incompatible with automatic processing, different results are likely to be obtained at different times or by different people. Manual thresholding errors are probably responsible for more problems in subsequent image analysis than any other cause [16].

A number of algorithms have been developed for automating the thresholding procedure. Probably the simplest method of all is to locate the peaks in the histogram and set the thresholds midway between them. This approach is robust, because the peaks usually have well-defined shapes and easily found positions. However, the method is not very accurate at defining the actual structures present as there is no particular reason to expect the midpoint to correspond to the boundary between regions [16].

The Trussell algorithm [17] is probably the most widely used automatic method because it usually produces a fairly good result and is easy to implement. It finds the threshold setting that produces two populations of pixels (brighter and darker) with the largest value of the Student's t-statistic, which is calculated from the means of the two groups (μ), their standard deviations(σ), and the number of pixels (n) in each.

Clustering Based Image Segmentation

Clustering is the process of partitioning a set of objects into subsets of similar objects called clusters. Clustering is often seen as an unsupervised classification of pixels. Normally a priori knowledge is not required during the clustering process. On color images, colors create dense clusters in the color space in a natural way. One of the most popular and fastest clustering techniques is the k-means technique [18].

The k-means method was proposed in the 1960s [19]. The first step of this technique requires determining a number of clusters k and choosing initial cluster centers C_i:

$$C_1, C_2, \ldots C_k \; where \; C_i = [R_i, G_i, B_i], \; i = 1, 2, \ldots k \tag{2.1}$$

During the clustering process, each pixel x is allocated cluster K_j with the closest cluster center using a predefined metric such as the Euclidean metric and the city-block metric. The main idea of k-means is to change the positions of cluster centers as long as the sum of distances between all the points of clusters and their centers will be minimal. After each allocation of the pixels, new positions of cluster centers are computed as arithmetical means. In the next step, a difference between new and old positions of the centers is checked. If the difference is larger than some threshold, then the next iteration is starting, and the distances from the pixels to the new centers, pixels membership, and so forth, are calculated. If the difference is smaller than threshold, then the clustering process is stopped [18].

Region Growing

Region-based techniques group pixels into homogeneous regions [18]. Region-based techniques include region growing, region splitting, region merging, and others. The region-growing technique, long ago proposed for gray-level images [20], is also popular in color image processing.

The region-growing technique is a typical bottom-up technique [18]. Neighboring pixels are merged into regions, if their attributes, i.e., colors are sufficiently similar. This similarity is often represented by a homogeneity criterion. If a pixel satisfies the homogeneity criterion, then the pixel can be included in the region, and the region attributes (a mean color, an area of region, etc.) are updated. The region-growing process, in its classical version, starts from chosen pixels called seeds and is continued as long as all the pixels are assigned to the regions. Each of these techniques varies in homogeneity criteria and methods of seed location. The advantages of region-growing techniques result from taking into consideration two important elements: color similarity and pixel proximity in the image. On RGB color images, the homogeneity criterion based on the Euclidean metric has the following form:

$$\sqrt{(R-\bar{R})^2+(G-\bar{G})^2+(B-\bar{B})^2} <= d \tag{2.2}$$

where, R, G, B are color components of the tested pixel; $\bar{R}$, $\bar{G}$, $\bar{B}$ are color components of the mean color of the creating region; and d is a very important parameter for segmentation results.

The region-growing process uses a four-connectivity or an eight-connectivity concept [18]. A pixel satisfying the requirements of four-connectivity or eight-connectivity is merged into the creating region, if its color fulfills the homogeneity criterion.

Seeded Region Growing In the 1990s, two versions of the seeded region growing (SRG) algorithm for grayscale images were proposed [21][22]. The seeds in this technique are often chosen in the region of interest (ROI). For complete segmentation of an image into N regions, one seed in each potential region should be chosen and a proper value of parameter d should be selected. For each image, there is a range of values of the parameter d that is necessary for good segmentation. The SRG technique with application of a manual location of seeds works properly if the image contains a limited number of objects [23].

The SRG technique results depend on the positions of seeds. The seeds can be located in the image manually, randomly, or automatically. An operator uses his or her knowledge of the image for seed location. The random choosing of seeds is particularly risky in the case of a noisy image, because a seed can also be located on a noisy pixel. The seeds can also be found using the color histogram peaks. Additional edge information is sometimes applied to the location of seeds inside closed contours. Sinclair proposed a position of seeds on the peaks in the Voronoi image [24]. His method primarily requires that edges in the original image be found. A binarized edge image is a base for generating the Voronoi image. The gray level in the Voronoi image is the function of distance from the pixel to the closest image. The larger this distance, the brighter the pixel in the Voronoi image. Therefore, the seeds are located in the brightest pixels of a Voronoi image.

Unseeded Region Growing Another set of region growing techniques is based on the concept of region growing without requiring the seeds to start the segmentation process [18]. At the beginning of the algorithm, each pixel has its own label (one-pixel regions). A pixel is included in a region if it is four-connected or eight-connected to this region and has a color value in the specified range from the mean color of an already constructed region. After each inclusion of the pixel in the region, the region's mean color is updated. This updating process uses recurrent formulas. One or two simple raster scans of the image are applied: One pass from the left to the right and from the top to the bottom can be followed by an additional reverse pass over the image [25]. In the case of four-connectivity, four neighbors of the tested pixel are checked during both scans. In the case of eight-connectivity, eight neighbor pixels are checked during scanning. The pixel aggregation process results in a set of regions characterized by their mean colors, sizes and lists of pixels that belong to proper regions. The regions in this process are generated sequentially.

SRG techniques are very useful for cases where an image should be segmented to a small number of regions. On the other hand, unseeded region growing is suitable for application in the complete image segmentation.

Watershed Segmentation

Watershed segmentation is another region-based method that has its origins in mathematical morphology [26]. The general concept was introduced by Digabel and Lantuejoul [27]. A breakthrough in applicability was achieved by Vincent and Soille [28], who presented an algorithm that is orders of magnitudes faster and more accurate than previous ones. Watershed segmentation has been widely applied to a variety of medical image segmentation tasks and can now be regarded as a general and important segmentation method [15].

Watershed segmentation is based on the idea of regarding an image as a topographic landscape with ridges and valleys [15]. The elevation values of the landscape are typically defined by the intensity values or gradient magnitude of the respective pixels. Based on such representation, the watershed transform decomposes an image into regions called catchment basins [15]. For each local minimum, a catchment basin catches all points whose path of steepest descent terminates at this minimum. Watersheds are the border lines that separate basins from each other. The watershed transformation decomposes an image completely so that each pixel is either assigned to a region or a watershed. With noisy medical image data, typically a large number of small regions arise and causes the over-segmentation problem, since these regions are generally much smaller than the anatomical target structure.

Livewire Segmentation

Livewire denotes an edge-based segmentation method. Introduced by Mortensen et al. [29] and Udupa et al. [30], it is also referred to as Intelligent Scissors. As a result of livewire segmentation, the contours of the target structure in each slice of a three-dimensional data set are available.

Livewire is based on the definition of a cost function, and it selects paths with minimal costs (movement along boundaries with strong gradients generally have low costs) [15]. Minimal cost paths are computed by Dijkstra's graph search algorithm [31]. For this purpose, an image is represented as a graph, with vertices representing image pixels and edges representing costs of connections between neighboring pixels. The edges are directed and the orientation directly opposes to each other. The inside of a directed edge is considered to be the left of this edge. Costs are assigned to every directed edge [32]. Intensity to the left and to the right, gradient magnitude and direction, and the Laplacian zero crossing (the approximated second-order derivative of the image data) may be part of the cost function computation.

A general cost function for the computation of the local cost of an edge connecting the pixels p and q is thus represented as:

$$l(p,q) = w_z \cdot f_z(q) + w_g \cdot f_g(q) + w_d \cdot f_D(p,q). \tag{2.3}$$

The function is a weighted sum of different components: the Laplacian zero crossing $f_z(q)$, which indicates proximity of the pixel q to an edge; $f_g(q)$, which represents the gradient magnitude at pixel q; and $f_D(p, q)$, which represents the gradient direction. The latter term is employed as a smoothness constraint: high costs are associated with sharp changes in the boundary direction. The weights w_z, w_g, and w_d can be adapted to the properties of the segmentation target object.

In livewire segmentation, costs are computed from a user-selected control point to the current mouse position. The contour associated with the lowest accumulated cost is highlighted. Ideally, the resulting path wraps around the target object. The success of this segmentation method critically depends on the suitability of the cost function. In general, the cost function considers gray values of the relevant object (mean value and standard deviation), gray values of surrounding objects, strength of image gradients, and second-order derivatives in the gradient direction. Each factor might be assigned a certain weight that determines its influence [15].

Livewire is a segmentation approach operated in a slice-oriented way. However, when segmenting larger three-dimensional objects across multiple slices, the livewire approach is still rather laborious [15].

Level Sets

If region growing is considered as a dynamic process, the processing boundary can be regarded as wave front propagating through the target object [15]. This interpretation is the basis for the level sets method [33]. The wave propagation is guided by image features such as image intensity and gradient. Level sets are considered an implicit formulation of deformable models. However, the contour is not manipulated directly. Instead, the contour is embedded as the zero level set in a higher dimensional function, the level set function $\psi(X, t)$. The parameters are specified in such as way that progression toward the target object's boundaries slows. More precisely, the level set function is evolved under control of a partial differential equation. The evolving contour can be determined by extracting the zero-level set.

$$\Gamma((X), t) = \psi(X, t) = 0 \tag{2.4}$$

Applying the equation determines all points at height 0 of the embedding function. Level set segmentation handles complex anatomical shapes with arbitrary topology. There are many applications of level set methods for images segmentation. As an example, Avants and Williams [34] and Van Bemmel et al. [35] segment vascular structures with level sets. The segmentation of layered anatomical structures, such as the cortex, can be performed well with level sets [36].

2.3.3 Semi-Automatic Segmentation on Cryosection Images

On cryosection images, however, none of above segmentation methods is able to perform automatic segmentation. Thus, semi-automatic segmentation is performed in existing systems, such as the VOXEL-MAN framework. For accurate human parts segmentation, we apply the Livewire method on cryosection images. The Livewire method is based on edge detection. The result of the Livewire method is an outline of human parts boundary.

Using a pen graphic tablet, we move the cursor on human part boundaries. For one human part, we start from one pixel on its boundary and move the cursor on the boundary counter clock. Livewire tracks human part boundary between last cursor position and current cursor position. When we move back to the starting pixel, we get a closed contour of the human part boundary. We then fill the human part using pre-specified artificial colors, resulting a color mask image on which each human part is masked by one artificial color. Table 2.1 lists all 30 different colors used in VHASS for segmentation and corresponding human part names.

2.3.4 Segmentation Index Volumes

To create surface models for segmented human parts, we create *segmentation index volumes*, or label map, from color mask images. In the volume, we assign different values $v \in (1, \ldots, n)$ to different segmented parts so that each segmented part has unique segmentation

Table 2.1: Artificial Colors in Human Parts Extraction

Human Parts	**Color (R, G, B)**
Skin	140 30 30
Bone	200 250 250
Maxilla	40 15 250
Zygomatic bone	45 140 30
Empty space in human body	1 1 1
Muscles	220 60 230
Systemic arteries	230 50 10
Venae cavae	30 200 50
Portal vein	20 250 200
Spinal cord, gray matter, and lymph	250 200 50
Liver	210 150 250
Gall bladder and bile duct	180 250 100
Glands	180 125 50
Spleen	250 120 120
Oesophagus, stomach, and duodenum (superior part)	160 130 205
Urethra	30 230 250
Lung	250 200 250
Larynx and bronchia	160 250 250
Kidney and urinary bladder	250 210 135
Reproductive glands	20 110 40
White matter	240 230 230
Eye	240 160 80
Heart	230 150 10
Pulmonary vein	250 90 100
Pulmonary artery	30 170 100
Lacrimal ducts	230 100 70
Duodenum (descending and horizontal part)	110 90 240
Colon	30 40 230
Teeth	45 50 150
Fascia	240 250 200

index. We reserve 0 for blank voxels (empty spaces between segmented human parts). A segmentation index volume can store multiple human parts at the same time. Surface models of human parts are created at run time from segmentation index volumes.

The color mask images take the same storage space as the original cryosection images. We have developed an utility application that allows users to create segmentation index volumes from subsets of the entire color mask image stack. Users can also choose different down-sampling rate to create segmentation index volumes with different resolutions. Another function of the utility application is to extract color volumes from the original cryosection volume. Generally, the extracted color volumes and the segmentation index volumes match to have the same bounding box in physical space. Color volumes are used to define colors and textures of human parts.

Chapter 3: Photo-Realistic Virtual Human Parts via Direct Volume Rendering

This chapter is organized as follows. In the first section, we introduce direct volume rendering techniques. The second section focuses on GPU (Graphics Processing Unit) accelerated direct volume rendering. Our new opacity transfer functions are outlined in the third section. Finally, our implementation of GPU ray casting volume rendering on unsegmented cryosection images is discussed in the fourth section.

3.1 Volume Rendering

Traditionally, in computer graphics, three-dimensional objects are represented using surface representations such as polygonal meshes or parameterized surface patches. The visual properties of surfaces, such as color and reflectance, are modeled by means of a shading algorithm such as the Phong model [37] or anisotropic bidirectional reflectance distribution function. In these methods, light transport is evaluated only at points on the surface; thus light interaction in the atmosphere or in the interior of an object is not taken into account [38].

On the contrary, volume rendering [39][40] generates images from three-dimensional scalar data. Volume rendering techniques are originally motivated by scientific visualization, where volume data is acquired by measurement or numerical simulation of natural phenomena. Typical examples are medical data of the interior of the human body obtained by computed tomography (CT) or magnetic resonance imaging (MRI). Other examples are computational fluid dynamics (CFD), geological and seismic data, as well as abstract mathematical data such as three-dimensional probability distributions of pseudo-random numbers [38]. Many artists and researchers have generated volume data synthetically to supplement surface models, i.e., procedurally [41], which is especially useful for rendering high-quality special effects [38].

3.1.1 Volume Data

Discrete volume data set can be treated directly as a simple three-dimensional array of cubic elements (voxels[1])[42], each representing a unit of space [38]. However, people have found that it is more appropriate to identify each voxel with a sample obtained at a single infinitesimally small point from a continuous three-dimensional signal, represented thus:

$$f(\vec{v}) \in \mathbb{R} \ with \ \vec{v} \in \mathbb{R}^3. \tag{3.1}$$

On a continuous signal with a cut-off-frequency f_c (the signal contains no frequencies higher than f_c), the Nyquist-Shannon sampling theorem allows exact reconstruction, if the single is evenly sampled at more than twice the cut-off-frequency. However, two major problems prohibit the ideal reconstruction of sampled volume data in practice [38].

First, according to sampling theory, ideal reconstruction requires the convolution of the sample points with a *sinc* function in the spacial domain. For the one-dimensional case, the *sinc* function reads

$$sinc(x) = \frac{sin(\pi x)}{\pi x}. \tag{3.2}$$

The three-dimensional version of this function is obtained by tensor-product ($\otimes$). Since the function has infinite extent, for exact reconstruction of the original signal at an arbitrary position all the sampling points must be considered, not only those in a local neighborhood. In practice this is computationally infeasible.

Second, generally the real data is not limited with a cut-off-frequency. Sharp boundaries between different materials represent infinite extent step functions in the frequency domain. Sampling and reconstruction of a signal which is not band-limited will produce aliasing

---

[1] volume elements

artifacts [38].

In practice, in order to reconstruct a continuous signal from discrete voxels, the ideal three-dimensional *sinc* filter is usually replaced by either a box filter or a tent filter. The box filter calculates nearest-neighbor interpolation resulting in sharp discontinuities between neighboring cells and produceing a block appearance. On the other hand, trilinear interpolation, achieved by convolution with a three-dimensional tent filter, represents a good trade-off between computational cost and smoothness of the output signal [38].

3.1.2 Direct Volume Rendering

Direct Volume Rendering methods display the volume data by evaluating an optical model that describes how the volume emits, reflects, scatters, absorbs, and occludes light [43]. The scalar value of the volume data is virtually mapped to physical quantities which describe light interaction at the respective point in three-dimensional space [38]. The mapping process is named classification and is usually performed using a transfer function. A transfer function can be a simple ramp, a piecewise linear function, or an arbitrary table. To composite the final image, the light propagation is computed by integrating light interaction effects along viewing rays based on the optical model. The corresponding integral is known as the volume rendering integral.

Optical Models

Direct volume rendering algorithms regard volume data as a distribution of light-emitting particles of a certain density. These densities are mapped to RGBA quadruplets using transfer functions for compositing along the viewing ray. A physical-based optical model is used to specify the compositing process. The most important optical models for direct volume rendering are described in a survey paper by Nelson Max [43]. We briefly summarize these models here [38]:

- **Absorption only**. The volume is assumed to consist of black particles that absorb all the light that impinges on them. These particles do not emit or scatter light.

- **Emission only**. The volume is assumed to consist of particles that emit but do not absorb any light.

- **Absorption plus emission**. This optical model is the most common one in direct volume rendering. Particles emit light, and occlude, absorb incoming light. However, there is no scattering or indirect illumination.

- **Scattering, shading and shadowing**. This model includes scattering of illumination that is external to a voxel. Light can either be assumed to impinge unimpeded from a distant light source, or it can be shadowed by particles between the light and the voxel under consideration.

- **Multiple scattering**. This sophisticated model includes support for incident light that has already been scattered by multiple particles.

The Volume Rendering Integral

The most basic volume rendering algorithm, ray-casting, is the most direct numerical method for evaluating the volume rendering integral. The ray-casting is a process that, for each pixel in the image to render, casts a single ray from the eye through the pixel's center into the volume, integrating the optical properties obtained from the encountered volume densities along the ray [38].

On discrete volume data, the evaluation of the volume rendering integral is approximated numerically. The integral is usually substituted by a Riemann sum. We denote a ray cast into the volume $\vec{x}(t)$, and parameterize it by the distance t from the eye. The scalar value corresponding to a position along the ray is denoted by $v(\vec{x}(t))$. Using the emission-absorption model, the volume rendering equation integrates *absorption coefficients* $\kappa(v)$ (accounting for the absorption of light), and *emissive colors* $r(v)$ (accounting for radiant

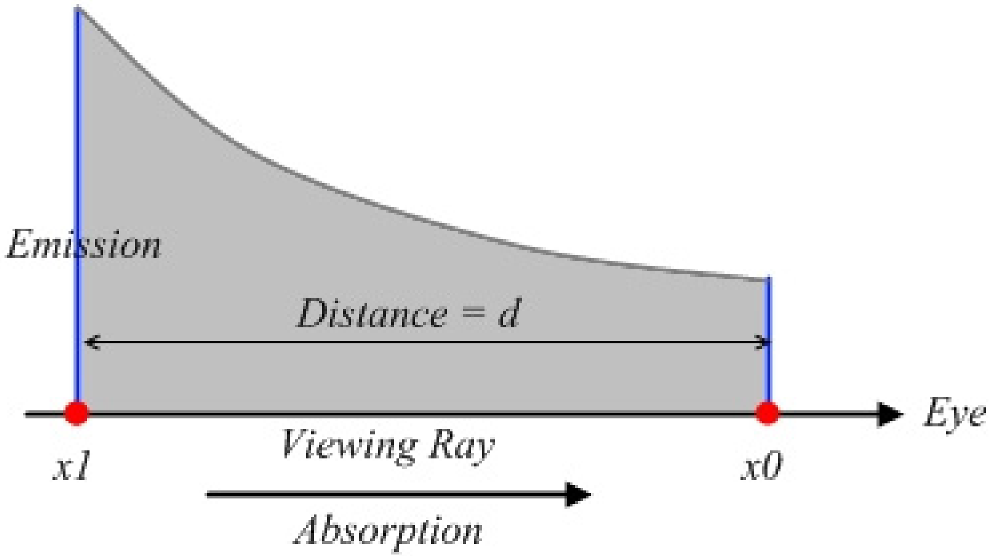

Figure 3.1: Emission and Absorption in Volume Data

energy actively emitted) along a ray [38]. For simplicity, we denote emission r and absorption coefficients κ as functions of the eye distance t instead of the scalar value v:

$$r(t) \;:=\; r(v(\vec{x}(t)) \; and \; \kappa(t) \;:=\; \kappa(v(\vec{x}(t))). \tag{3.3}$$

Figure 3.1 illustrates the emission and absorption process in volume data. An amount of radiant energy, emitted at position x_1 along the viewing ray, is continuously absorbed until it reaches the eye. This means that only a portion r' of the original radiant energy r emitted at x_1 will eventually reach the eye. Given the distance between x_0 and x_1 to be d, if there is a constant absorption $\kappa = const$ along the ray, r' is calculated using the following equation.

$$r' \;=\; r \cdot e^{-\kappa d}. \tag{3.4}$$

When the absorption κ is not constant along the ray, but itself depends on the position, then the amount of radiant energy r' reaching the eye is computed by integrating the absorption coefficient along the distance d:

$$r' \ = \ r \cdot e^{-\int_0^d \kappa(\hat{t})d\hat{t}}. \tag{3.5}$$

The integral over the absorption coefficients in the exponent,

$$\tau(d_1,\ d_2) \ = \ \int_{d_1}^{d_2} \kappa(\hat{t})d\hat{t}. \tag{3.6}$$

is also called the *optical depth* [38]. The total amount of radiant energy R reaching the eye from the viewing direction is calculated by taking into account the emitted radiant energy from all possible positions t along the ray:

$$R \ = \ \int_0^{\infty} r(t) \cdot e^{-\tau(0,\, t)}dt. \tag{3.7}$$

On discrete volume data, the integral is evaluated numerically through either back-to-front or front-to-back compositing of samples along the viewing ray. In the next section, we use the *ray-casting* method to illustrate the evaluation process.

Ray-Casting

The ray-casting direct volume rendering algorithm uses a straightforward numerical evaluation of the volume rendering integral [39]. For each pixel of the final image, a single ray is cast into the volume. At equally spaced intervals along the ray, the discrete volume data is resampled with tri-linear interpolation. At each resampling location, the scalar values of eight neighboring voxels are weighted according to their distance to the actual location for which a data value is needed. After resampling, the scalar data value is mapped to optical properties via a lookup table to get an RGBA color that includes the corresponding emission and absorption coefficients [39] for the location. The solution of the volume rendering integral is then approximated via alpha blending in either back-to-front or front-to-back order [38].

The optical depth τ (Equation 3.6), which is the cumulative absorption up to a certain position $\vec{x}(t)$ along the ray, can be approximated by a Riemann sum

$$\tau(0,\, t) \;\approx\; \tilde{\tau}(0,\, t) \;=\; \sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t)\, \Delta t, \tag{3.8}$$

with Δt denoting the distance between successive resampling locations. The summation in the exponent can immediately be substituted by a multiplication of exponential terms:

$$e^{-\tilde{\tau}(0,\, t)} \;=\; \prod_{i=0}^{\lfloor t/\Delta t \rfloor} e^{-\kappa(i \cdot \Delta t)\Delta t}. \tag{3.9}$$

Here we introduce *opacity* A, well-known from alpha blending, by defining

$$A_i \;=\; 1 \;-\; e^{-\kappa(i \cdot \Delta t)\Delta t}, \tag{3.10}$$

and rewriting equation 3.9 as:

$$e^{-\tilde{\tau}(0,\, t)} \;=\; \prod_{i=0}^{\lfloor t/d \rfloor} (1 \;-\; A_j). \tag{3.11}$$

This allows opacity A_i to be used as an approximation for the absorption of the i-th ray segment, instead of absorption at a single point.

Similarly, the emitted color of the i-th ray segment can be approximated by:

$$R_i \;=\; r(i \cdot \Delta t)\Delta t. \tag{3.12}$$

With the above approximated emissions and absorptions along a ray, we can state the approximate evaluation of the volume rendering integral as (denoting the number of samples by $n = \lfloor t/\delta t \rfloor$):

$$\tilde{R} = \sum_{i=0}^{n} R_i \prod_{j=0}^{i-1} (1 - A_i). \tag{3.13}$$

Equation 3.13 can be evaluated iteratively by alpha blending in either back-to-front or front-to-back order.

Alpha Blending

Back-to-front Alpha Blending In back-to-front alpha blending, Equation 3.13 is computed iteratively by stepping i from $n-1$ to 0:

$$R'_i = R_i + (1 - A_i) R'_{i+1}. \tag{3.14}$$

A new value R'_i is calculated from the color R_i and opacity A_i at the current location i, and the compositing color R'_{i+1} from the previous location $i+1$. The starting condition is $R'_n = 0$.

Front-to-back Alpha Blending In front-to-back alpha blending, we use the following alternative iterative formation to evaluate Equation 3.13 by stepping i from 1 to n:

$$R'_i = R'_{i-1} + (1 - A'_{i-1}) R_i, \tag{3.15}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1}) A_i. \tag{3.16}$$

New values R'_i and A'_i are calculated from the color R_i and opacity A_i at the current location i, and the compositing color R'_{i-1} and opacity A'_{i-1} from the previous location $i-1$. The starting condition is $R'_0 = 0$ and $A'_0 = 0$.

Note that in all blending equations, we use *opacity-weighted colors* [44], also known as *associated colors* [45]. An opacity-weighted color denotes a color that has been premultiplied by its associated opacity. It can be shown that interpolating color and opacity separately leads to artifacts, whereas interpolating opacity-weighted colors achieves correct results [44].

3.2 GPU Volume Rendering

3.2.1 Programmable GPUs

Programmable GPU was introduced with configurable rasterization and vertex processing in late 1999. Prominent examples are Nvidia's *register combiners* or ATI's *fragment shader* OpenGL extensions. Back then, it was not easy to access these vendor-specific features in a uniform way.

Today's graphics processors are capable of true programmability, meaning that user-written micro-programs can be uploaded to graphics memory and executed directly by the geometry processing stage (*vertex shaders* and *geometry shaders*) and the rasterization unit (*fragment* or *pixel shaders*).

Vertex Shaders

Vertex shaders are user-written programs that substitute major parts of the fixed-function computation of the geometry processing unit. They allow customization of the vertex transformation and the local illumination model. The vertex shader is executed once per vertex. Every time a vertex enters the pipeline, the vertex processor receives an amount of data, executes the vertex shader and writes the attributes for exactly one vertex. The vertex shader can not create vertices from scratch or remove incoming vertices from the pipeline.

Geometry Shaders

A geometry shader can generate new graphics primitives, such as points, lines, and triangles, from those primitives sent to the beginning of the graphics pipeline. Geometry shaders run after vertex shaders and before pixel shaders. They take input as a whole primitive. For example, when operating on triangles, input of the geometry shader will be the three vertices compositing one triangle. The shader can then emit zero or more primitives that are ultimately passed to a pixel shader.

Pixel Shaders

Pixel shaders are user-written programs executed by the rasterization unit. Used to compute the final color and depth values of a pixel, the pixel shader is executed once per pixel. Every time that polygon rasterization creates a pixel, the pixel processor receives a fixed set of attributes, such as colors, normal vectors or texture coordinates; it executes the pixel shader and writes the final color and z-value of the pixel to the output registers.

3.2.2 Volume Rendering via Texture Mapping

As mentioned in Section 3.1, the two major operations in volume rendering are interpolation and compositing, both of them can be efficiently done on modern graphics hardware. Texture mapping operations basically interpolate a texture image to obtain color samples at locations that do not coincide with the original volume grid. Thus, texture mapping operations in modern graphics hardware are ideal for performing repetitive resampling tasks. On the other hand, pixel operations in modern graphics hardware can be used to composite individual samples.

Volume rendering using texture mapping is an object-order approach that divides the object into primitives and then calculates which set of pixels are influenced by a primitive. This is done by rendering a proxy geometry with interpolated texture coordinates (usually comprised of slices rendered as texture-mapped quads), and by compositing all the parts (slices) of this proxy geometry from back to front via alpha blending [38]. The volume data is stored in a stack of two-dimensional texture images or a single three-dimensional texture. In case of two-dimensional texture image stack, each texture in the stack corresponds to an axis-aligned slice through the volume. In a single three-dimensional texture image, each texel corresponds to a single voxel. If the volume is too large to fit into texture memory, it needs to be split into several smaller three-dimensional textures [38].

Texture-mapping based volume rendering generally has good performance. High texture memory bandwidth and built-in interpolation methods on graphics hardware result in a fast resampling process. Moreover, high performance rasterization units on graphics hardware ensure a fast compositing process. A rasterization unit converts the two-dimensional image space scene representation into raster format and determines the correct resulting pixel colors. Depending on the way the proxy geometry is generated, there are several different texture-mapping based volume rendering methods. In the following sections, we will introduce the concept of proxy geometry and different texture-mapping based volume rendering methods one by one.

Proxy Geometry

Since graphics hardware does not support any volumetric rendering primitives, we need to convert our volumetric representation into rendering primitives supported by graphics hardware. A set of graphics hardware supported primitives representing the volumetric object is called a proxy geometry.

The conceptually simplest example of proxy geometry is a set of view-aligned slices (quads that are parallel to the viewport, usually also clipped against the bounding box of the volume), with three-dimensional texture coordinates that are interpolated over the interior of these slices and ultimately used to sample a single three-dimensional texture map at the corresponding locations [38]. For view-aligned slices, three-dimensional texture mapping is mandatory, since a single slice would have to fetch data from several different two-dimensional textures.

If the proxy geometry is aligned with the original volume data, texture-fetch operations for a single slice can be generated to stay within the same two-dimensional texture. In this case, the proxy geometry is comprised of a set of object-aligned slices.

Two-dimensional Textured Object-Aligned Slices

In this case, the proxy geometry is a stack of planar slices, all of which are required to be aligned with one of the major axes of the volume (either the x, y, or z axis), mapped with two-dimensional textures, which in turn are resampled by the bi-linear interpolation on graphics hardware [46]. When a slice is rendered, only two dimensions are available for texture coordinates since only two-dimensional texture mapping is used. Thus, slices must be aligned with a major axis. In rendering, the third coordinate selects the texture to use from the stack of slices, and the other two coordinates become the actual two-dimensional texture coordinate for the slice. Rendering proceeds from back to front, blending one slice on top of the other.

Although a single stack of 2D slices is able to store the entire volume, one slice stack does not suffice for rendering. When the viewer rotates about the object, it would be possible that imaginary viewing rays pass though the object without intersecting any slices polygons. This can not be prevented with only one slice stack. The solution for this problem is to store three slice stacks, one for each of the major axes. During rendering, the stack with slicing direction most parallel to the viewing direction is chosen.

Artifacts are visible when the slice stack in use is switched from one stack to the next because the actual locations of sampling points change abruptly when the stacks are switched. Another drawback is that an extra amount of texture memory is used to store three slice stacks.

Two-dimensional Slice Interpolation

For two-dimensional textured object-aligned slices, no interpolation between slices is performed. Only bi-linear interpolation is used within each slice. As a result, artifacts are visible when there are too few slices, because the sampling frequency is too low with respect to frequencies contained in the volume.

On the graphics hardware supporting multi-texturing, we can perform on-the-fly inter-slice interpolation by binding two textures simultaneously instead of just one when rendering the current slice. Linear interpolation is performed between the two bound textures [47]. To do this, we specify fractional slice positions, where the integers correspond to slices that actually exist in the source slice stack, and the fractional part determines the position between two adjacent slices. The number of rendered slices is now independent of the number of slices in the volume, and can be adjusted arbitrarily.

For each slice to be rendered, two textures are activated, which correspond to the two neighboring original slices from the source slice stack. The fractional position between these slices is used as a weight for the inter-slice interpolation. Standard bi-linear interpolation is employed for each of the two neighboring slices, and the interpolation between the two obtained results achieves tri-linear interpolation.

Three-dimensional Textured View-Aligned Slices

Three-dimensional textured view-aligned slices are the simplest type of proxy geometry. In this case, the volume is stored in a single three-dimensional texture map, and three-dimensional texture coordinates are interpolated over the interior of the proxy geometry polygons. These texture coordinates are then used directly for indexing the three-dimensional texture map at the corresponding location, and thus resampling the volume.

Three-dimensional textured view-aligned slices can be oriented arbitrarily within the three-dimensional texture map. Thus, it is natural to use slices aligned with a viewport. In this case such slices closely mimic the sampling used by the ray-casting algorithm. The number of slices can also be adjusted on-the-fly. The graphics hardware performs tri-linear interpolation within the volume for each resampling location.

Three-dimensional Textured Spherical Shells

Proxy geometry that uses planar slices has the basic problem that the distance between successive samples used to determine the color of a single pixel is different from one pixel to the next in the case of perspective projection.

When incorporating the sampling distance in the numerical approximation of the volume rendering integral, this pixel-to-pixel difference can not easily be accounted for. A possible solution to this problem is the use of spherical shells instead of planar slices [48]. In order to attain a constant sampling distance for all pixels using perspective projection, the proxy geometry has to be spherical, i.e., be comprised of concentric spherical shells. In practice, these shells are generated by clipping tessellated spheres against both the viewing frustum and the bounding box of the volume data.

Slabs

An inherent problem of using slices as proxy geometry is that the number of slices directly determines the sampling frequency, and thus the quality of the rendered result. When high frequencies are contained in the volume data, the required number of slices can be very high. Even though the majority of texture-based implementations allow the number of slices to be increased on demand via interpolation done entirely on the graphics hardware, the fill rate demands increase dramatically.

A solution is to use *slabs* instead of slices [49]. A slab is not a new geometrical primitive, but simply the space between two adjacent slices. During rendering, the solution of the integral of ray segments that intersect this space is properly accounted for by looking up a pre-computed solution. This solution is a function of the scalar values of both the back slice and the front slice. It is obtained from a pre-integrated lookup table stored as a 2D texture. Geometrically, a slab can be rendered as a slice with its immediately neighboring slice projected onto it.

3.2.3 GPU Ray Casting

Ray casting [39] is a well-known approach which represents a direct numerical evaluation of the volume rendering integral. For each pixel in the image, a single ray is cast into the volume. Then the volume data is resampled at discrete positions along the ray. Figure 5.3 illustrates ray casting.

The ray casting algorithm can be split into the following major parts. First, a light ray needs to be set up according to given camera parameters and the respective pixel position. Then the ray has to be traversed by stepping along the ray. Optical properties are accumulated in the traversing. Finally this process stops when the volume is traversed. Ray casting can easily incorporate a number of acceleration techniques. Early ray termination allows us to truncate light rays as soon as we know that volume elements further away from the camera are occluded. Ray traversal can be stopped when the accumulated opacity reaches a certain user-specified limit. The step size for one ray can be chosen independently from other rays. For example, empty regions can be completely skipped or uniform regions can be quickly traversed by using large step sizes.

Ray casting exhibits an intrinsic parallelism in the form of completely independent light rays. This parallelism can be easily mapped to GPU processing, for example, by associating the operations for a single ray with a single pixel. In this way, the built-in parallelism for a GPU's multiple pixel pipelines is used to achieve efficient ray casting. In addition, volume data and other information can be stored in textures and thus accessed with the high internal bandwidth of a GPU.

3.3 Opacity Estimation on Human Part Volumes

Color photographic volume data of human parts offers a challenge to traditional volume rendering techniques. Determining the opacity of each voxel of the data set is the problem. In traditional direct volume rendering, an image is produced from a volume of scalar data, using transfer functions from scalar value to color and opacity. The design of effective color and opacity transfer functions from scalar values has been the subject of substantial research over the past decades [50][51][52], with the design of the color transfer function (one-dimensional to three-dimensional mapping) often being much more difficult than the design of an effective opacity transfer function (one-dimensional to one-dimensional mapping). In contrast, volume rendering from photographic volume data sets reverses the difficulty of transfer function design. Photographic volume data sets need an opacity transfer function from vector color data to scalar opacity data (three-dimensional to one-dimensional). This is difficult because photographic volume data does not contain a clear mapping from the multi-valued color values to a scalar density or opacity. Moreover, traditional differential geometric tools, such as intensity gradients, density and Laplacians, are distorted by the nonlinear non-orthonormal color spaces that are the domain of the voxel values. On the other hand, the design of an appropriate color transfer function is not generally required for photographic datasince the color of each voxel is already known.

One way to visualize human parts photographic volume data is to segment human parts on cryosection volumes and assign different opacity values to different human parts [53]. Then human parts photographic volume data is rendered using standard ray-casting algorithm. On the other hand, researchers have developed opacity transfer functions on photographic volume data [54][55]. Instead of segmenting human parts on cryosection volumes, the opacity transfer functions highlight human parts boundary and rely on users to distinguish different human parts. The efficient of these transfer functions in highlighting human parts boundary directly affects the final rendering results. We have implemented existing opacity transfer functions on cryosection volumes. We also have developed a new series of opacity transfer functions to produce better visual results.

3.3.1 Color Distance Gradients Based Transfer Functions

Ebert et al. have developed opacity transfer functions based on color distance gradients, defined in the following section [54].

Color Distance Gradients

In color photographic volume data, every voxel location is mapped to a color represented by three separate values. In RGB color space, given $color\_vol[x][y][z]$, a volume of color data points, imagine its image generating function $C(x, y, z)$ such that for every voxel location $[x_i][y_i][z_i]$,

$$color\_vol[x_i][y_i][z_i] = C(x_i, y_i, z_i) = (R_i, G_i, B_i). \tag{3.17}$$

Usually the color difference is measured with respect to spatial location. For instance, consider the difference between a color voxel value $C(x_i, y_i, z_i)$ and the color from a neighboring location measured at a distance, h, along the x direction, $C(x_j, y_j, z_j)$, where $x_j = x_i + h, y_j = y_i$, and $z_j = z_i$. Then the color difference vector is expressed as:

$$C(x_i, y_i, z_i) - C(x_j, y_j, z_j) = C(x_i, y_i, z_i) - C(x_i + h, y_i, z_i) = (R_i, G_i, B_i) - (R_j, G_j, B_j). \tag{3.18}$$

The difference of two adjacent colors is itself a color vector:

$$(R_i, G_i, B_i) - (R_j, G_j, B_j) = (R_i - R_j, G_i - G_j, B_i - B_j). \tag{3.19}$$

As h is reduced and made infinitesimally small, the partial derivative of $C(x, y, z)$ with respect to x is,

$$\begin{aligned} \frac{\partial}{\partial x} C(x, y, z) &= \lim_{h \to 0} C(x, y, z) - C(x + h, y, z) \\ &= (\frac{\partial}{\partial x} R(x, y, z), \frac{\partial}{\partial x} G(x, y, z), \frac{\partial}{\partial x} B(x, y, z)). \end{aligned} \tag{3.20}$$

The combined partial derivatives of $C(x, y, z)$ in the x, y, and z direction results in $\nabla C(x, y, z)$, the gradient of $C(x, y, z)$. Since each partial derivative is a color vector, $\nabla C(x, y, z)$ is a tensor, a vector of three color vectors:

$$\nabla C(x,y,z) = (\frac{\partial}{\partial x}C(x,y,z), \frac{\partial}{\partial y}C(x,y,z), \frac{\partial}{\partial z}C(x,y,z)). \tag{3.21}$$

The magnitude of $\nabla C(x,y,z)$ is evaluated as its tensor dot product. This is accomplished by multiplying $\nabla C(x,y,z)$ with its transpose $\nabla C(x,y,z)^T$, resulting in Λ, the "magnitude matrix."

$$\Lambda = (\nabla C(x,y,z))(\nabla C(x,y,z)^T) = \nabla C(x,y,z) \cdot \nabla C(x,y,z). \tag{3.22}$$

The square roots of the diagonal elements of Λ reflects the color changes along the cardinal Cartesian directions and their influence on the color distance gradient magnitude.

$$\begin{aligned} grad.x &= \sqrt{\Lambda_{11}} = \sqrt{\frac{\partial}{\partial x}C(x,y,z) \cdot \frac{\partial}{\partial x}C(x,y,z)}, \\ grad.y &= \sqrt{\Lambda_{22}} = \sqrt{\frac{\partial}{\partial y}C(x,y,z) \cdot \frac{\partial}{\partial y}C(x,y,z)}, \\ grad.z &= \sqrt{\Lambda_{33}} = \sqrt{\frac{\partial}{\partial z}C(x,y,z) \cdot \frac{\partial}{\partial z}C(x,y,z)}, \end{aligned} \tag{3.23}$$

The vector quantity $(grad.x, grad.y, grad, z)$ is called the *color distance gradient*. This vector is not a true gradient since it does not capture absolute direction (sign ambiguities in each of the cardinal directions) nor does it capture the absolute magnitude of the matrix Λ. However, as an abstraction of the magnitude matrix, it can be gainfully used in the generation of opacity transfer functions [54].

On discrete voxel data, Ebert et al. use central difference to compute approximate derivative values.

$$\begin{aligned} grad.x &= color\_distance(color\_vol[x-1][y][z], color\_vol[x+1][y][z]), \\ grad.y &= color\_distance(color\_vol[x][y-1][z], color\_vol[x][y+1][z]), \\ grad.z &= color\_distance(color\_vol[x][y][z-1], color\_vol[x][y][z+1]), \end{aligned} \tag{3.24}$$

where, for two RGB colors $c1$ and $c2$,

$$color\_distance(c1, c2) = \sqrt{(c1.r - c2.r)^2 + (c1.g - c2.g)^2 + (c1.b - c2.b)^2}. \tag{3.25}$$

The Euclidean color space distance between colors is used as a scalar metric for the color difference. In RGB color space, this is not guaranteed to give an accurate measure of perceptual difference due to the nonlinearities of the color space, but no better metric is available. The color difference metric used is unsigned distance, giving the sizes but not direction of the difference. Because the difference metric is scalar, it captures the perceptibility of change in that region, but not the nature. Specifically, the metric is not sensitive to differences in particular color space components, only the total difference [54].

Opacity Transfer Functions

Based on the definition of the color distance gradients, Ebert et al. have developed three opacity transfer functions, discussed in following paragraphs.

Color Distance Gradient Magnitudes In volume rendering, it is often useful to accentuate the transitions or boundaries between regions that have contiguous colors, gray-levels, or other properties. Unlike the scalar gradient magnitude values that arise from gray-level volume data, the tensor gradients of color photographic data are not easily used to generate opacity values. Ebert et al. take the magnitude of the color distance gradient vector or $\|(grad.x, grad.y, grad.z)\|$ for opacity measurement. This measurement will make the transitions between regions more opaque when adjacent voxel colors are more distant [54]. The resulting opacity transfer function is:

$$voxel\_opacity = \|grad(voxel)\|. \tag{3.26}$$

where $grad(voxel)$ is the color distance gradient vector generated on color volume data.

Color Distance Gradient Dot Products (MAX) The color distance gradient magnitude only captures changes in the length of the color distance gradient vector, not changes in its spatial orientation. Since a gradient vector can be computed for each voxel, a normal and its corresponding tangent plane can be associated with each voxel, denoting the orientation of the tangent space that passes through that location. In areas of smoothly changing orientations, edge information is dominated by the color distance gradient magnitudes. However, in turbulent or highly textured regions, the orientation of nearby normal vectors becomes highly variable. The color distance gradient at a voxel and its angular difference with its 6-adjacent neighboring voxel gradients can be used to capture information regarding the relative smoothness or roughness of volumetric textural features. The angular separation between the color difference gradients of two neighboring voxels is inversely proportional to the dot product of their normalized color difference gradient vectors. Therefore, the following transfer function highlights areas of large angular variation of color difference gradient vectors:

$$voxel\_opacity = \max_{i=1}^{6}[(1 - (Grad(voxel) \cdot Grad(neighbor_i)) \times \|grad(voxel)\|], \tag{3.27}$$

where

$$Grad(voxel) = \frac{grad(voxel)}{\|grad(voxel)\|}.$$

This function highlights areas with the large color distance gradients having the largest angular variation with their neighboring voxels. This function is therefore useful for highlighting the largest color changes [54].

Color Distance Gradient Dot Products (MIN) The minimum angular color distance gradient difference enhances areas where there are large gradients but small angular color changes with one of the 6-adjacent neighbor voxels [54]. The following transfer function highlights oriented patterns (e.g., muscle fibers) within the color data:

$$voxel\_opacity = \max_{i=1}^{6}[(Grad(voxel) \cdot Grad(neighbor_i)) \times \|grad(voxel)\|]. \tag{3.28}$$

For transfer functions above, the following final opacity transfer function is used in the rendering process to allow control over the opaqueness of the volume:

$$final\_opacity = (voxel\_opacity \times scalar)^{exponent}. \tag{3.29}$$

The *scalar* variable allows the overall opaqueness of the volume to be controlled, while the *exponent* variable allows the sharpness of the opacity fall-off to be easily controlled (larger *exponent* produces sharper opacity transitions).

Figure 3.2, 3.3, and 3.4 shows rendering results of above transfer functions.

3.3.2 Gradient Vector Flow Based Transfer Functions

Wang et al. derive opacity transfer functions based on gradient vector flow [55]. Gradient vector flow is first used to describe the external force of the snake model [56]. Snake model, or active contour model [57], is a dynamic force model, which draws the closed curve or surface to the object edge or surface. The driving forces consist of internal forces and external forces:

$$E = \int_0^1 E_{int} + E_{ext} ds. \tag{3.30}$$

where E_{int} are internal forces that hold the curve together and keep the curve from bending too much. And E_{ext} are external forces that attract the curve toward the object boundaries.

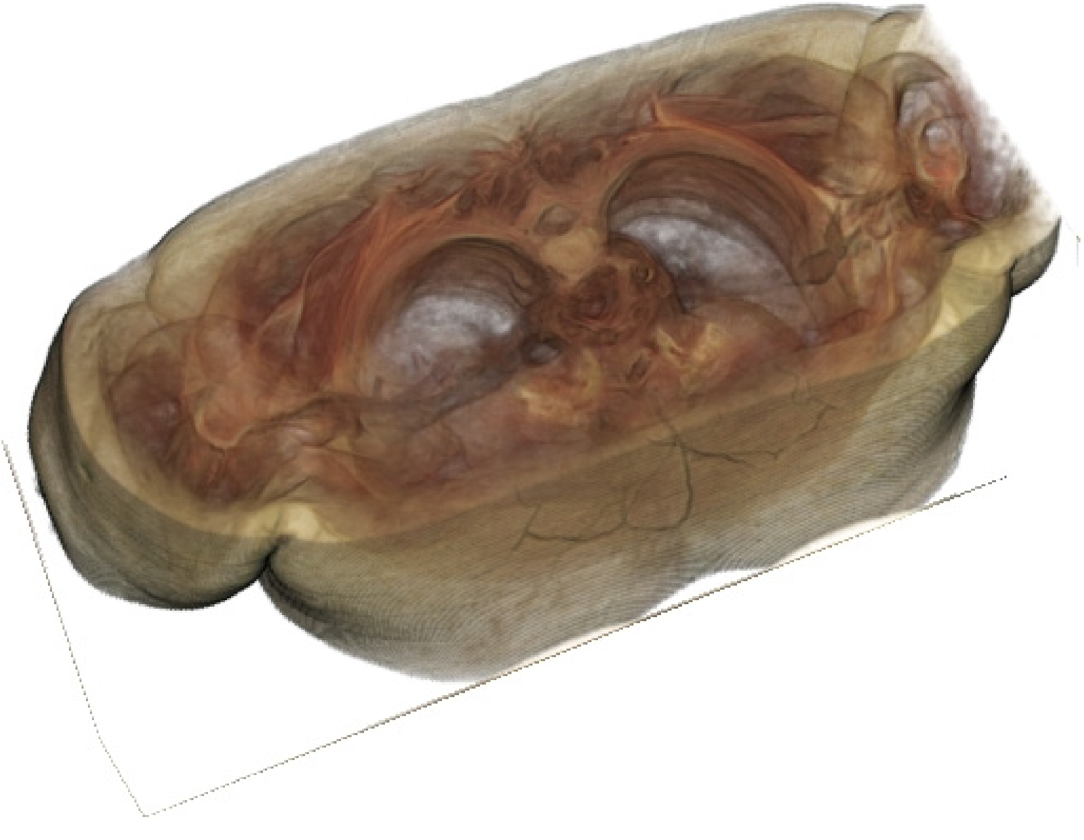

Figure 3.2: Color Distance Gradient Magnitudes ($scale = 0.5$, $exponent = 1.25$)

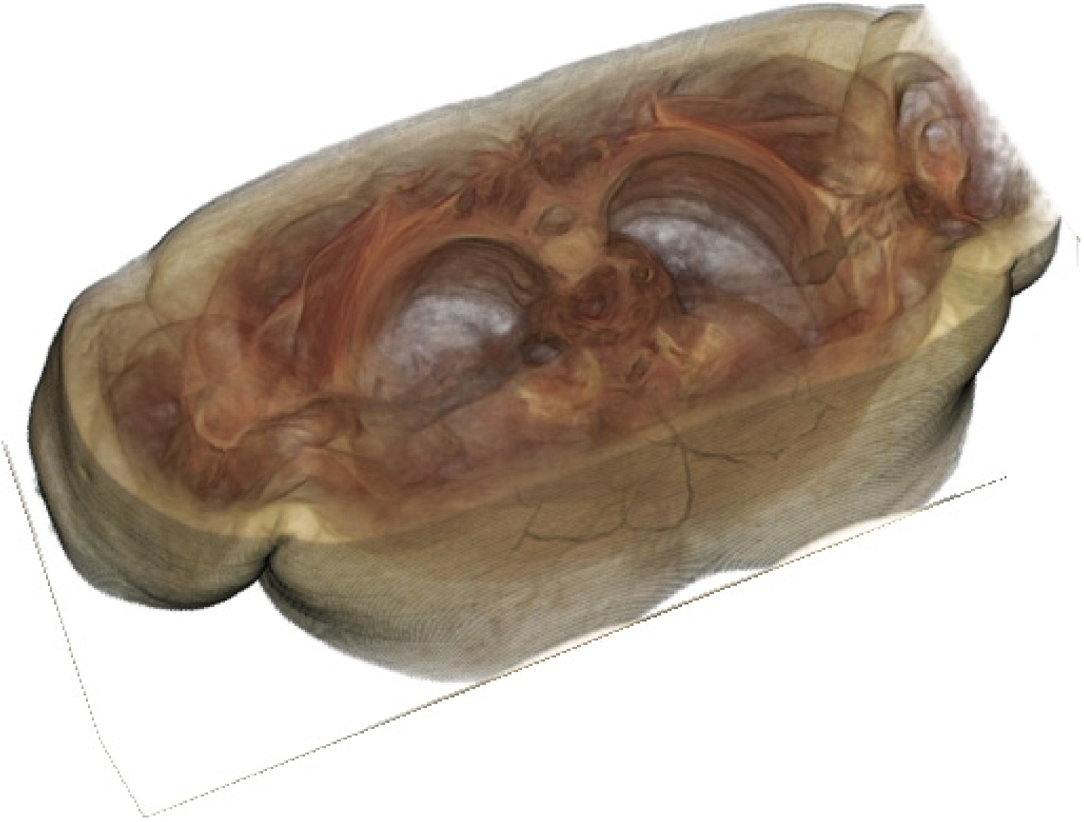

Figure 3.3: Color Distance Gradient Dot Products (MAX) ($scale$ = 0.5, $exponent$ = 1.25)

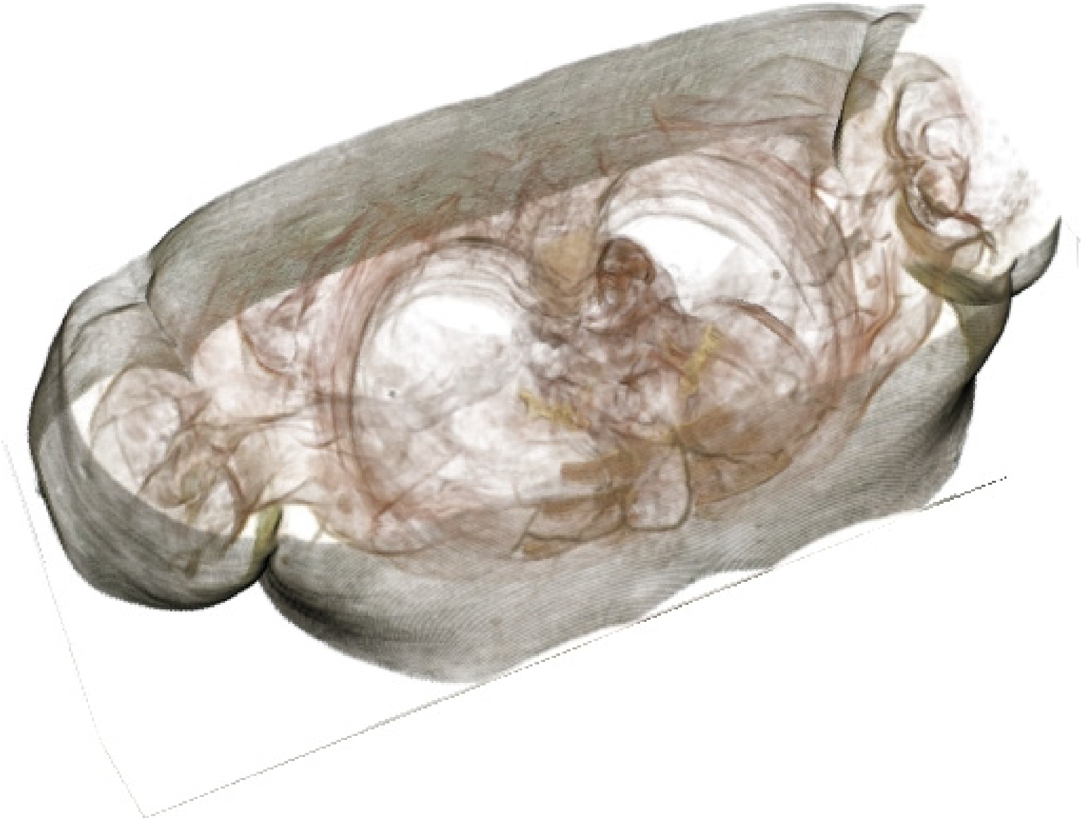

Figure 3.4: Color Distance Gradient Dot Products (MIN) (*scale* = 1.0, *exponent* = 1.0)

Typical external forces are:

$$\begin{aligned}E_{ext}(x,y) &= -|\nabla I(x,y)|^2, && (3.31)\\ E_{ext}(x,y) &= -|\nabla[G_\sigma(x,y) * I(x,y)]|^2, && (3.32)\\ E_{ext}(x,y) &= I(x,y), && (3.33)\\ E_{ext}(x,y) &= G_\sigma(x,y) * I(x,y), && (3.34)\end{aligned}$$

where $G_\sigma(x,y)$ is a two-dimensional Gaussian function with standard deviation σ and ∇ is the gradient operator [56].

In the snake model, it is important to design efficient external forces that move the active contour toward the edges by difference of intensity or intensity gradient. Gradient fields have some important properties. First, gradient vectors are normal to the edge. Second, the magnitudes get the large value in the immediate vicinity of the edges. Third, in homogeneous regions, the gradient magnitude is nearly zero. Because of the last two properties, the capture range of the gradient is small. Also, the active contour can not converge to the concave boundary, i.e., boundary of an U-shaped object.

Gradient vector flow field is an external force field that has a much larger capture range than any other field-based gradient [55]. It diffuses the gradient vectors of an edge map $f(x,y)$ computed from a two-dimensional image. Gradient vector flow is defined as the vector field $\mathbf{v}(x,y) = [u(x,y), v(x,y)]$ that minimizes the energy functional

$$E = \int\int \mu\nabla^2\mathbf{v} + |\nabla f|^2|\mathbf{v} - \nabla f|^2 \; dxdy. \qquad (3.35)$$

On color images, Wang et al. conclude that using color components as an edge map for the gradient vector flow field is superior than using edge detector outputs as an edge map. The reason is that when the edge map is processed by a gradient edge detector, the anisotropy is accumulated so that non-homogeneity is present in the final opacity results.

We have implemented the gradient vector flow based opacity transfer function. The gradient vector flow based opacity transfer function smoothes human part boundaries along normal direction. The result renders human parts boundaries with more regular and uniform opacity. Figure 3.5 renders the human thorax using gradient vector flow based transfer function.

3.3.3 Extending Color Distance Gradient Transfer Functions

Signed Color Distance Gradient

In the original color distance gradient transfer functions, the color difference metric is an unsigned distance. The unsigned distance gives the sizes but not direction of the difference. We have extended the original color difference metric to use signed distance. The signed distance captures both the sizes as well as the direction of the difference.

The sign of the color distance is determined by the result of the following equation:

$$grad_{sign} = (v1.r - v2.r) \times 0.3 + (v1.g - v2.g) \times 0.59 + (v1.b - v2.b) \times 0.11. \tag{3.36}$$

If the result of the equation is negative, then we assign the color distance to be negative. Otherwise, we assign the color distance to be positive. We assign signs to three components of the gradient vector.

Figure 3.6 shows the comparison between the original MAX function and the signed MAX function. In the signed MAX function, we use the same equation to calculate gradients but also assign signs to gradient vectors. The color difference gradient magnitudes in the signed MAX function are the same as the original MAX function. However, the direction information in the signed gradient vector provides more accurate measurement of the angular color distance gradient difference. In Figure 3.6, the opacity difference between the original MAX function and the signed MAX function is overlapped on the opacity scalar image generated by the original MAX function. The pixel colors in the figure are defined as follows:

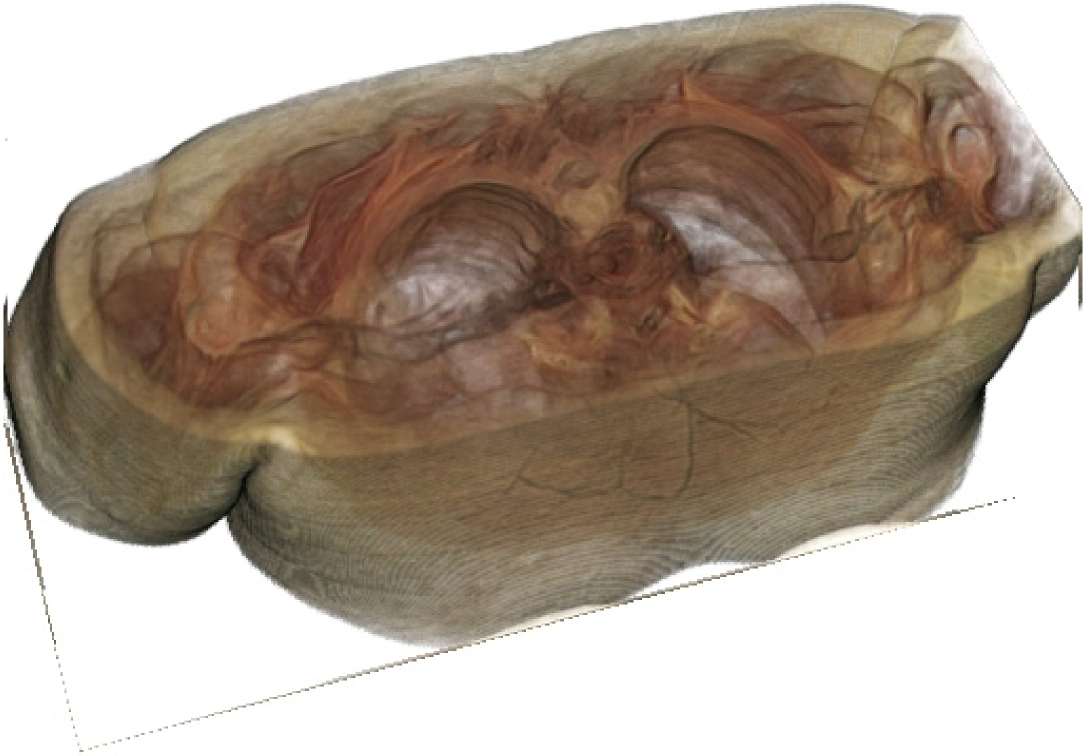

Figure 3.5: Gradient Vector Flow Opacity Mapping ($scale = 0.5$, $exponent = 1.25$)

- **Black** Black pixels in the figure stand for background and homogeny areas. Pixels in such areas have zero opacity value.
- **Gray** Gray pixels in the figure stand for opacity estimation results. Brighter gray pixels represent higher opacity scalar value. Gray pixels also means that the original MAX function and the signed MAX function have the same opacity estimation results for such pixels.
- **Green and Red** Green and red pixels in the figure stand for opacity difference between the original MAX function and the signed MAX function. Green pixels means that on such pixels the signed MAX function produces higher opacity estimation results than the original MAX function. Red pixels means that the original MAX function produces higher opacity estimation results than the signed MAX function. Brighter green and red pixels represent larger opacity differences.

From Figure 3.6, it is apparent that the signed MAX function is superior to the original MAX function in capturing color differences. Green pixels in the figure means that the signed MAX function estimates higher opacity value at the human part boundaries. The lack of red pixels in the figure means that the original MAX function estimates lower or equal opacity at the human part boundaries. In the figure, green and red colors have been scaled up for better illustration.

Combination of Color Distance Gradient Transfer Functions

The maximum angular color distance gradient difference transfer function (MAX) highlight areas with the large color distance gradients that have the largest angular variation in relation to their neighboring voxels. Therefore, the MAX function is useful for highlighting the largest color changes. The minimum angular color distance gradient difference transfer function (MIN) enhances areas where there are large gradients, but small angular color changes with one of the six adjacent voxels. Therefore, the MIN function is useful for highlighting oriented patterns within the color data, such as muscle fibers. We combine

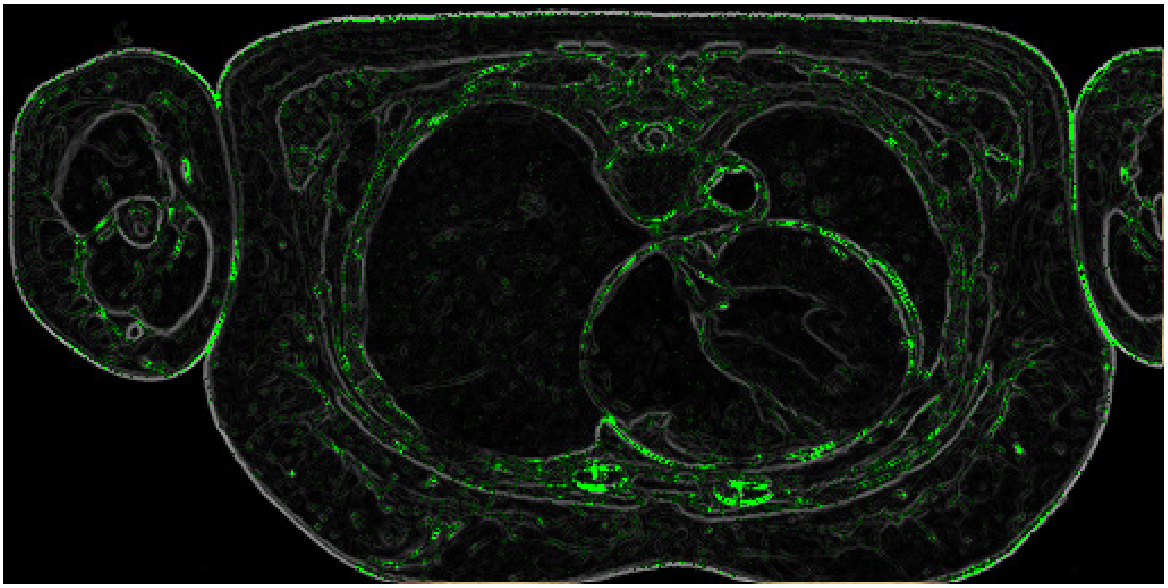

Figure 3.6: Signed Color Distance Gradient Transfer Function

the MAX and MIN transfer functions to highlight both largest color change as well as the oriented patterns.

Weighted Sum of MAX and MIN In this transfer function, we calculate the final opacity using the weighted sum of MAX and MIN transfer functions:

$$voxel\_opacity = \alpha_1 \times MAX + \alpha_2 \times MIN. \tag{3.37}$$

In experiment, we found that the following weight pair produces good results: $(\alpha_1 = 0.25, \alpha_2 = 1.0)$. Figure 3.7 shows the rendering result of the weighted sum of MAX and MIN transfer function.

Maximum of Weighted MAX and MIN Another way to combine MAX and MIN transfer functions is to use the maximum results of weighted MAX and MIN transfer functions.

$$voxel\_opacity = \max(\alpha_1 \times MAX, \alpha_2 \times MIN). \tag{3.38}$$

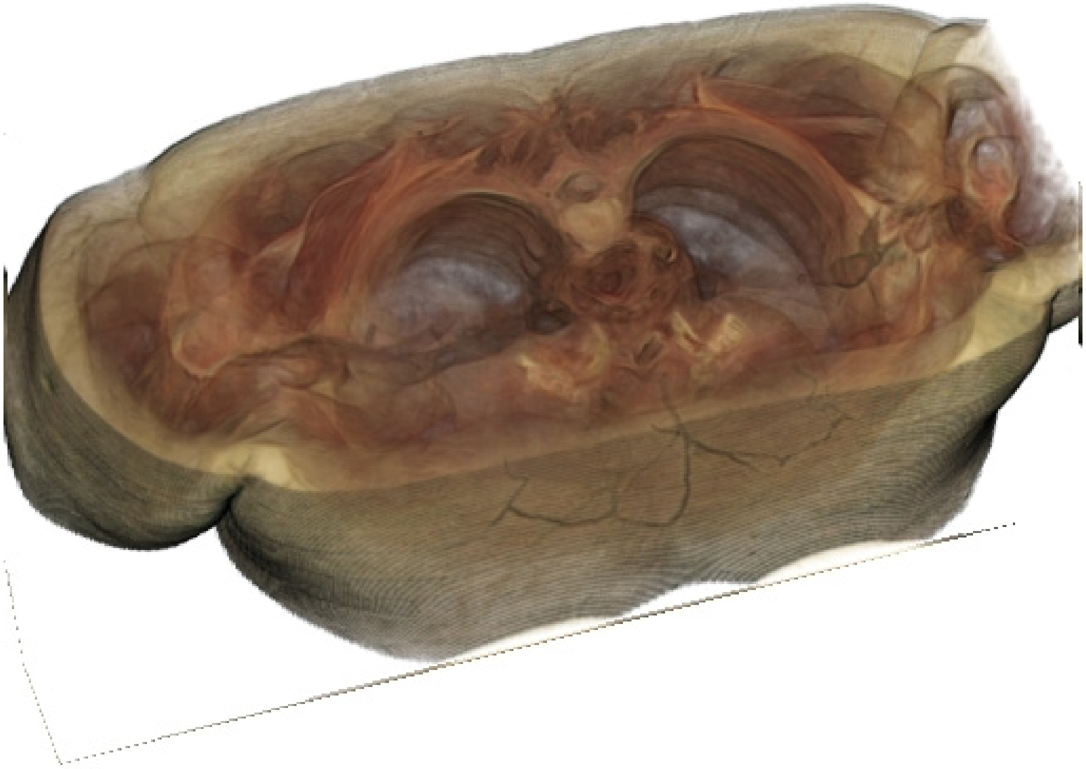

Figure 3.7: Weighted Sum of MAX and MIN ($scale = 1.0$, $exponent = 1.0$, $\alpha_1 = 0.25, \alpha_2 = 1.0$)

This transfer function produces fairly good visual results, which can be used to help weight selection in the weight color distance gradient dot products mixture transfer function. Figure 3.8 shows the rendering result of the maximum of weighted MAX and MIN transfer function.

Sobel Based MAX and MIN Functions

The original color distance gradients are calculated using central difference, which is equivalent to applying a one-dimensional mask in x and y direction on each color component:

$$G_x = \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}, and\ G_y = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}. \tag{3.39}$$

To better capture color differences, we apply Sobel edge detector on each color component for gradient generation. The Sobel operator performs a two-dimensional spatial gradient measurement on an image. Typically it is used to find the approximate absolute gradient magnitude at each point in an input gray level image. The Sobel edge operator uses a pair of 3x3 convolution masks, one estimating the gradient in the x-direction (columns) and the other estimating the gradient in the y-direction (rows). A convolution mask is usually much smaller than the actual image. As a result, the mask is slid over the image, manipulating a square of pixels at a time. The actual Sobel masks are shown below:

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, and\ G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}. \tag{3.40}$$

For three-dimensional spatial gradient measurement on a volume, we use the following

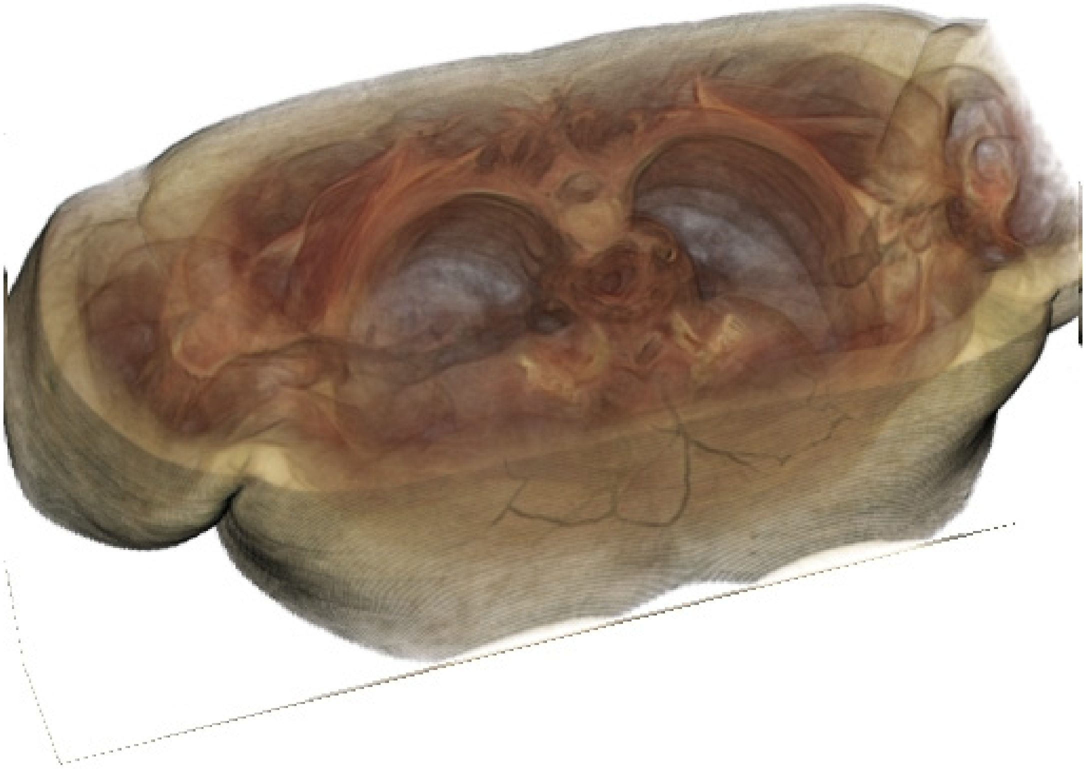

Figure 3.8: Maximum of Weighted MAX and MIN ($scale = 1.0$, $exponent = 1.0$, $\alpha_1 = 0.25, \alpha_2 = 1.0$)

three-dimensional directional Sobel filter in x, y, and z direction:

$$\left(\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -2 & 0 & 2 \\ -4 & 0 & 4 \\ -2 & 0 & 2 \end{pmatrix} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}\right). \tag{3.41}$$

Figure 3.9 shows the result of the maximum of weighted MAX and MIN transfer function using the Sobel operator.

3.3.4 Canny Edge Detection Based Opacity Transfer Function

The Canny edge detection algorithm is known to many as the optimal edge detector[58]. Canny intended to enhance the many edge detectors already out at the time he started his work. He succeeded in achieving his goal. He followed a list of criteria to improve current methods of edge detection. The first and most obvious is low error rate. It is important that edges occurring in images should not be missed and that there be no responses to non-edges. The second criterion is that the edge points be well localized. In other words, the distance between the edge pixels as found by the detector and the actual edge is to be minimumal. A third criterion is to have only one response to a single edge. This was implemented because the first two were not substantial enough to completely eliminate the possibility of multiple responses to an edge.

Based on these criteria, the Canny edge detector first smoothes the image with a Gaussian filter to eliminate noise. This results in a slightly blurred version of the original, which is not affected by a single noisy pixel to any significant degree. An example of a 5 × 5

Figure 3.9: Maximum of Weighted MAX and MIN (Sobel) ($scale = 0.75$, $exponent = 1.25$, $\alpha_1 = 0.125, \alpha_2 = 0.0025$)

Gaussian filter with $\sigma = 0.4$ is listed below:

$$\begin{pmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{pmatrix}.$$

It then finds the image gradient to highlight regions with high spatial derivatives. Using common gradient operators (Roberts, Sobel, Prewitt, etc.), the edge gradient G and direction Θ are determined:

$$G = \sqrt{G_x^2 + G_y^2}, \tag{3.42}$$

$$\Theta = \arctan \frac{G_y}{G_x}. \tag{3.43}$$

Whenever the gradient in the x direction is equal to zero, the edge direction has to be equal to 90 degrees or 0 degrees, depending on what the value of the gradient in the y direction. If G_y has a value of zero, the edge direction will equal 0 degrees. Otherwise the edge direction will equal 90 degrees. The edge direction angle is rounded to one of four angles representing vertical, horizontal, and the two diagonals (for example, 0, 45, 90 and 135 degrees).

Given estimates of the image gradients, a search is then carried out to determine if the gradient magnitude assumes a local maximum in the gradient direction. For example, if the rounded angle is zero degrees, the point will be considered to be on the edge, if its intensity is greater than the intensities in the north and south directions. If the rounded angle is 90 degrees, the point will be considered to be on the edge, if its intensity is greater than the intensities in the east and west directions. If the rounded angle is 135 degrees, the point will be considered to be on the edge, if its intensity is greater than the intensities in the north east and south west directions. If the rounded angle is 45 degrees, the point will be considered to be on the edge, if its intensity is greater than the intensities in the south east and north west directions. This is worked out by passing a 3x3 grid over the intensity map.

Large intensity gradients are more likely to correspond to edges than if they are small. In most cases it is impossible to specify a threshold at which a given intensity gradient switches from corresponding to an edge into not doing so. Therefore, Canny uses thresholding with hysteresis. Thresholding with hysteresis requires two thresholds, high and low. Making the assumption that important edges should be along continuous curves in the image allows us to follow a faint section of a given line and to discard a few noisy pixels that do not constitute a line but have produced large gradients. Therefore, we begin by applying a high threshold to mark out the edges we can be fairly sure are genuine. Starting from these, using the directional information derived earlier, edges can be traced through the image. While tracing an edge, we apply the lower threshold, allowing us to trace faint sections of edges as long as we find a starting point.

We perform three-dimensional Canny edge detection on cryosection volume. The Canny edge detection based transfer function efficiently highlights human part boundaries, helping to distinguish different human parts with similar colors. Figure 3.10 renders the human thorax using Canny edge detection based opacity transfer function.

3.3.5 Edge-Preserving Filtering in Opacity Estimation

While previous opacity estimation methods are able to highlight human part boundaries by capturing color differences on cryosection images, they also capture too much weak boundary information that blurs view and increases difficulty in distinguishing different human parts. We apply edge-preserve filtering on cryosection images to smooth human parts while preserving the boundary of human parts. The result is to reduce weak boundary information inside human parts while capturing strong boundary information distinguishing different human parts.

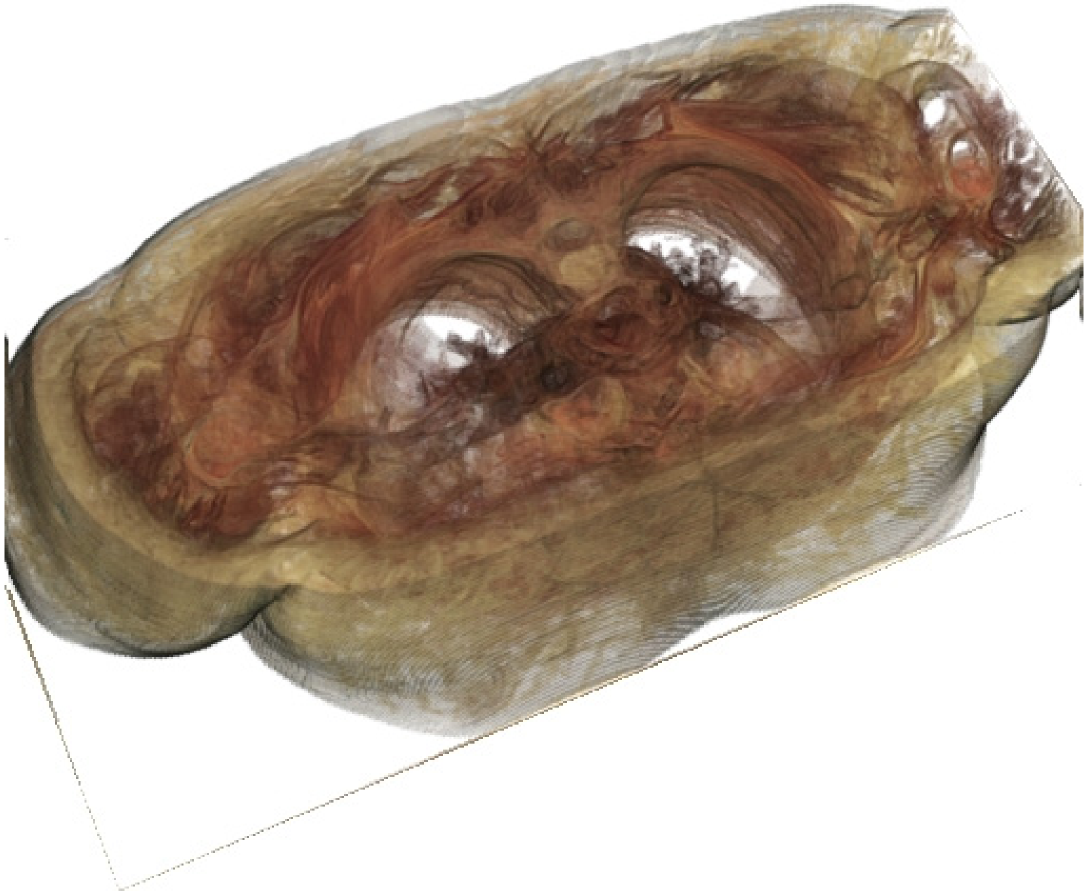

Figure 3.10: Opacity Estimation with Canny Operator (*scale* = 0.25, *exponent* = 1.0, *var* = 1.5, *low* = 5, *high* = 12)

Anisotropic Diffusion

Image smoothing tends to blur the sharp boundaries in the image that help to distinguish between anatomical structures. Perona and Malik [59] introduced an alternative, anisotropic diffusion, to the linear filtering (smoothing). Anisotropic diffusion is closely related to the earlier work of Grossberg [60], who used similar nonlinear diffusion processes to model human vision. The motivation for anisotropic diffusion (also called non-uniform or variable conductance diffusion) is that a Gaussian smoothed image is a single time slice of the solution to the heat equation; it has the original image as its initial conditions. The solution to the equation

$$\frac{\partial g(x,y,t)}{\partial t} = \nabla \cdot \nabla g(x,y,t), \tag{3.44}$$

where $g(x,y,0) = f(x,y)$ is the input image, is $g(x,y,t) = G(\sqrt{2t}) \bigotimes f(x,y)$, where $G(\partial)$ is a Gaussian with standard deviation σ.

Anisotropic diffusion includes a variable conductance term that depends on the differential structure of the image [61]. Thus, the variable conductance can be formulated to limit the smoothing at edges in images, as measured by high gradient magnitude, for example,

$$g_t = \nabla \cdot c(|\nabla g|)\nabla g, \tag{3.45}$$

where, for notational convenience, we leave off the independent parameters of g and use the subscripts with respect to those parameters to indicate partial derivatives. The function $c(|\nabla g|)$ is a fuzzy cutoff that reduces the conductance at areas of large $|\nabla g|$, and can be any one of a number of functions. Commonly used function

$$c(|\nabla g|) = e^{\frac{|\nabla g|^2}{2k^2}}, \tag{3.46}$$

is quite effective. Since the conductance term introduces a free parameter k, the *conductance parameter*, controling the sensitivity of the process to edge contrast, anisotropic diffusion entails two free parameters: 1) the conductance parameter, k, and 2) the time parameter, t, that is analogous to σ, the effective width of the filter when using Gaussian kernels.

Applying anisotropic diffusion on vector-valued images is described by Whitaker in a series of papers. In his papers, Whitaker described a detailed analytical and empirical analysis and invented a numerical scheme that virtually eliminated directional artifacts in the original algorithm.

Vector-valued anisotropic diffusion is useful for denoising data from devices that produce multiple values such as color photography. When performing nonlinear diffusion on an color image, the color channels are diffused separately, but linked through the conductance term. The output of anisotropic diffusion is an image or set of images that demonstrates reduced noise and texture but preserves, and can also enhance, edges.

Curvature Anisotropic Diffusion

Curvature anisotropic diffusion performs anisotropic diffusion on an image using a modified curvature diffusion equation (MCDE).

MCDE does not exhibit the edge enhancing properties of classic anisotropic diffusion, which can under certain conditions undergo a “negative” diffusion that enhances the contrast of edges. Equations of the form of MCDE always undergo positive diffusion, with the conductance term only varying the strength of that diffusion. Qualitatively, MCDE compares well with other non-linear diffusion techniques. It is less sensitive to contrast than classic Perona-Malik style diffusion, and preserves finer detailed structures in images [61].

The MCDE equation is given as [61]

$$f_t = |\nabla f| \nabla \cdot (|\nabla f|) \frac{\nabla f}{|\nabla f|}, \tag{3.47}$$

where the conductance modified curvature term is

$$\nabla \cdot \frac{\nabla f}{|\nabla f|}. \tag{3.48}$$

We perform curvature anisotropic diffusion on cryosection volumes. In Figure 3.11, a part of the human abdomen is rendered using the Canny edge detection based opacity transfer function. After curvature anisotropic diffusion filtering, we apply the Canny opacity transfer function on the filtered volumes for opacity estimation. The colors of voxels are defined using the unfiltered cryosection volumes.

Figure 3.12 shows a rendering result of the thorax section on curvature anisotropic diffusion filtered cryosection volume. Compared to the unfiltered rendering result in Figure 3.10, curvature anisotropic diffusion filtered rendering results shows more clear boundaries between anatomical structures while making the inside part of anatomical structures transparent.

3.4 GPU Ray Casting Volume Rendering Implementation

In VHASS, we have implemented direct volume rendering based on GPU ray-casting techniques as proposed by Krüger and Westermann [62]. The method stores the volume data as a three-dimensional texture uses a volume bounding box for viewing ray setup. For each vertex of the box, three-dimensional texture coordinates are issued as per-vertex color. The front faces and the back faces of the box are rendered to separate two-dimensional textures in two rendering passes. Entry points of viewing rays are determined by the front faces colors. Exit points of viewing rays are determined by colors on the back faces. In other words, the method defines viewing rays in three-dimensional texture space and stores intermediate ray information as RGB colors into two-dimensional textures.

Krüger and Westermann use a fixed number of rendering passes. The number of passes depends on the maximum length of a ray, the traversal step size, and the number of intermediate steps m. In each pass, m steps along the rays are performed and rendering is directed to a two-dimensional texture that can be accessed in the consecutive passes. The reason is that graphics hardware did not support conditional loops and dynamic branching. Now with the graphics hardware supporting these features, we can perform the ray traversal in one single pass in a pixel shader. In the pixel shader, for each ray starting from

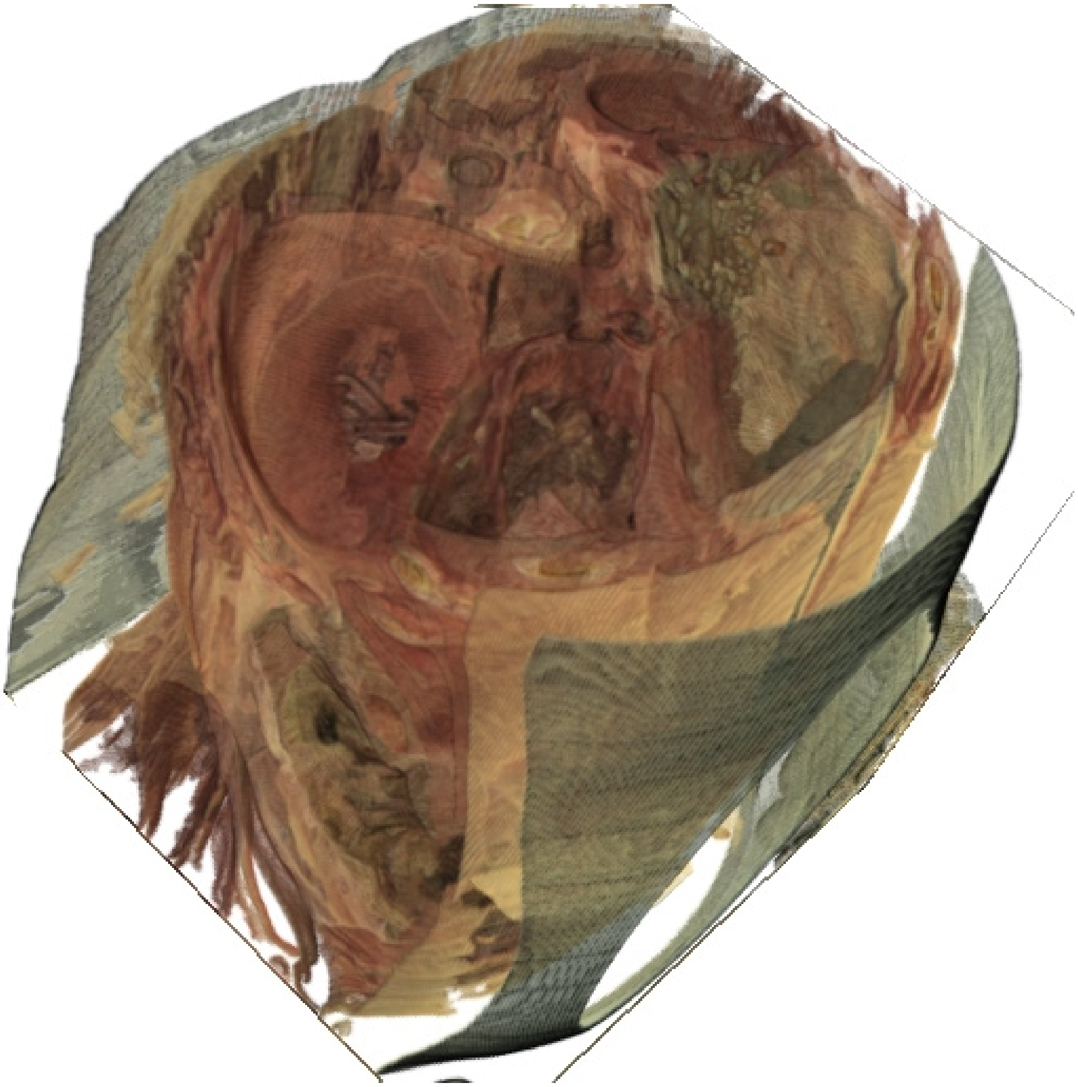

Figure 3.11: Curvature Anisotropic Diffusion (Abdomen) ($scale = 0.25$, $exponent = 1.0$)

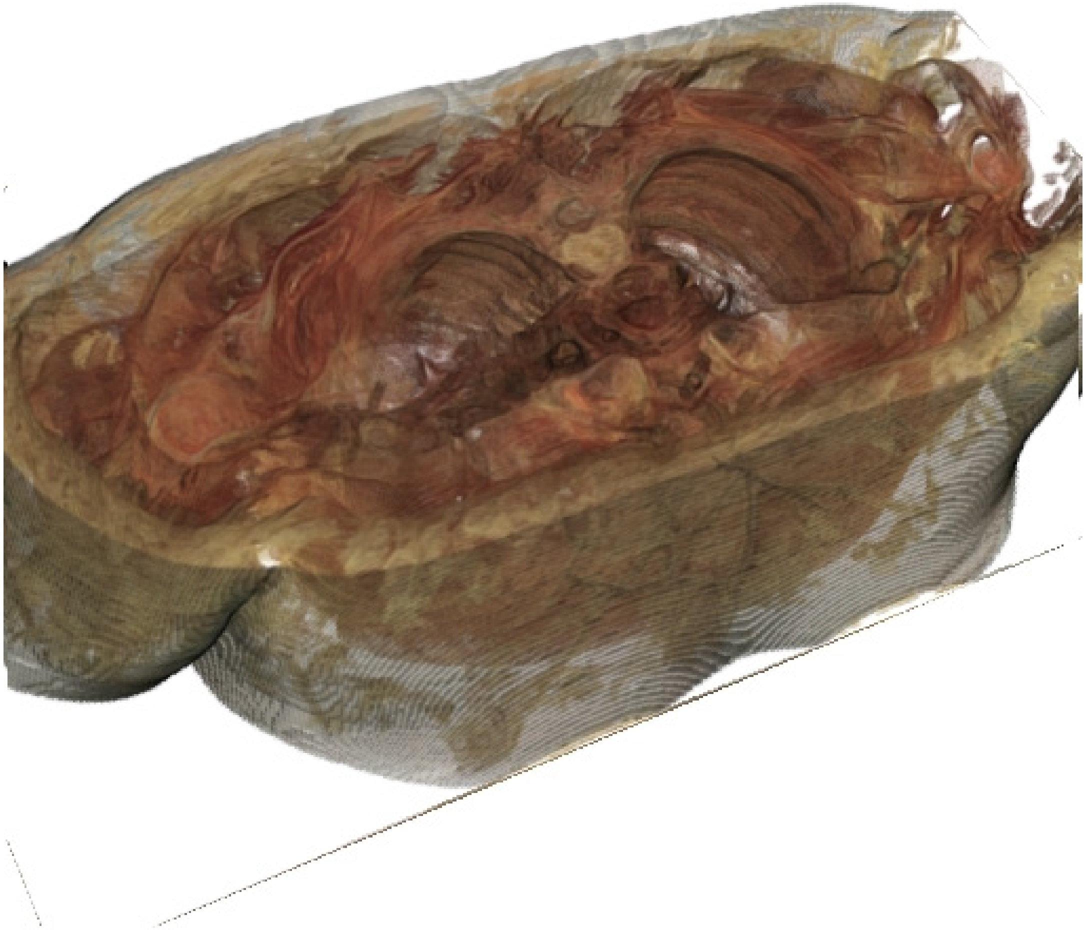

Figure 3.12: Opacity Estimation with Canny Operator (Thorax) ($scale = 0.25$, $exponent = 1.0$, $var = 1.5$, $low = 5$, $high = 12$)

the entry point, we continue to move forward along the ray based on a given traversal step size. In each step, we use the current location to access the three-dimensional texture for the cumulated color and opacity. The cumulated color and opacity are calculated using equation 3.15 and 3.16. We implement early ray termination by checking the cumulated opacity and the ray traversal distance. If the cumulated opaque is beyond a given threshold or the ray traversal distance has longer than the ray length, the ray traversal is terminated and the pixel color is determined by the cumulated color and opacity.

3.4.1 Hit-Point Refining

Hit point refining was developed by Scharsach [63]. We have implemented the algorithm in our system for better visual effect. The algorithm is a simple way to get better estimation of the real intersection point with the iso-surface after ray casting with constant sampling distance.

The algorithm starts from the first estimation of the intersection, for example, the first sample point along the ray where the density is greater than a given threshold. The algorithm goes half the previous sampling distance and checks the density value at the new position. The next step will again be half the previous step size, making this one-fourth of the original sampling distance. Depending on the density on the new position, if it is still above the threshold the next step will also be backwards, otherwise forwards. It is proven that repeating five or six times will be sufficient for most applications, no matter how low the original sampling distance is [63].

3.4.2 Fragment/Pixel Shader

In this section, we list our pixel shader code for direct volume rendering.

```
uniform sampler2D TextureFront;

uniform sampler2D TextureBack;

uniform sampler3D TextureVol;

uniform vec4 VsizeStepOffset;
```

```
uniform vec4 OpaqueRange;

void main()
{
    vec2 rasterPos = gl_FragCoord.xy/VsizeStepOffset.xy;
    vec4 rayOrigin = texture2D( TextureFront, rasterPos);
    vec4 rayEnd = texture2D( TextureBack, rasterPos);
    vec3 rayDir = rayEnd.xyz - rayOrigin.xyz;
    float rayLen = length(rayDir);
    rayDir = normalize(rayDir);

    float t = VsizeStepOffset.z + VsizeStepOffset.w;
    {//hit-point refine
        vec3 nextRayOrigin = rayOrigin.xyz + rayDir * t;
        vec4 tempColor = vec4(0.0, 0.0, 0.0, 0.0);
        while(tempColor.w == 0.0 && t < rayLen)
        {
            tempColor = texture3D( TextureVol, nextRayOrigin);
            t += VsizeStepOffset.z;
            nextRayOrigin = rayOrigin.xyz + rayDir*t;
        }

        t -= VsizeStepOffset.z;
        float step = VsizeStepOffset.z * 0.5;
        for (int i = 0; i < 8; i++)
        {
            if (tempColor.w == 0.0)
                t += step;
```

```
        else
            t -= step;
        nextRayOrigin = rayOrigin.xyz + rayDir * t;
        tempColor = texture3D( TextureVol, nextRayOrigin);
        step *= 0.5;
    }
    t = 0.0;
    rayOrigin.xyz = nextRayOrigin;
    rayDir = rayEnd.xyz - rayOrigin.xyz;
    rayLen = length(rayDir);
    rayDir = normalize(rayDir);
}

//do ray casting
vec4 pixelColor = vec4(0);
float alpha = 0.0;
vec3 nextRayOrigin = rayOrigin.xyz;
while( (t < rayLen) && (alpha < 1.0) )
{
    vec4 nextColor = texture3D( TextureVol, nextRayOrigin);
    if (nextColor.w > OpaqueRange.x)
    {
        float tempAlpha = nextColor.w;
        if (tempAlpha > OpaqueRange.y)
            tempAlpha += OpaqueRange.w;
        else
            tempAlpha += OpaqueRange.z;
        tempAlpha = tempAlpha < 0.0 ? 0.0 : tempAlpha;
```

```
            tempAlpha = tempAlpha > 1.0 ? 1.0 : tempAlpha;


            tempAlpha = (1.0-alpha)*tempAlpha;
            pixelColor += nextColor*tempAlpha;
            alpha += tempAlpha;
        }
        t += VsizeStepOffset.z;
        nextRayOrigin = rayOrigin.xyz + rayDir*t;
    }
    gl_FragColor = vec4(pixelColor.xyz, 1.0);
}
```

Chapter 4: Photo-Realistic Virtual Human Parts via Surface Reconstruction

This chapter is organized as follows. In the first section, we introduce the idea to combine surface and volume together for new 3D models. The second section focuses on existing surface reconstruction methods. Our new neighbor-based surface reconstruction method is outlined in the third section. In the fourth section, we discuss the modification to the Marching Cubes method to work on our data. Finally, our implementation of high quality surface models rendering is discussed in the fifth section.

4.1 Combining Surface Models and Volume Data

In VHASS, segmented human organs are rendered as triangle mesh surface models. Compared to direct volume rendering described in Chapter three, surface models are easier to render using the standard graphics pipeline. Surface models are also more flexible than direct volume rendering. For example, direct volume rendering usually requires viewport size to be power of 2, such as 256 by 256, 512 by 512, and 512 by 1,024. The performance of direct volume rendering is also strongly affected by viewport size: the larger the viewport the more pixels involved in volume the rendering integral. The effect of viewport size on rendering performance for surface models is more moderate compared to direct volume rendering. While reducing the number of triangles in surface models is a common method to improve rendering performance without sacrificing too much on rendering quality, reducing the number of voxels dramatically decreases rendering quality without improving on rendering performance.

However, traditional surface models are not enough for human parts exploration. First, most surface reconstruction methods do not take colors into account. The results of these methods includes only vertex coordinates and surface topology information. While this problem could be solved by using cryosection image stack as three-dimensional texture and performing three-dimensional texture mapping in rendering, the internal of human parts remains undefined, which decreases realistic portrayal of virtual human parts.

To solve the problem, we bind volume data and surface models together. Rather than use pre-created surface models, we extract surface from the segmentation index volume and use a three-dimensional color texture created from a cryosection image stack for accurate cadaver appearance. When exploring human parts, we update the segmentation index volume based user interaction and re-create surface from the updated binary mask volume. For this purpose, the segmentation index volume includes from all pixels belonging to human parts rather than human part boundary pixels only. This is different from boundary points based surface reconstruction algorithm such as Delauny triangulation based methods.

Since we are doing run-time human parts surface reconstruction during interaction, we need a fast surface reconstruction algorithm to ensure run-time performance. Among the existing surface reconstruction algorithms, the well-known Marching Cubes method has very fast speed to be a good candidate. However, the original Marching Cubes has to be modified to work on our segmented human parts. We also developed a new neighbor-based surface reconstruction algorithm to work on our data.

4.2 Surface Reconstruction Methods

4.2.1 Marching Cubes Method

In 1987, William E. Lorensen et al. developed the Marching Cubes algorithm for surface reconstruction [64]. The marching cubes algorithm works on well organized sample points to extract an iso-surface. An iso-surface, a three-dimensional analog of an iso-contour, is a surface that represents points of a constant value (e.g. pressure, temperature, velocity, density) within a volume of space.

Given a set of parallel slices, the algorithm constructs logic cubes based on pixels from

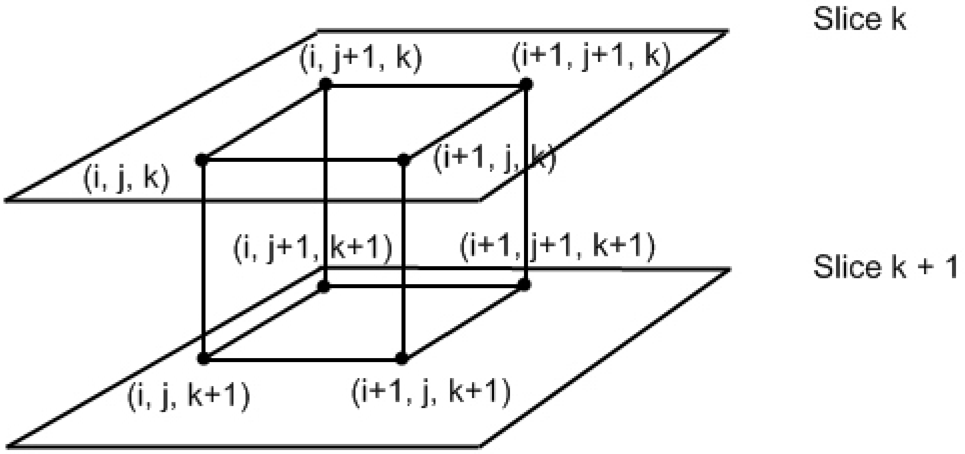

Figure 4.1: Marching Cube

adjacent slices (Figure 4.1). In Figure 4.1, black dots are pixels on slices. For each cube, the algorithm determines how the surface intersects this cube, then moves to the next cube. To find the surface intersection in a cube, a one is assigned to one cube vertex if the data value at the vertex exceeds or equals the value of the surface. A zero is assigned to one cube vertex if the data value at the vertex less than the value of the surface. Then the topology of the surface within one cube is determined by this assumption.

For each cube, depending on if the eight vertices is inside or outside an iso-surface, a surface intersects the cube in $2^8 = 256$ ways. A lookup table for surface-edge intersection is created by enumerating these 256 ways. Using the complementary symmetry and the rotational symmetry, the Marching Cubes method reduces the 256 ways to 14 patterns. For each pattern, a triangulation schema is created to produce one or more triangles. The only exception is the first pattern in which no surface intersects the cube edges.

The Marching Cubes method calculates normal for each triangle vertex using pixel density gradient vector. To estimate the gradient vector at the surface of interest, the Marching Cubes method first estimate the gradient vectors at the cube vertices and linearly interpolate the gradient at the point of intersection [64]. The gradient at cube vertex (i, j, k), is obtained using central differences along the three coordinates axes:

$$G_x(i, j, k) = \frac{D(i+1, j, k) - D(i-1, j, k)}{\Delta x}, \tag{4.1}$$

$$G_y(i, j, k) = \frac{D(i, j+1, k) - D(i, j-1, k)}{\Delta y}, \tag{4.2}$$

$$G_z(i, j, k) = \frac{D(i, j, k+1) - D(i, j, k-1)}{\Delta z}, \tag{4.3}$$

where $D(i, j, k)$ is the density at pixel (i, j) in slice k and $\Delta x, \Delta y, \Delta z$ are the lengths of the cube edges. Dividing the gradient by its length produces the unit normal at the vertex required for rendering [64]. Then this normal is interpolated to the point of intersection.

Decimation on Marching Cubes Surfaces

The Marching Cubes algorithm generates an excessively large number of triangles to represent an iso-surface. Generating many triangles increases the rendering time, which is directly proportional to the number of triangles. In 1996, Raj Shekhar et al. presented a decimation method to reduce the number of triangles generated by the Marching Cubes algorithm [65]. The algorithm is based on the observation that if a voxel with a piece of surface in it is known, the 6-adjacent neighbor voxels where the surface may extend can be determined based on the continuity of surface.

The algorithm uses octree-based adaptive down-sampling as a way to reduce the number of triangles. The octree is traversed level by level from bottom to up. At any level of the octree, five different merging criteria are applied to attempt to replace eight child voxels with one parent voxel. Since the merging process modifies the octree, a crack-patching strategy is used to align surfaces generated on high and low resolution voxels. As a result, the algorithm approximates the surface with large triangles at low frequency regions and small triangles at high frequency regions.

Improvements on Surface Quality

Bilinear and Trilinear Interpolations The original Marching Cubes algorithm is based upon linear interpolation along edges of the voxels. The surface generated by the Marching Cubes method may include some erroneous results consisting of iso-surfaces with holes. The problem is due to an incomplete case analysis. In 1991, Nielson and Hamann introduced the concept of deciding on ambiguous faces by using bilinear interpolation on faces [66]. In 2003, Nielson presented another improved Marching Cubes algorithm [67]. The new algorithm improves the original algorithm by using trilinear interpolation on the interior of voxels.

Dual Marching Cubes In 2004, Nielson presented the definition and computational algorithm for a new class of surfaces, which in geometry are dual to the iso-surface produced by the original Marching Cubes algorithm [68]. These new iso-surfaces have the same separating properties as the Marching Cubes surface, but they are comprised of quad patches that tend to eliminate the common negative aspect of poorly shaped triangles of the Marching Cube iso-surfaces. First the concept of the Marching Cubes surface is extended to a patch version. The patch version eliminates the edge of the Marching Cubes surface interior to cubes to obtain a surface S with polygon-bounded patches, where each vertex is included in exactly four patches. Then for each patch of S a vertex of the dual surface is associated. The dual surface consists of quad patches that are connected in exactly the same manner as the connectivity of the vertices of the surface S. As a result, the dual surface is smoother than the original Marching Cube surface.

Improvements on Robustness and Accuracy In 2003, Lopes et al. improved the robustness and accuracy of the Marching Cubes algorithm by creating a representation of the surface in the interior of each grid cell [69]. The representation correctly models the topology of the trilinear interpolation within the cell and is robust under perturbations of the data and threshold value. To achieve this, a small number of key points in each cube interior are identified. These key points, critical to the final surface definition, lie on the iso-surface. The representation is robust in the sense that the surface is visually continuous as the data and threshold change in value. Compared to the Marching Cubes algorithm, the new algorithm generates much more triangles.

4.2.2 Delaunay Triangulation Based Surface Reconstruction

Another set of surface reconstruction algorithms is based on Delaunay triangulation. A triangulation is a subdivision of an area (volume) into triangles (tetrahedrons). The Delaunay triangulation has the property that the circumcircle (circumsphere) of every triangle (tetrahedron) does not contain any points of the triangulation. The Delaunay triangulation is the dual structure of the Voronoi diagram. In 1998, Nina Amenta et al. developed a simple combinatorial algorithm that computes a piecewise-linear approximation of a smooth surface from a finite set of sample points [70]. After the Delaunay triangulation, the algorithm uses Voronoi vertices to remove unnecessary triangles. Based on the three-dimensional Voronoi diagram and Delaunay triangulation, the algorithm is very effective when processing highly non-uniform sample points.

Although the Delaunay triangulation based methods output smooth surface mesh, the major complaint against Voronoi and Delaunay based methods is that they are slow and cannot handle large data with current computing resources. In 2001, Dey et al. combined Delaunay based method and octree subdivision to process large data [71]. The entire set of sample points is partitioned into smaller clusters using an octree subdivision. Then Delaunay based surface reconstruction is applied to each of these clusters separately. To ensure the surface patches match with each other, sample points from neighboring clusters are padded into each cluster for stitching to be computed consistently over all clusters. On a 733M Hz Pentium III processor with 512M Bytes RAM, the method could process one million sample points in 53 minutes.

4.2.3 Other Methods

In 1992, Hoppe et al. presented an algorithm that takes as input an unorganized set of points on or near an unknown manifold M and produces a surface that approximates M [72]. Their algorithm estimates a tangent plane at each sample point using the k nearest neighbors. Then the distance to the plane of the closest sample point is used as a signed distance function. The zero set of the function is then approximated by a continuous piecewise-linear surface using the marching cubes algorithm. The algorithm would fail in case of arbitrarily dense sets of sample with almost collinear nearest neighbor sets.

In 2005, Wang et al. presented a new approach for surface reconstruction from unorganized point clouds without Delaunay triangulation [73]. Surface reconstruction using the approach involves three major steps. First, the space containing the point cloud is subdivided to create a voxel representation. Based on the voxel representation, a voxel surface is computed using gap-filling and topological-thinning operations. Finally, a polygonal mesh surface is extracted from the voxel surface. This approach is rather fast compared with existing surface reconstruction methods. For a model with 1,769,513 points, the approach could reconstruct surface within one minute.

4.3 Neighbor-Based Surface Reconstruction

As mentioned in Chapter 2, automatic accurate segmentation on color cryosection images is still an open area of research. Thus, we performed surface reconstruction on the segmented human parts. It is natural to visualize the point clouds of the segmented human parts as little boxes piled together and to use unblocked faces of these little boxes for the boundary surface. The output of this case has the same appearance as the Cuberille Voxel Representation [74]. However, the surface generated in this way has a strong block-like effect. To overcome the problem of the low visual quality, two methods, normal refinement and voxel subdivision, were proposed. The normal refinement method refines the normal approximation scheme by taking into account the orientation of the neighbor cubes. These neighbor orientations index a lookup table of different normal vector orientations that are finally used as normal [75]. Since the underlying triangle mesh is not modified, the normal-refinement method still exhibits strong block effect, especially on the silhouette. On the other hand, the voxel subdivision method adaptively subdivides a voxel into subvoxels until the cuberille primitives approximate the surface of the object accurately. Since we are already using high resolution volume (512^3 or higher resolution) for surface resolution, subdivision will produce too many voxels.

Our method is to cut voxels to reduce block effect. Vertex normal is generated based on cutting result for smooth shading. We also get smoothed surface and silhouette without increasing voxel count. In the segmentation index volumes, for each non-zero value v, foreground voxel is defined to be voxels having the value v; other voxels having value $\tilde{v}$ $\in (1, \ldots, v-1, v+1, \ldots, n)$ are defined as background voxels. A foreground voxel having one or more background neighbors is considered to be visible on the boundary. However, a foreground voxel having all background voxels is considered to be noise and will not be processed. A foreground voxel having all foreground neighbors is considered to be inside human parts. Such foreground voxel does not contribute to the boundary surface. Note one foreground voxel for one specific human part can be background voxel for other segmented human parts. For foreground voxels on the boundary of the volume, one or more adjacent neighbors may locate outside the volume. We consider such neighbors to be background voxels.

4.3.1 Implementation

For each voxel, the voxel as well as the 26-adjacent neighbors are a $3 \times 3 \times 3$ matrix just like a Rubic's cube. We number 26-adjacent neighbors as well as the central voxel together from 0 to 26 (27 in total), starting form the voxel having the minimum x, y, and z indices. We encode the existence of all 27 voxels into a 27-bit length binary string, which is stored in an 32-bit integer. Each binary bit indicates the classification of one neighbor: 0 means the neighbor is a background voxel, and 1 means the neighbor is a foreground voxel. For

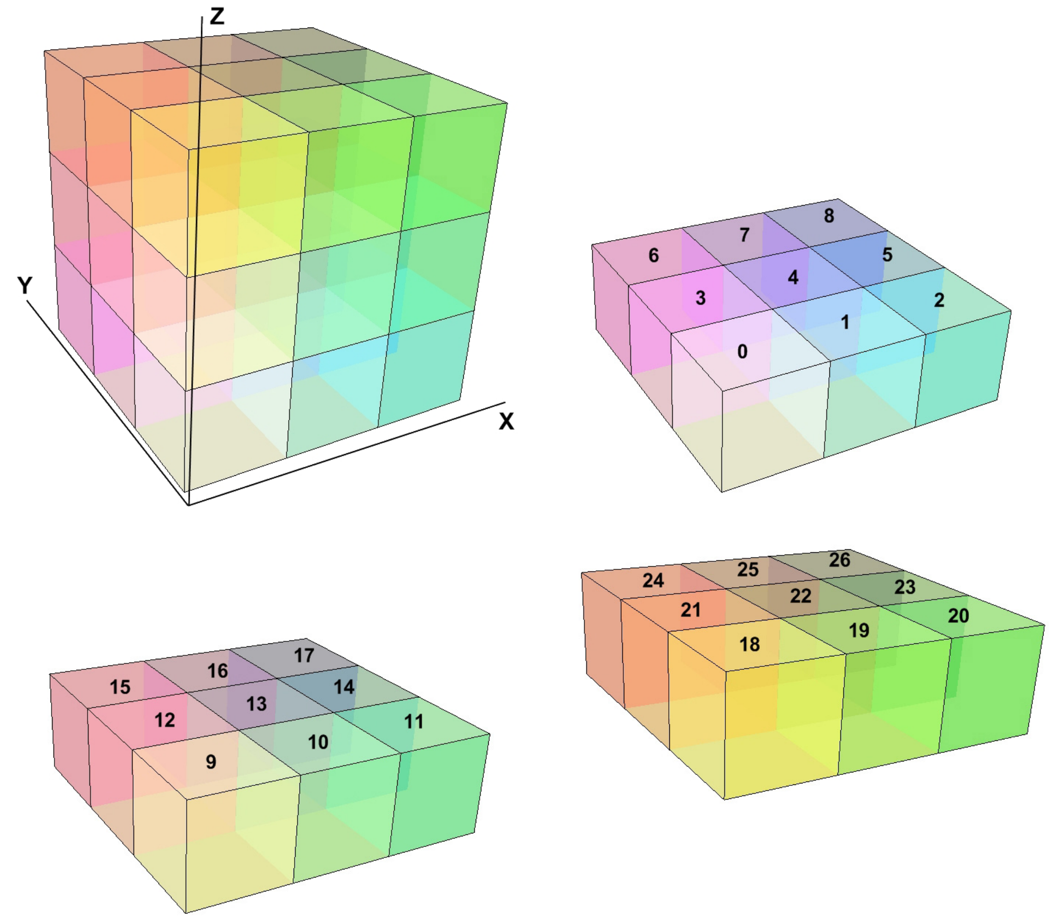

Figure 4.2: Voxel Numbering.

a voxel on the boundary of the volume, some of its neighbors may be outside the volume. Such neighbors are assigned value 0 in the binary string. Figure 4.2 shows the numbering schema.

In order to determine how to cut one voxel, we examine its 26-adjacent neighbors. Fortunately, only part of the 26-adjacent neighbors needs to be considered. We found that maximally 14 neighbors are used for cutting, although different voxels can have different subsets of their 26-adjacent neighbors needed for cutting.

We first examine 6-adjacent neighbors to test if a voxel is a visible foreground voxel.

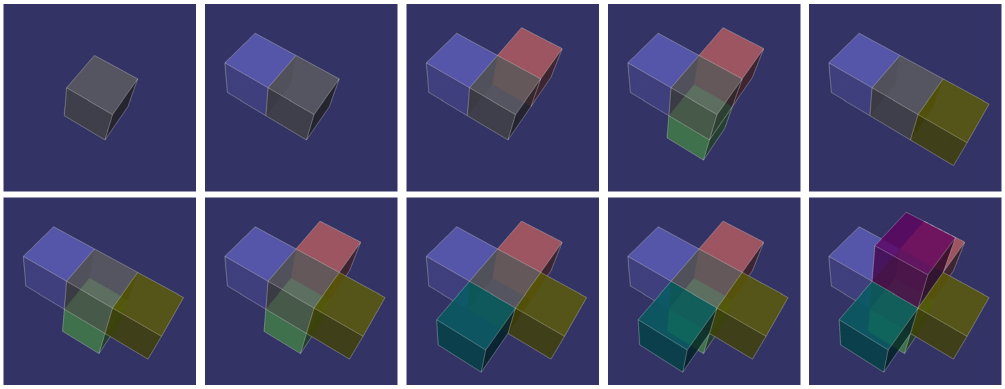

Figure 4.3: Ten Patterns.

The index of the 6-adjacent neighbors for the central voxel are 4, 10, 12, 14, 16, 22. Clearly, the presence or absence of all 6-adjacent neighbors could have $2^6 = 64$ different cases, which we are call *base cases*. Using complementary symmetry and rotation symmetry, we reduce the 64 base cases to 10 different patterns. Each pattern includes one or more base cases. Figure 4.3 shows all 10 different patterns. In the figure, color boxes are adjacent neighbors of the central gray color box. Ten patterns in the figure are indexed from a to j, left to right, top to bottom. In these patterns, pattern (a) and pattern (j) are two special cases, completely outside the human part and completely inside the human part. Voxels that completely inside or outside human part are not parts of human part boundary surface. Hence, in surface reconstruction, we do not process pattern (a) and pattern (j). Pattern (h) and pattern (i) are two other special cases in that we do not need to perform a cut. We generate two quad polygons for the two visible faces of voxels in pattern (h). In pattern (i), we generate one quad polygon for the visible face of the voxel.

For each voxel, we generate 15 vertices for possible polygons generated. Vertices are located at the bounding box and the center of the voxel. Figure 4.4 shows positions of these 15 vertices.

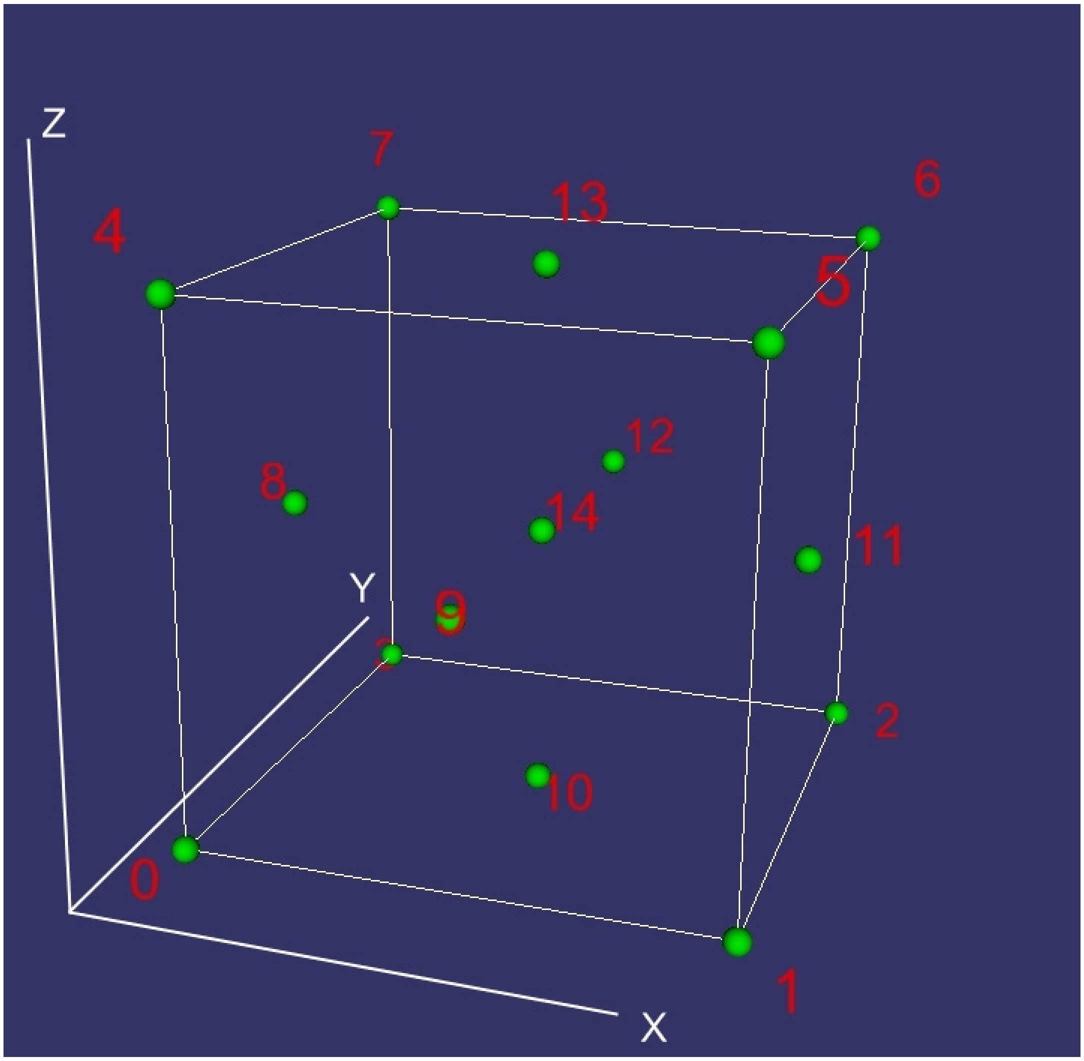

Figure 4.4: Vertices for Cutting.

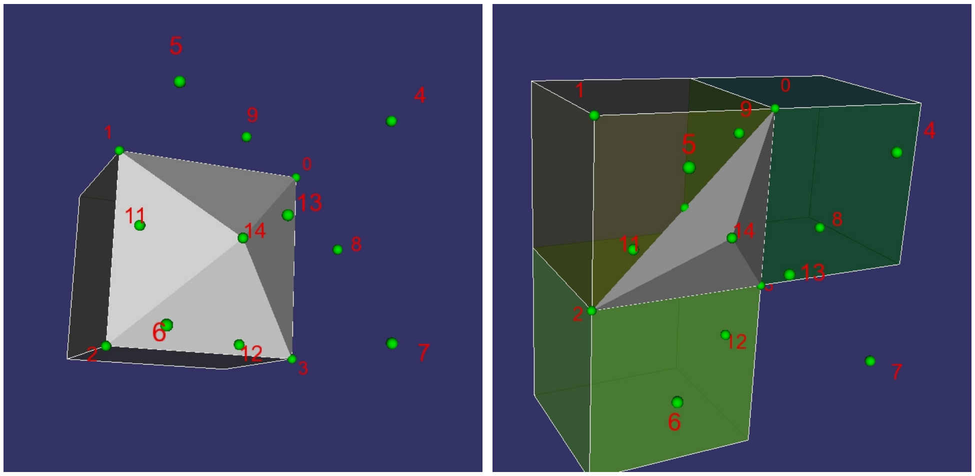

Figure 4.5: Cutting Results in Pattern (b)

Cutting on Voxels

On pattern (b), (c), (d), (e), (f), and (g), we consider 6-adjacent neighbors and a subset of remaining neighbors for cutting.

- **Pattern (b)**. Pattern (b) includes 6 different base cases (base case numbers: 1, 2, 4, 8, 16, 32). From the first one to the last one, we consider the following non-6-adjacent neighbors: (3, 7, 5, 1), (9, 1, 11, 19), (3, 9, 21, 15), (23, 11, 5, 17), (7, 15, 25, 17), (23, 25, 21, 19). These non-6-adjacent neighbors are called *extended neighbors*. Each base case has four extended neighbors used in cutting.

  It is easy to recognize that four extended neighbors have 16 different cases. These cases are called *extended cases*. Two different cutting results are created for extended cases. Figure 4.5 shows the two results for the first base case in the pattern. Extended neighbors are also shown in the figure. Other base cases have similar cutting results with rotation symmetry and complementary symmetry. In the figure, cutting results are numbered from left to right, top to bottom, starting from 0.

  The table lists cut results, default triangle indices, and corresponding extended cases.

Table 4.1: Cutting Results, Default Triangle Indices, and Corresponding Extended Cases (Pattern (b))

Cutting Results	**Default Triangle Indices**	**Extended Cases**
0	(2,3,14),(3,0,14),(0,1,14),(1,2,14)	0,1,2,4,5,7,8,10,11,13,14,15
1	(1,14,0),(14,3,0),(1,3,14)	3,6,9,12

Table 4.2: Cutting Results, Default Triangle Indices, and Corresponding Extended Cases (Pattern (c))

Cutting Results	**Default Triangle Indices**	**Extended Cases**
0	(1,2,5),(2,3,4),(2,4,5),(3,0,4)	remaining extended cases
1	(2,4,1),(2,3,4),(3,0,4)	3,35,51,83,99,115,9,41,57, 89,105,121,65,69,71,75, 77,79,17,21,23,27,29,31
2	(2,4,1),(4,2,0)	19,73
3	(0,5,2),(5,1,2)	67,25

- **Pattern (c)**. Pattern (c) includes 12 different base cases (base case numbers: 3, 5, 6, 9, 10, 17, 20, 24, 34, 36, 40, 48). From the first one to the last one, we consider the following extended neighbors: (1, 9, 19, 11, 5, 7, 3), (3, 1, 5, 7, 15, 21, 9), (9, 19, 11, 1, 3, 15, 21), (5, 7, 3, 1, 11, 23, 27), (11, 1, 9, 19, 23, 17, 5), (7, 3, 1, 5, 17, 25, 15), (15, 7, 17, 25, 21, 9, 3), (17, 5, 11, 23, 25, 15, 7), (19, 11, 1, 9, 21, 25, 23), (21, 25, 23, 19, 9, 3, 15), (23, 19, 21, 25, 17, 5, 11), (25, 15, 7, 17, 23, 19, 21). Each base case has seven extended neighbors used in cutting.

  Seven extended neighbors have 128 different extended cases. Four different cut results are created for extended cases. Figure 4.6 shows the four cut results for the first base case in the pattern. Extended neighbors are also shown in the figure. Other base cases have similar cutting results with rotation symmetry and complementary symmetry. In the figure, cutting results are numbered from left to right, top to bottom, starting from 0.

- **Pattern (d)**. Pattern (d) includes 8 different base cases (base case numbers: 7, 11,

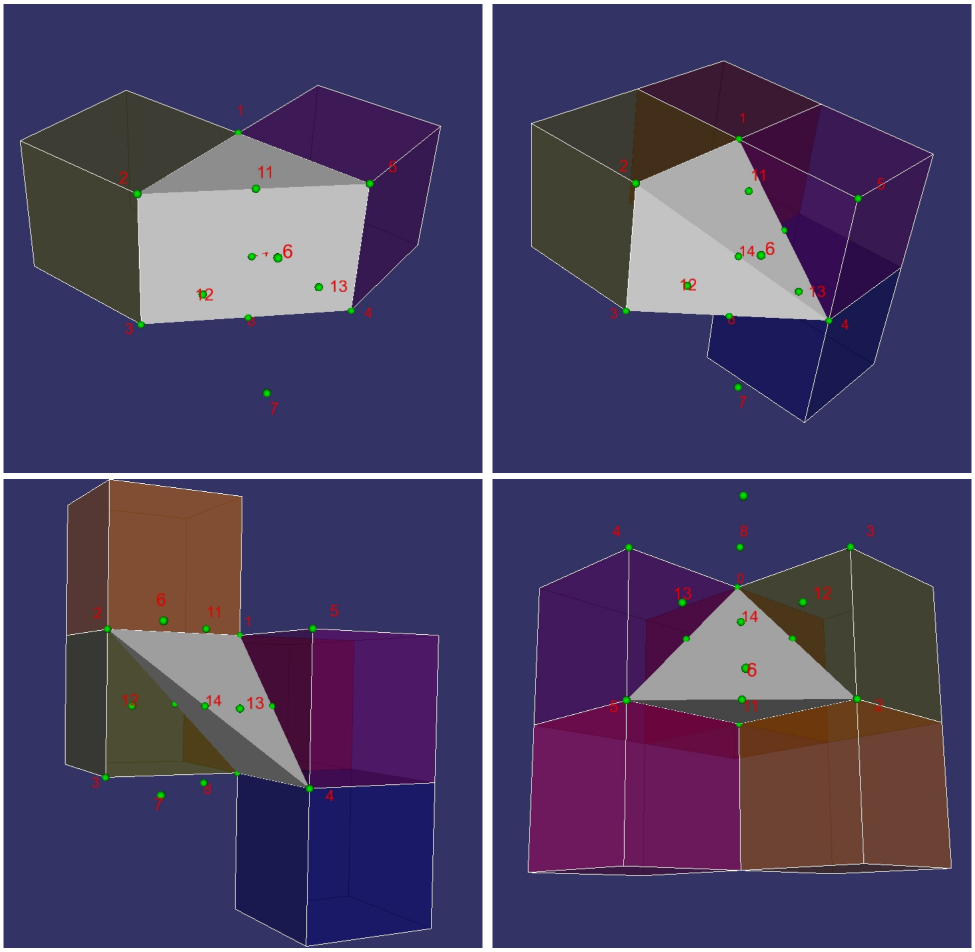

Figure 4.6: Cutting Results in Pattern (c)

Table 4.3: Cutting Results, Default Triangle Indices, and Corresponding Extended Cases (Pattern (d))

Cutting Results	**Default Triangle Indices**	**Extended Cases**
0	(1,3,4)	0
1	(1,3,14),(1,14,5),(14,3,7), (5,14,4),(4,14,7)	5,6,7,9,10,11,13,14,15, 17,18,19,33,34,35,49,50,51, 20,24,28,36,40,44,52,56,60
2	(1,3,7),(1,7,4)	1,2,3,4,8,12,16,32,48
3	(1,2,14),(14,2,3),(14,3,7), (14,7,4),(1,14,5),(5,14,4)	remaining extended cases

21, 25, 38, 42, 52, 56). From the first one to the last one, we consider the following extended neighbors: (15, 21, 19, 11, 5, 7), (7, 3, 9, 19, 23, 17), (17, 25, 21, 9, 1, 5), (25, 15, 3, 1, 11, 23), (25, 23, 11, 1, 3, 13), (17, 5, 1, 9, 21, 15), (7, 17, 23, 19, 9, 3), (5, 11, 19, 21, 15, 7). Each base case has six extended neighbors used in cutting.

Six extended neighbors have 64 different extended cases. Four different cut results are created for extended cases. Figure 4.7 shows the five cut results for the first base case in the pattern. Extended neighbors are also shown in the figure. Other base cases have similar cutting results with rotation symmetry and complementary symmetry. In the figure, cutting results are numbered from left to right, top to bottom, starting from 0.

- **Pattern (e)**. Pattern (e) includes 3 different base cases (base case numbers: 12, 18, 33). From the first one to the last one, we consider the following extended neighbors: (11, 5, 17, 23, 15, 3, 9, 21), (9, 1, 11, 19, 17, 7, 15, 25), (1, 3, 7, 5, 25, 21, 19, 23). Each base case has eight extended neighbors used in cutting.

  Eight extended neighbors have 256 different extended cases. Six different cut results are created for extended cases. Figure 4.8 shows the six cut results for the first base case in the pattern. Extended neighbors are also shown in the figure. Other base cases have similar cutting results with rotation symmetry and complementary symmetry. In the figure, cutting results are numbered from left to right, top to bottom, starting

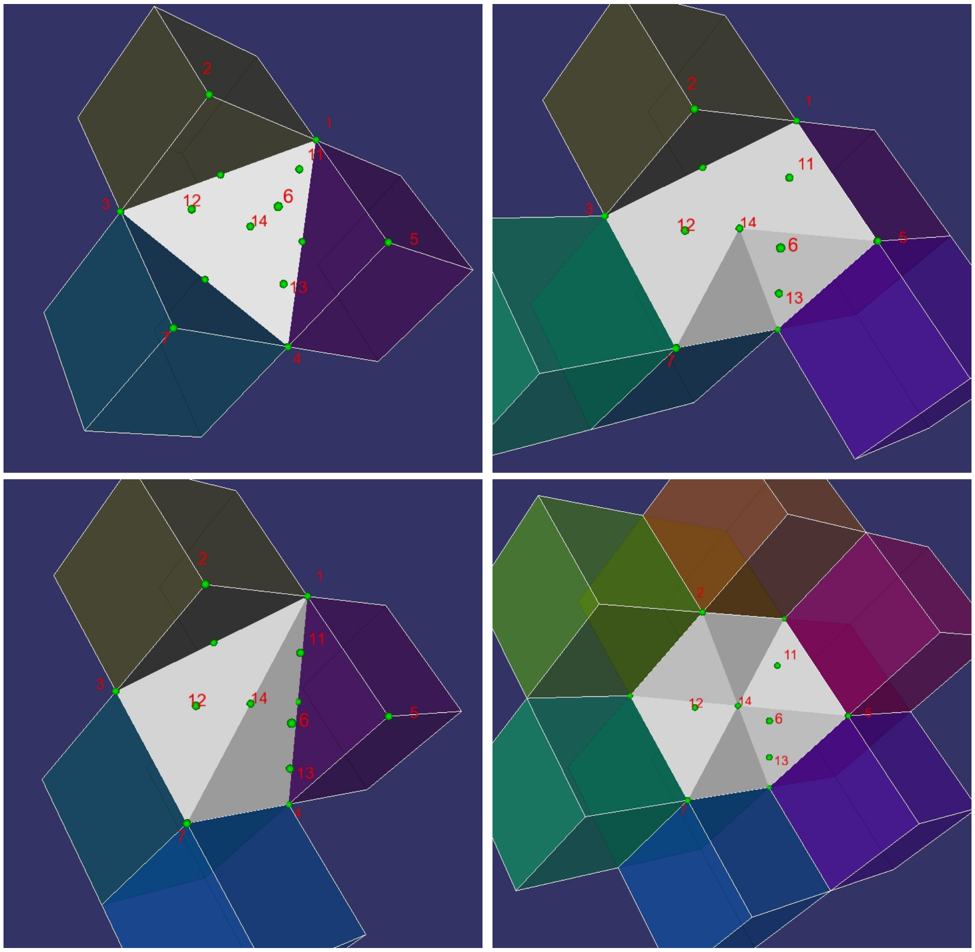

Figure 4.7: Cutting Results in Pattern (d)

Table 4.4: Cutting Results, Default Triangle Indices, and Corresponding Extended Cases (Pattern (e))

Cutting Results	**Default Triangle Indices**	**Extended Cases**
0	(0,3,2),(0,2,1),(3,7,6), (3,6,2),(7,4,5),(7,5,6), (5,4,0),(5,0,1)	remaining extended cases
1	(2,7,5),(0,5,7),(3,2,1), (3,1,0),(2,3,7),(0,1,5)	51,60,102,105,150,153,195,204
2	(0,1,6),(0,6,7),(7,6,2) (7,2,3),(3,2,1),(3,1,0)	54,99,156,201
3	(0,5,7),(1,3,6),(6,3,7), (1,0,3),(1,5,0),(5,6,7)	57,108,147,198,204
4	(5,2,7),(2,3,7),(3,2,1) (3,1,0),(5,4,0),(5,0,1) (5,7,4)	3,19,35,67,83,115,131,163,179,211,227,243,6,22,38,70,86,118,134,166,182,214,230,246,9,25,41,73,89,121,137,169,185,217,233,249,12,28,44,76,92,124,140,172,188,220,236,252,48,49,50,52,53,55,56,58,59,61,62,63,96,97,98,100,101,103,104,106,107,109,110,111,144,145,146,148,149,151,152,154,155,157,158,159,192,193,194,196,197,199,200,202,203,205,206,207,

from 0.

- **Pattern (f)**. Pattern (f) includes 12 different base cases (base case numbers: 13, 14, 19, 22, 26, 28, 35, 37, 41, 44, 49, 50). From the first one to the last one, we consider the following extended neighbors: (5, 17, 23, 11, 3, 15, 21, 9), (9, 21, 15, 3, 11, 23, 17, 5), (7, 15, 25, 17, 1, 9, 19, 11), (9, 1, 11, 19, 15, 7, 17, 25), (11, 19, 9, 1, 17, 25, 15, 7), (15, 3, 9, 21, 17, 5, 11, 23), (1, 3, 7, 5, 19, 21, 25, 23), (3, 7, 5, 1, 21, 25, 23, 19), (5, 1, 3, 7, 23, 19, 21, 25), (23, 11, 5, 17, 21, 9, 3, 15), (7, 5, 1, 3, 25, 23, 19, 21), (25, 17, 7, 15, 19, 11, 1, 9). Each base case has eight extended neighbors used in cutting.

  Eight extended neighbors have 256 different extended cases. Four different cut results are created for extended cases. Figure 4.9 shows the four cut results for the first base

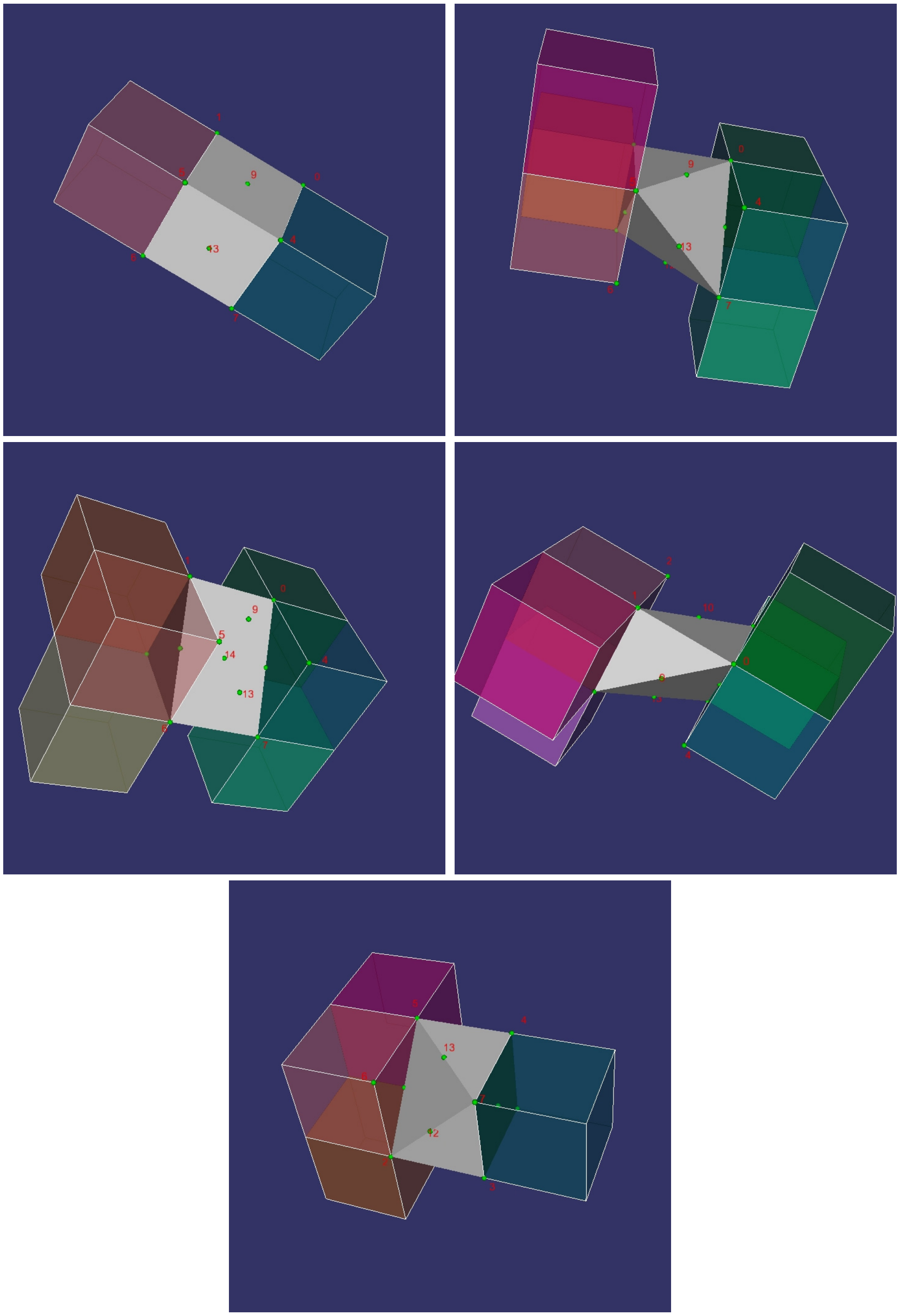

Figure 4.8: Cutting Results in Pattern (e)

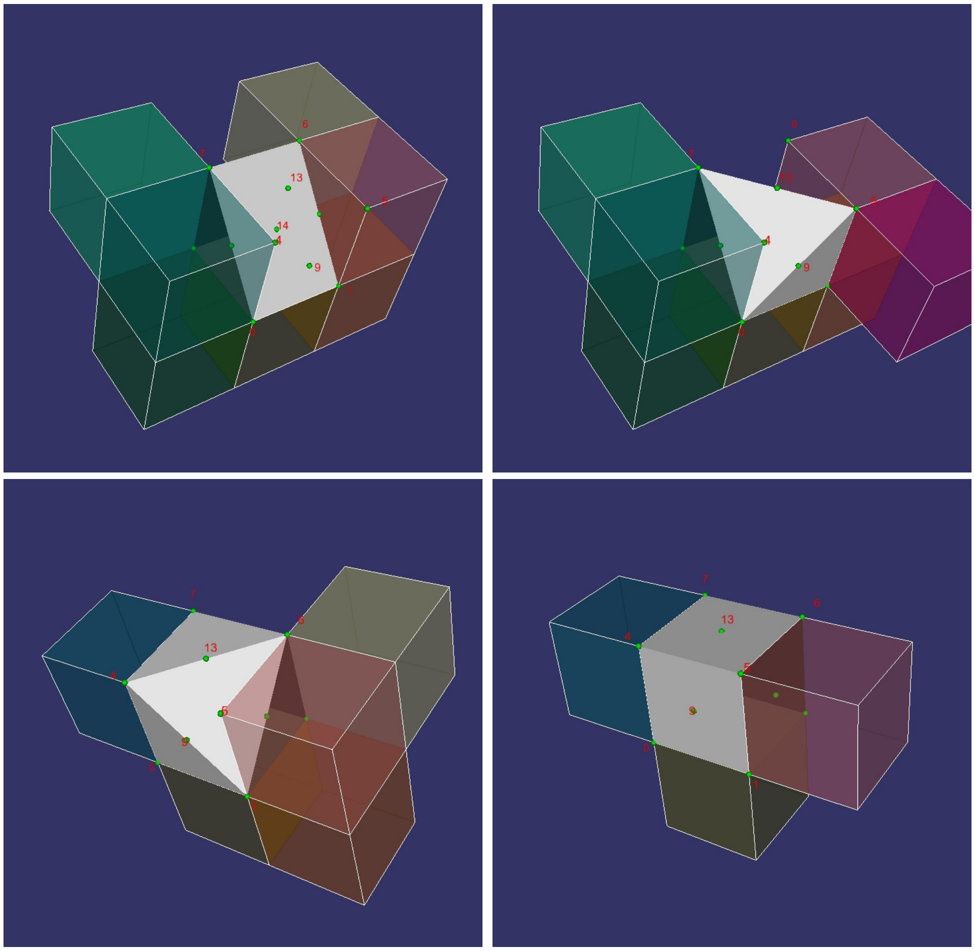

Figure 4.9: Cutting Results in Pattern (f)

case in the pattern. Extended neighbors are also shown in the figure. Other base cases have similar cutting results with rotation symmetry and complementary symmetry. In the figure, cutting results are numbered from left to right, top to bottom, starting from 0.

- **Pattern (g)**. Pattern (g) includes 12 different base cases (base case numbers: 15, 23, 27, 29, 39, 43, 46, 53, 54, 57, 58, 60). From the first one to the last one, we consider the following extended neighbors: (15, 21, 23, 17), (17, 25, 19, 11), (25, 15, 9, 19), (21, 9, 11, 23), (25, 23, 5, 7), (7, 3, 21, 25), (17, 5, 3, 15), (23, 19, 1, 5), (7, 17, 11, 1), (3, 1, 19, 21), (15, 7, 1, 9), (9, 3, 5, 11). Each base case has four extended neighbors

Table 4.5: Cutting Results, Default Triangle Indices, and Corresponding Extended Cases (Pattern (f))

Cutting Results	**Default Triangle Indices**	**Extended Cases**
0	(1,6,7),(1,7,0),(2,3,7),(2,7,6)	51,153
1	(5,2,7),(5,7,0),(5,0,1),(2,3,7)	57,147
2	(1,6,4),(6,7,4),(1,4,0),(7,6,2),(7,2,3)	3,19,35,67,83,99,115,131,163,179,195,211,227,243,9,25,41,73,89,105,121,137,169,185,201,217,233,249,48,49,50,52,53,54,55,56,58,59,60,61,62,63,144,145,146,148,149,150,151,152,154,155,156,157,158,159
3	(1,5,4),(1,4,0),(5,6,7),(5,7,4),(6,2,3),(6,3,7)	remaining extended cases

Table 4.6: Cutting Results, Default Triangle Indices, and Corresponding Extended Cases (Pattern (g))

Cutting Results	**Default Triangle Indices**	**Extended Cases**
0	(3,4,5),(3,5,2)	0
1	(7,2,3),(7,5,2),(4,5,7)	1,2,3,4,8,12
2	(3,4,5),(3,5,2),(7,4,3) (5,6,2)	remaining extended cases

used in cutting.

Eight extended neighbors have 256 different extended cases. Three different cut results are created for extended cases. Figure 4.10 shows the four cut results for the first base case in the pattern. Extended neighbors are also shown in the figure. Other base cases have similar cutting results with rotation symmetry and complementary symmetry. In the figure, cutting results are numbered from left to right, top to bottom, starting from 0.

Overall, for all patterns, we examine 6,187 different cases for surface reconstruction. All these cases are stored in a two-level lookup table: the first level of the table is indexed using

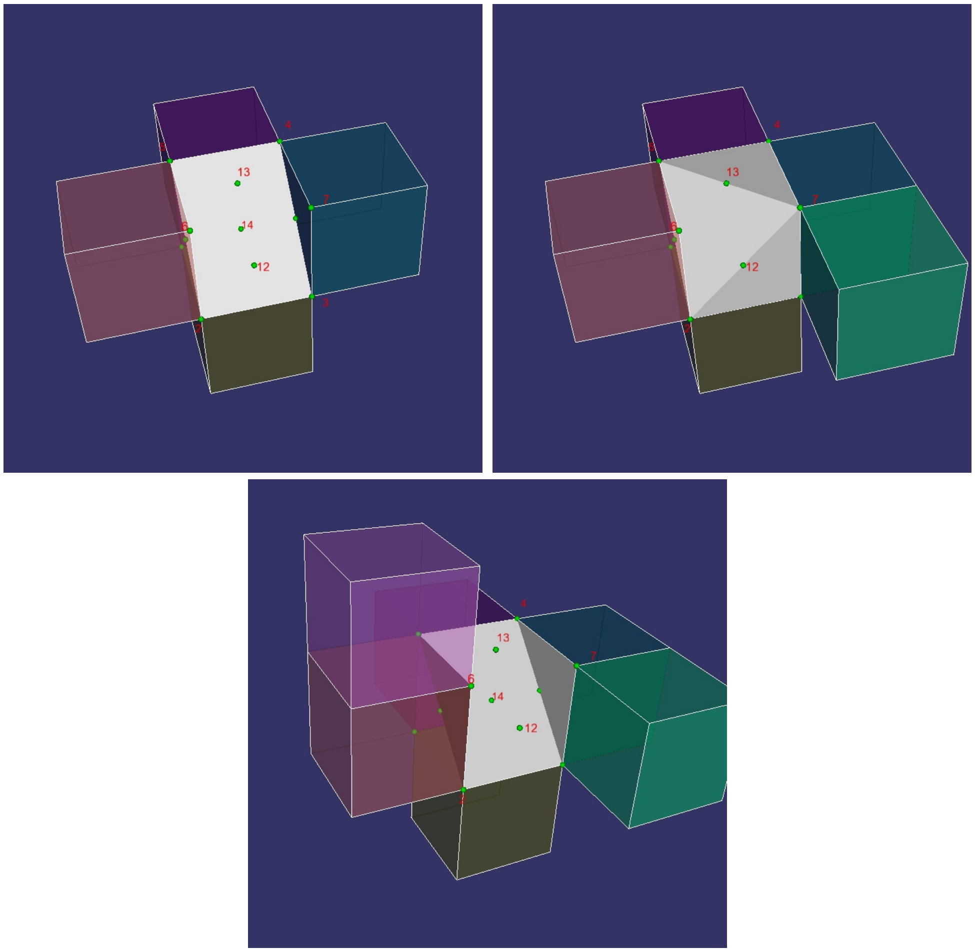

Figure 4.10: Cutting Results in Pattern (g)

the base case number, and the second level of the table is indexed using the extended case number. The table also stores extended neighbors used in each case.

4.3.2 Performance Analysis

Similar to the Marching Cubes method, the neighbor based surface reconstruction method also performances in a linear manner. The surface extraction process involves simple neighbor code analysis and lookup table searching for triangle mesh indices. The neighbor code analysis includes extracting 6-adjacent neighbor and other possible neighbors that can be done easily using C++ binary right-shift operator ($\gg$) and binary OR operator ($|$).

The lookup table searching requires two steps. In the first step, a 6-adjacent base case number is used to retrieve possible extended neighbors. If no extended neighbors are stored, a default set of triangle mesh indices is retrieved and the lookup table searching process ends. If there are extended neighbors stored, a second step is performed to access the second level in the lookup table using an extended case number. Finally triangle mesh indices are retrieved for surface reconstruction. Although this process involves one more step than in the Marching Cubes method, it is still a linear operation. Thus, the overall performance of the neighbor based surface reconstruction method is in the same order as the Marching Cubes method.

4.3.3 GPU Acceleration

Considerable work has been done to accelerate various algorithms on GPU, using shaders (parallel programmable processing units on a graphic hardware) [76][77][78][79][80]. Three different shaders (vertex shader, pixel shader and geometry shader) are available. The first two were introduced with DirectX 8, and the geometry was introduced recently with DirectX 10. Different from vertex shader and pixel shaders, a geometry shader processes primitives. Its input is a full primitive, meaning it has three vertices for a triangle, two vertices for a line, or a single vertex for a point. Given an input primitive, the geometry shader can discard the primitive, or emit one or more new primitives. Primitives generated by a geometry shader will be sent to a pixel shader. It is also possible to stream geometry shader generated primitives to the system memory for further processing. With the primitive producing capability of the geometry shader, we are able to implement our algorithm on GPU in a straightforward fashion.

On modern graphics cards, the number of shaders varies from single digit to hundreds. We write programs for a single shader. A graphics card with shaders runs the program in parallel on multiple shaders. In our application, we use CPU to process voxels one by one for surface reconstruction. By implementing our surface reconstruction algorithm on a graphics hardware, we are able to process multiple voxels at one time to speed up surface reconstruction. We have implemented our algorithm on a NVIDIA 8800 GTX video card using OpenGL and Cg 2.0. The graphics card has 128 stream processors. The card supports an unified shader model meaning that the 128 stream processors could be used as vertex shaders, geometry shader, or pixel shaders at different time when necessary. Once we issue voxels data to the graphics card, the card could process 128 voxels in a parallel manner simultaneously.

In our implementation, the input for a geometry shader is voxel position and encoded neighbor code. With the neighbor code, the geometry shader generates triangles based on cutting results in section 4.3.1. The geometry shader also generates normal for triangle vertices using the local neighbor gradient vector. In our cubic representation of voxels, a corner vertex of one voxel could have maximally eight voxels connected to it. On each of three coordinate axes, we count the number of connected voxels in both negative and positive directions, with the vertex as the origination point. Then, we estimate the gradient at the corner vertex (x, y, z) using central differences along the three coordinate axes:

$$
\begin{aligned}
G_x(x, y, z) &= \frac{N(x_+, y, z) - N(x_-, y, z)}{R_x} \\
G_y(x, y, z) &= \frac{N(x, y_+, z) - N(x, y_-, z)}{R_y} \\
G_z(x, y, z) &= \frac{N(x, y, z_+) - N(x, y, z_-)}{R_z}
\end{aligned}
\tag{4.4}
$$

In these equations, $N(x+, y, z)$ is the neighbor number at cuboid vertex (x, y, z) in the positive x direction, and R_x, R_y, R_z are the cuboid sizes in the X, Y, and Z directions. In surface reconstruction, we perform linear interpolation on the corner vertex normal to generate vertex normal for the surface patch.

Implementation Details

For surface reconstruction, we pass voxel positions and neighbor codes into GPU. For each voxel, we pack voxel space coordinates and neighbor code into a 4-element integer vector. Voxel space coordinates are integer indices in three dimensions. For example, in a 256^3 volume, the first voxel has voxel space coordinates [0, 0, 0] and the last voxel has voxel space coordinates [255, 255, 255]. As mentioned in section 4.3.1, each neighbor code is a 32-bit integer in which 27 bits are used to encode a voxel plus 26 neighbors. We then pack these vectors into a vertex array and pass the array to the vertex shader.

Passing Large Integers into GPU In OpenGL, integer numbers are converted to float numbers before entering the graphics pipeline. In the graphics pipeline, OpenGL converts these float numbers back to integer numbers. This process has no problem when we use integer numbers within the resolution of a 32-bit float number. For a 27-bit integer neighbor code, the corresponding integer number is large enough to go beyond the resolution of a 32-bit float number. For example, integer 134217726 represents a voxel missing only neighbor 0, and the number may become 134217728.0 after converting it from integer to float (tested on an Intel T5300 Core2 CPU). The converted float number is meaningless in our application, because the maximum value of a 27-bit integer neighbor code is 134217727. To solve the problem, in application we avoid integer-to-float conversion by directly copying integer neighbor code into a float using the C function *memcpy*. Then in the shader programs, we use Cg function $floatToRawIntBits$ to return the raw 32bit integer representation of an IEEE 754 floating-point number. In this way, we pass in 27-bit integer neighbor code to GPU correctly.

Surface Reconstruction on Shaders We write a vertex shader program to calculate the world space voxel position from voxel space coordinates. We pass in two uniform variables from the application to the shader program. These uniform variables store the volume origin position and volume resolution. Then we use the following formula to calculate world space position:

$$P_w = P_v \times P_r + P_o, \tag{4.5}$$

where P_w is world space coordinates, P_v is voxel space coordinates, P_o and P_r are 3-float vectors for volume origin position and volume resolution. Volume origin position is the world space coordinates of voxel [0, 0, 0]. Volume resolution specifies the size of a single voxel in X, Y and Z directions. The vertex shader outputs voxel world position and neighbor code into the geometry shader.

The surface reconstruction is done in the geometry shader. Based on the voxel world position calculated in the vertex shader, we create the 15 vertices (section 4.3.1). We then use these vertices to generate triangles for visible surface of the voxel. These vertices include 8 vertices at the corner, 6 vertices at face central, and 1 vertex at the central portion of the box. These vertices are shown in Figure 4.4. We also generate normal for these vertices using equations mentioned above.

We encode cutting results for all different cases into a 256 by 140 two-dimensional rectangle texture. In the C++ application, we create and bind the texture to texture unit in the OpenGL graphics pipeline. In the texture, we encode three vertex indices of one triangle into one texel. The R, G, and B components of one texel are used for the first, second, and third vertex index of one triangle. Since we need accurate R, G, and B components for triangle vertex indices, we do not want the rectangle texture to be filtered. In the geometry shader, we use Cg function *texRECTfetch* to get non-filtered texel from the rectangle texture. The geometry shader outputs generated triangles using Cg function *emitVertex*. The function outputs vertex position with optional vertex attributes such as colors, texture coordinates and normal. In our geometry shader program, we call emitVertex function three times per triangle to output the entire triangle. Along with the vertex position, we also output the vertex texture coordinates and the vertex normal. Vertex texture coordinates are calculated using the following formula:

$$T_v = \frac{P_v - P_o}{L}, \tag{4.6}$$

where T_v is the vertex texture coordinates, P_v is the vertex world position, P_o is the origin of the volume, and L are lengths of the volume in X, Y and Z dimensions.

Results In our experiment, we use a 256^3 volume with origin position (0, 0, 0). The volume has a resolution $0.9mm \times 0.75mm \times 0.7mm$. The human part in the volume is a human liver. On a 1.73G Hz Intel Core2 CPU, the volume is processed in 1.2 seconds for surface generation. The generated surface includes 760k triangles. On the NVIDIA 8800GTX graphics card, we could process the same volume within 0.4 seconds. The performance gain is rather limited. The bottle neck is in passing primitives into the pixel shaders, which is done in the *emitVertex* function calling. The NVIDIA 8 series graphics card is the first generation of graphics card to support the geometry shader and the unified shader model. We expect to see better performance on coming graphics cards that have more stream processors and a higher core speed.

4.4 Double Thresholding Marching Cubes

The Marching Cubes method was designed to work on scalar volumes in which adjacent voxels with different scalar values may suggest the existence of an iso-surface. On color volumes the assumption does not hold anymore, because voxels are now associated with R, G, B colors instead of single scalar. Due to the non-linearity of color space, it is difficult to map R, G, B colors to single scalar for iso-surface extraction.

An alternative way is to use binary segmentation volumes. A binary segmentation volume is created from the segmented color volumes. Instead of storing R, G, B colors for each voxel, a binary segmentation volume stores value 1 for foreground voxels and stores value 0 for background voxels. Then the Marching Cubes algorithm is used to extract an iso-surface with scalar value between 0 and 1, for example, 0.5. However, when there are two or more different parts close together in the same binary segmentation volume, for example, human rib and lung, then the Marching Cubes is not able to distinguish them since they have the same scalar value 1.

Figure 4.11 illustrates the limitation of the Marching Cubes on human parts segmentation. In the figure, the empty (white) region has a scalar value 0.0, the light gray region has a scalar value 0.2, and the dark gray region has a scalar value 0.5. For iso-surface extraction using thresholding value 0.2, the result is drawn as red lines. For iso-surface extraction using thresholding value 0.5, the result is drawn as cyan lines. The actual boundary surface for the light gray region with scalar value 0.2 is drawn as yellow lines. It is easy to recognize that the original Marching Cubes method is not able to extract the whole boundary of the light gray region. The same thing happens on the segmentation index volumes, which use integer scalar values instead of float point scalar values.

Given a threshold t, the original Marching Cubes is equivalent to applying a step filter on volume data:

$$f(x) = \begin{cases} 0, & x < t; \\ 1, & x \geq t \end{cases} . \tag{4.7}$$

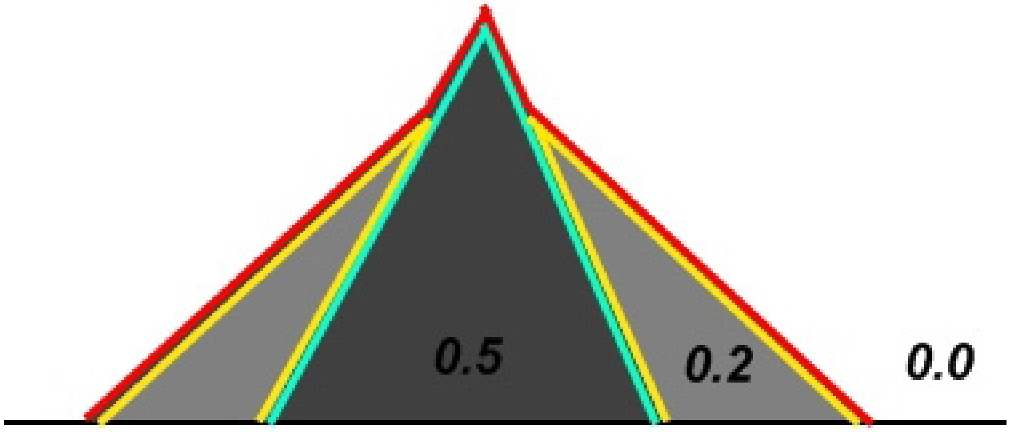

Figure 4.11: Marching Cubes Iso-surface Extraction (results drawn as 2D outlines)

We modify the original Marching Cubes method by using a box filter on volume data:

$$f(x) = \begin{cases} 0, & x < t_{low}; \\ 1, & t_{low} \leq x \leq t_{high}; \\ 0, & x > t_{high} \end{cases} . \tag{4.8}$$

Instead of using a single threshold t for vertex classification, we use two thresholds: t_{low} and t_{high}. Like the original Marching Cubes, cube vertices with values below the t_{low} are considered outside the surface. However, only cube vertices with values ($t_{low} \leq v \leq t_{high}$) are considered on or inside the surface. Cube vertices with values exceed the t_{high} are also considered outside the surface.

The modified Marching Cubes method is called Double Thresholding Marching Cubes that can extract surfaces on segmentation index volumes including multiple segmented human parts. One remaining issue is that Marching Cubes method produces open surface when the volume boundary intersects human parts. The reason is because the Marching Cubes method does not check voxels on the boundary of the volume. The open surface is acceptable for rendering purpose only. However, for human part surface models, we prefer the neighbor based surface reconstruction, which always produces close surface with the same speed of the Marching Cubes.

4.5 Rendering Human Parts

4.5.1 Sample of Segmented Human Parts

Figure 4.12 shows samples of segmented human parts. Starting from left to right, top to bottom, human parts are brain, liver, lung, bone, system artery, and muscles.

4.5.2 High Quality Rendering Using Texture Bricking

For accurate cadaver appearance of human parts, we use color volumes created from cryosection images for three-dimensional texture mapping on the extracted surface models. It is very easy to create large size three-dimensional textures overwhelming the texture memory on the mainstream graphics hardware. For example, a 512^3 color volume takes 512 MB storage space while mainstream graphics hardware nowadays has 256 MB or less video memory.

In volume rendering, the most straightforward method to deal with a large volume is the divide and conquer method, also called bricking. The large volume is subdivided into several blocks in such a way that a single sub-block (brick) fits into video memory. Texture bricks are stored in the main memory and sorted in a front-to-back or back-to-front manner depending on the rendering order.

We use the same concept to deal with the large-size color volumes in surface model rendering. First, we subdivide the large color volume into smaller sub-volumes. We then subdivide surface mesh into sub-meshes based on the bounding box of sub-volumes. In rendering, we render sub-meshes one by one with corresponding sub-volumes used for three-dimensional texture mapping.

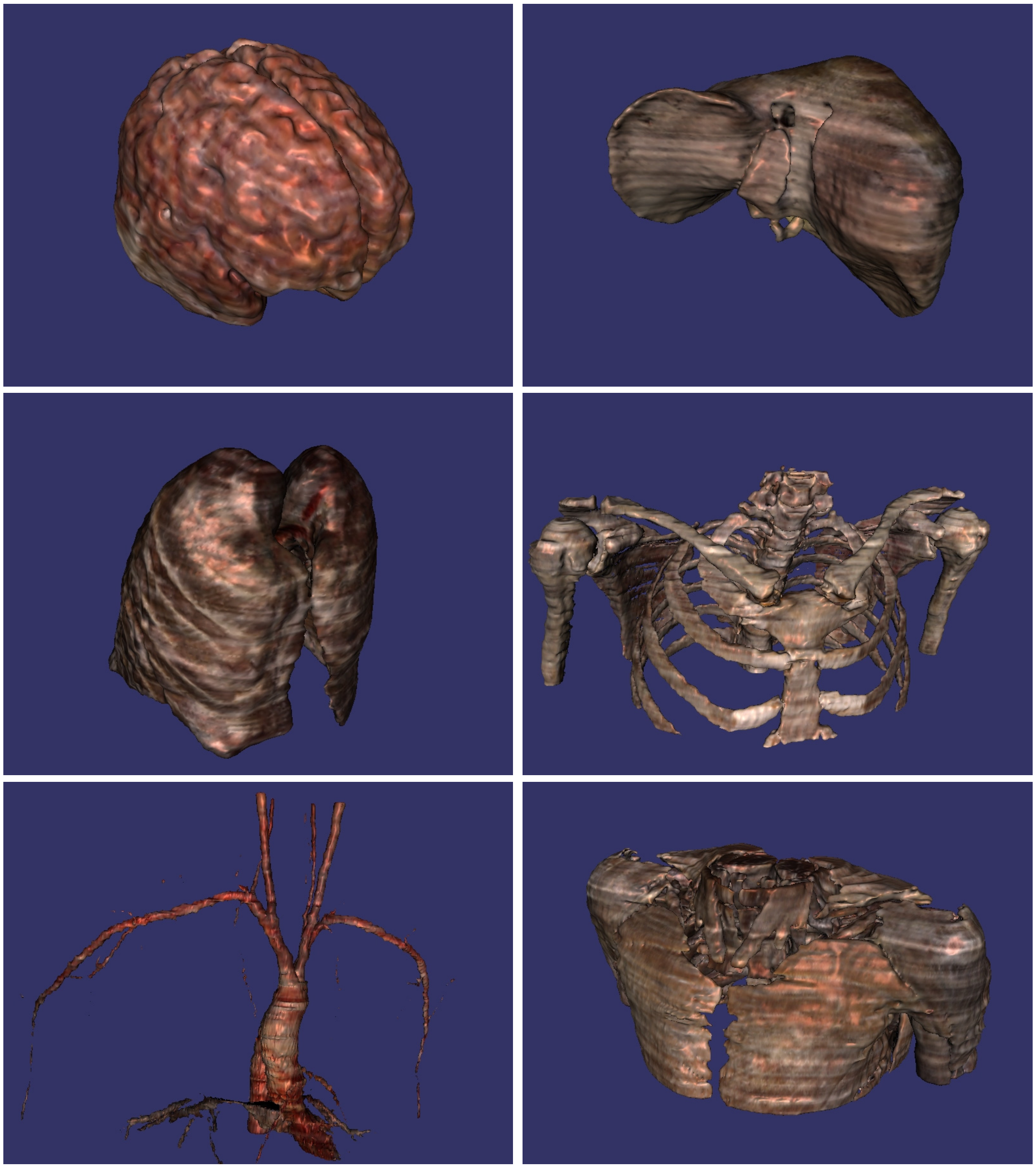

Figure 4.12: Sample of Segmented Human Parts

Dealing with Boundary

When we subdivide surface mesh into sub-meshes, inevitably some boundary triangles will belong to multiple sub-meshes. To avoid ambiguity in subdivision, boundary triangles are copied to all sub-meshes sharing the triangle. Without further processing, these triangles will produce a zig-zag effect because part of these triangles are outside the bounding-box of the corresponding sub-volumes, thus not being textured properly.

The problem is solved by using OpenGL clip planes. In OpenGL, minimally six user-defined clip planes are available. For each sub-mesh, we define clip planes based on the bounding box of the corresponding color volume. The graphics hardware automatically clips part of triangles outside the bounding box.

Results

On a graphics card with 256 MB memory, we successfully mapped a 512^3 three-dimensional texture to a human brain model. The texture needs a 384 MB storage space that is larger than the available graphics hardware. Figure 4.13 shows the rendering of high resolution texture mapping of human brain. Figure 4.14 shows the rendering of low resolution texture mapping of the same part. Figure 4.15 shows the side by side comparison of high and low resolution texture mapping. In the figure, the left part is high resolution texture mapping and the right part is low resolution texture mapping.

4.5.3 Mesh Smoothing

In our neighbor based surface reconstruction method, we trade off mesh smoothness for run-time performance. We agree that for viewing purposes a smooth mesh is more preferable than a mesh with block-like effect. For each segmented human part, we export extracted mesh to be smoothed. When users want to view human parts only, we load a pre-smoothed surface for rendering. When people initiate manipulation on human parts, we load extracted volume for surface reconstruction at run time.

In practice, we use VTK filter, *vtkWindowedSincPolyDataFiler*, for meth smoothing.

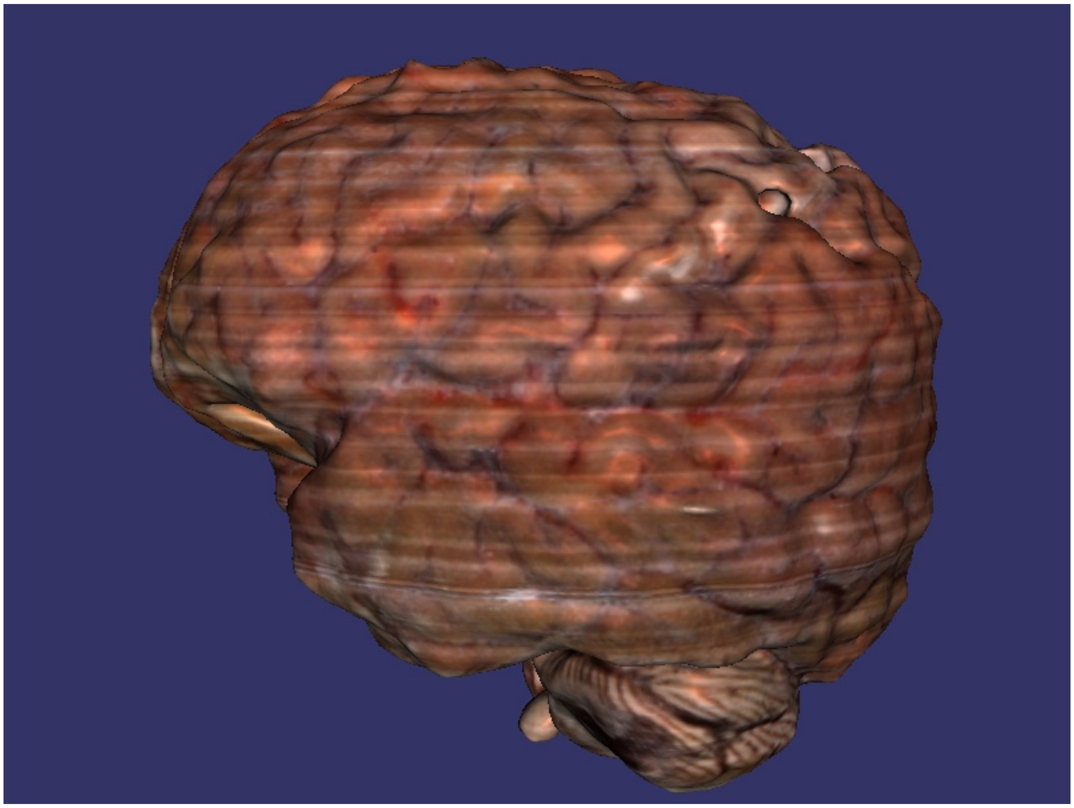

Figure 4.13: Human Brain with High Resolution Texture Mapping

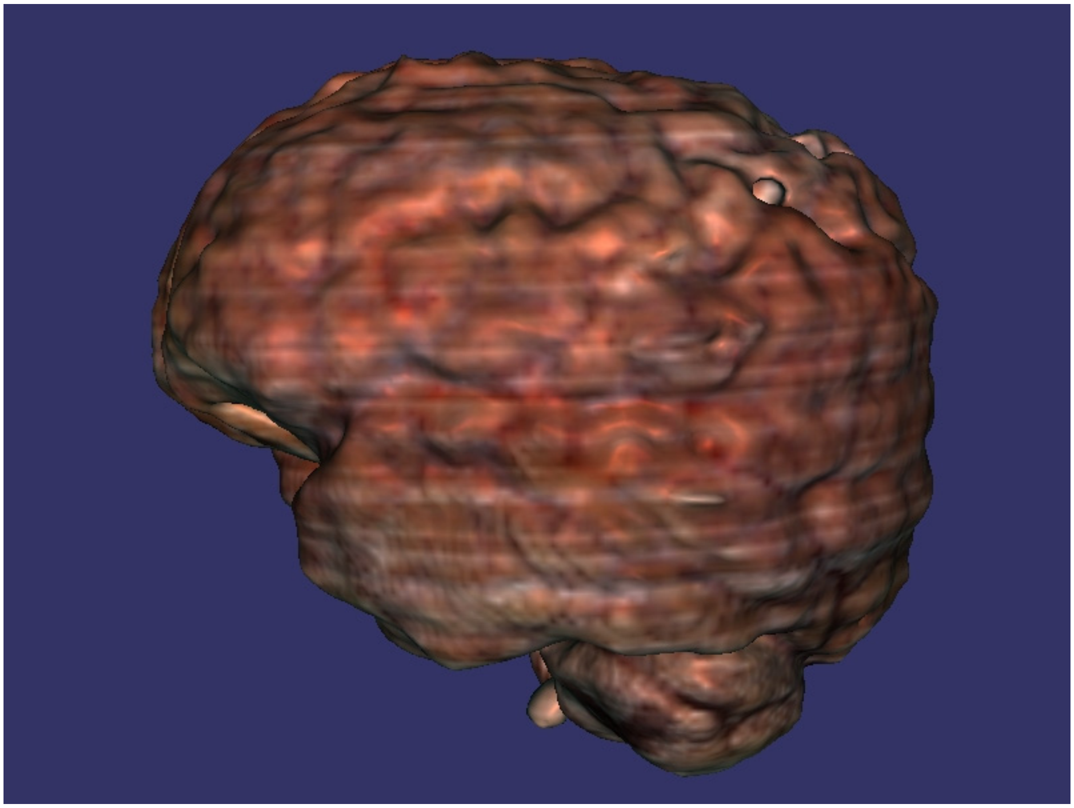

Figure 4.14: Human Brain with Low Resolution Texture Mapping

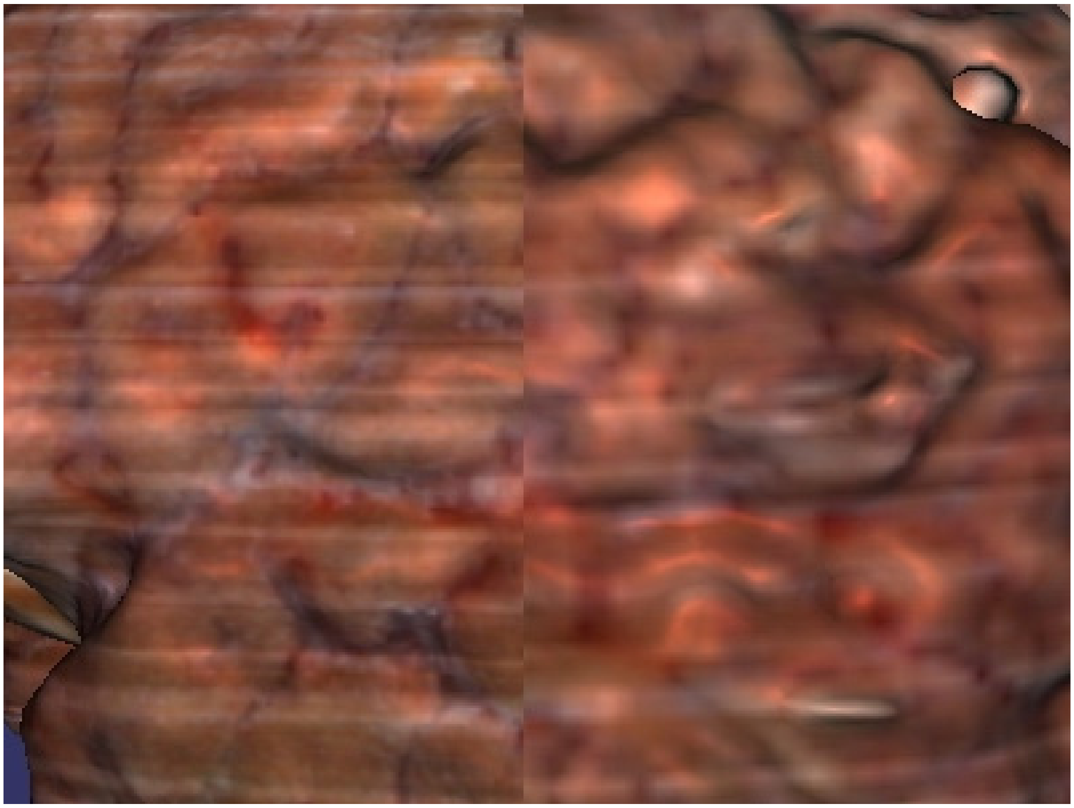

Figure 4.15: Side by Side Comparison of High and Low Resolution Texture Mapping

The *vtkWindowedSincPolyDataFiler* filter adjusts point coordinate using a windowed *sinc* function interpolation kernel. The effect is to relax the mesh, making the cells better shaped and the vertices more evenly distributed. The algorithm proceeds as follows. For each vertex v, a topological and geometric analysis is performed to determine which vertices are connected to v, and which cells are connected to v. Then, a connectivity array is constructed for each vertex. The connectivity array is a list of lists of vertices that directly attach to each vertex. Next, an iteration phase begins over all vertices. For each vertex v, the coordinates of v are modified using a windowed *sinc* function interpolation kernel [81].

Chapter 5: Virtual Human Anatomy Learning and Exploration

This chapter is organized as follows. In the first section, we introduce new exploring tools to virtual humans. The second section focuses on human anatomy learning in virtual environment. Finally, our implementation of surgery simulation is discussed in the third section.

5.1 Exploring Virtual Humans

5.1.1 Human Parts Manager

The human Parts Manager maintains labels and descriptions of human parts models of the segmented human parts. Using the Human Parts Manager, views can determine each part's name, volume model file, number of voxels included, and description. They can also customize appearance of each part by changing material attributes. Students can easily focus on different parts by changing their transparency and visibility.

In VHASS, users can either render reconstructed human parts using natural colors or manually assigned colors. Human parts rendered using assigned colors are easily distinguishable from other parts. At runtime, users can switch from natural colors to assigned artificial colors and vice versa. Figure 5.1 shows the anatomy structure of the human neck using both assigned artificial colors (left image) and a combination of natural colors and assigned artificial colors (right image). We can easily distinguish among skin, bones, muscles, and other parts. In the right image, bones and muscles are rendered with natural colors and other parts with assigned artificial colors.

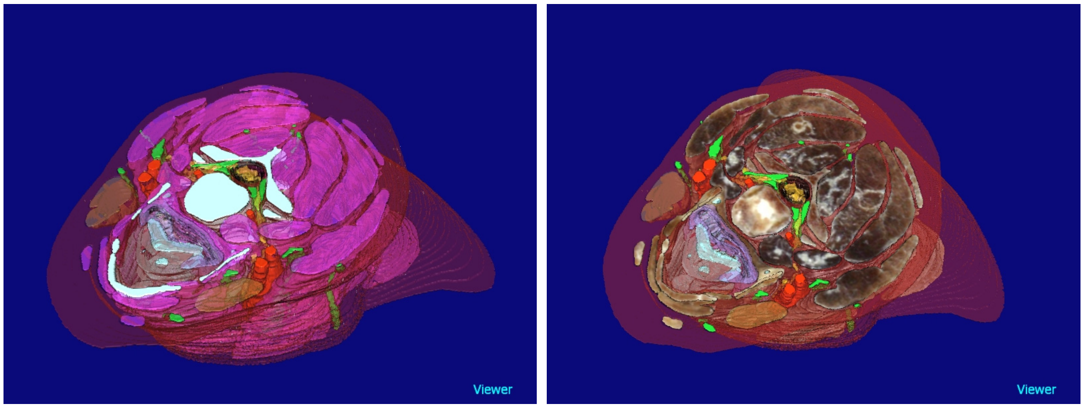

Figure 5.1: Human Parts Appearance Control

5.1.2 Cutting Tools

The virtual cadaver cutting tool lets users quickly remove large portions of human parts for human anatomy learning. VHASS provides multiple geometry shapes for cutting operations, including a box, sphere, and cylinder. One single geometry shape defines one region for cutting. Users can adjust the size, location, and orientation of each shape to define the cutting region. Our cutting tool provides two different operations, inclusive and exclusive, for each shape. Inclusive cutting operations cut off the portion of the human parts outside the cutting shape. Exclusive cutting operations cut off the portion of human parts inside the cutting shape. Figure 5.2 shows a cutting operation on the human neck based on a cutting box.

5.1.3 Scanning Virtual Human Body

One powerful tool that VHASS provides is the Realtime Human Body Scanner, which generates natural-appearance human body cross sections in real-time. Although CT and MRI images let medical students see through the human body, neither one can generate a natural-appearance cross section. Using our newly generated three-dimensional volume models for human parts, people can view continuous human body scanning with an interactive frame rate.

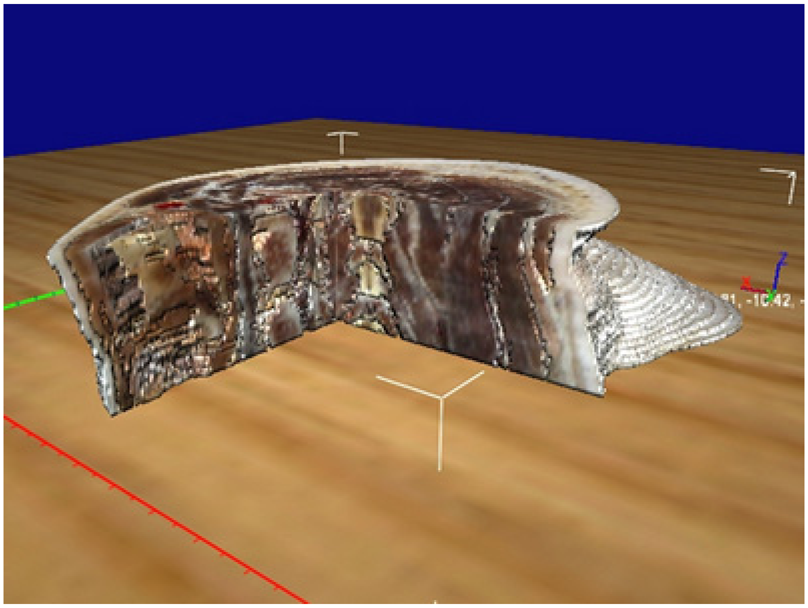

Figure 5.2: Cutting Tool Result

We define the scanning operation using a scan plane and the target human part. The scan plane's parameters include normal vector, origin coordination, motion mode, movement speed, and direction. Normal vector defines the portion of the target part the dissection plane cuts off; during scanning, the portion at the front of the plane, which the normal vector points to, will be cut off. Origin coordination defines the starting location. Movement parameters such as motion mode, speed, and direction control the movement of the scanning plane to produce different cross-section images.

To enable scanning, users need to select human parts to be scanned and then define a scan plane. The scanner could scan multiple human parts using different scan planes at the same time. We create an OpenGL clipping plane from the scan plane to cut off voxels at the front of the plane. Because the OpenGL driver supports at least six custom clipping planes, users could concurrently scan at least six human parts.

During scanning, we discover voxels intersecting the scan plane for cross-section generation. For each intersected voxel, we compute intersecting points on the voxel's edges. We generate cross sections for the voxel by connecting these points to be a triangle mesh.

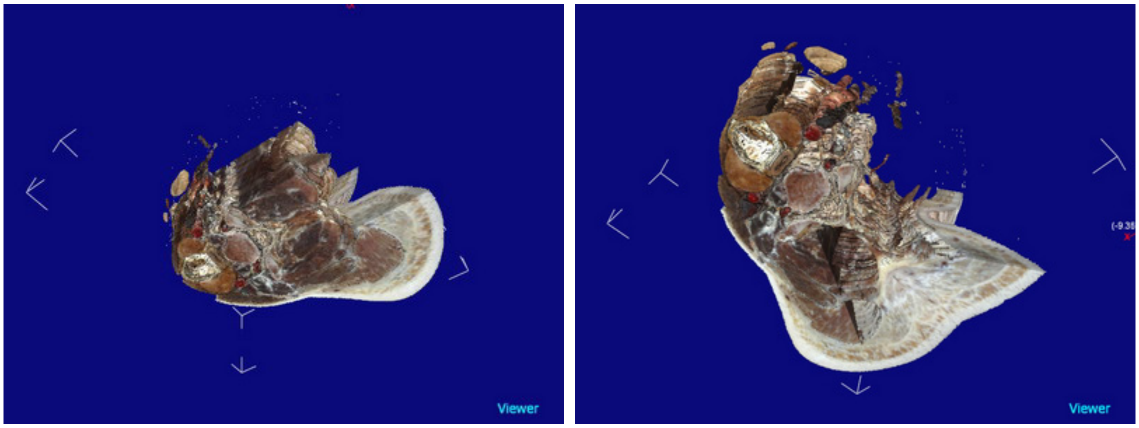

Figure 5.3: Scanning Human Parts

Figure 5.3 shows intermediate scanning results on human neck.

5.1.4 Sliced View

We have developed a new function, three-dimensional real-time sliced view, in VAHSS. The three-dimensional real-time sliced view function generates a sliced three-dimensional view for human parts at run time. Users can control the number of slices as well as position and thickness of each slice. Users are also able to select one slice for detailed examination and interactions, such as cutting. The three-dimensional sliced view function provides another way for VHASS users to explore inner structure of human parts with a photo-realistic rendering effect.

To improve performance of the three-dimensional sliced view tool, we generate a surface model for slices using GPU acceleration. We create neighbor-code volumes from our extracted human parts volumes. The neighbor code volume has the same dimension and resolution as the corresponding human part volume. Instead of storing voxel colors, voxels in a neighbor code volume store the voxel neighbor code that is stored in a 32-bit integer. Once the user has defined parameters, such as total number of slices and thickness of each slice, we perform slicing on the neighbor volume. Then we use the sliced neighbor volume to generate surface for slices.

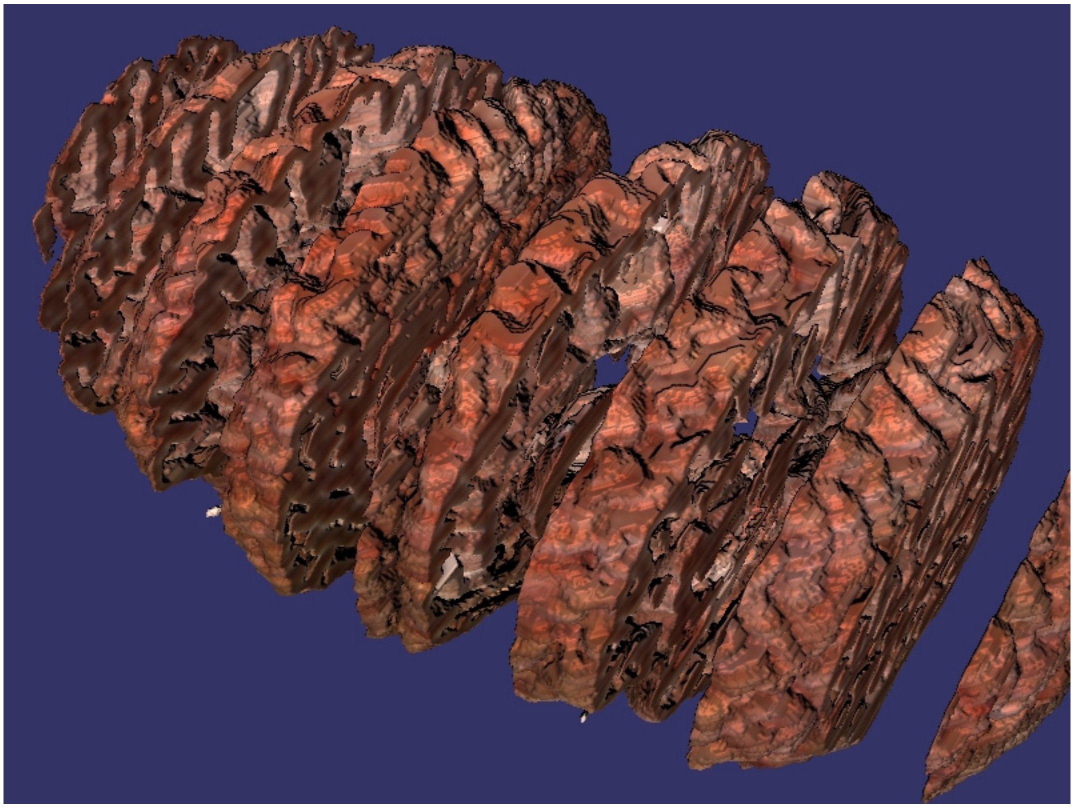

Figure 5.4: Sliced View of Human Brain

A sample slicing process is shown in Figure 5.4. We calculate bounding boxes for slices based on user specified slice position and thickness. For each bounding box, we examine voxels inside the bounding box. For each voxel, we check if there is any neighbor falling outside the bounding box. If one neighbor falls outside the bounding box, we update the corresponding bit in the neighbor code to be "0". Then we run our neighbor-based algorithm on the updated neighbor code volume to generate surface for slices.

5.1.5 Volume Exploration Tool

In direct volume rendering of unsegmented cryosection volumes, we have implemented a volume exploration tool using a spatial classification transfer function. Similar to Mueller's work [82], our tool uses spatial classification to enhance the anatomical structures of interest. In their work, Mueller et al. use additional three-dimensional textures to store the spatial classification function. We have implemented the spatial classification transfer function on GPU to provide more flexibility while saving graphics memory space from extra three-dimensional volumes.

Controlled by parameters such as center position and radius, the volume exploration tool defines a spherical region inside the cryosection volume. The center of the region has the maximum opacity while voxels outside the region have minimum zero opacity. Voxels inside the region have opacity interpolated using the distance between the current voxel and the center of the region. The opacity estimation results of the tool and the other opacity transfer functions are summed together using weight factors. Parameters of the tool are passed into GPU using uniforms. At run time, people can change location, size, and weight factor of the tool for different enhancements of the anatomical structures of interest.

Figures 5.5 shows the female head of the Visible Human Project data set. The left eye has been enhanced by the exploration tool.

5.2 Human Anatomy Learning in Virtual Environment

5.2.1 Stereo Display

Traditional display technology projects three-dimensional objects onto a two-dimensional surface and eliminates depth illusion of the objects. Stereoscopy technology, on the other hand, is capable of recording three-dimensional visual information or creating the illusion of depth in an image. The illusion of depth in a photograph, movie, or other two-dimensional image is created by presenting a slightly different image to each eye. Many three-dimensional displays use this method to convey images. Three-dimensional display was first invented by Sir Charles Wheatstone in 1840.

Traditional stereoscopic photography consists of creating a three-dimensional illusion starting from a pair of two-dimensional images. The easiest way to create depth perception in the brain is to provide the eyes of the viewer with two different images, representing two perspectives of the same object, with a minor deviation similar to the perspectives

Figure 5.5: Volume Exploration Tool

that both eyes naturally receive in binocular vision. On a computer, graphics hardware is of generating two such images by using slightly different viewpoints in rendering. Many different devices have been developed to help people to view such images. Some of them are introduced below.

LCD Shutter Glasses

LCD shutter glasses are glasses used in conjunction with computers or TV screens to create the illusion of a three-dimensional image, an example of stereoscopy. Glass containing liquid crystal and a polarizing filter has the ability to become dark when voltage is applied, but otherwise it is transparent. A pair of eyeglasses can be made using this material and connected to a computer video card. The video card alternately darkens over one eye and then the other in synchronization with the refresh rate of the monitor, while the monitor alternately displays different perspectives for each eye. This is called Alternate-frame sequencing. At sufficiently high refresh rates, the viewer's visual system does not notice the flickering; each eye receives a different image, and the effect is achieved.

Anaglyph Glasses

Anaglyph glasses are used with anaglyph images, which are made up of two color layers superimposed together. The two color layers are created to offset with each other to produce a depth effect. The final result contains two differently filtered colored images.

Viewing anaglyphs through appropriately colored glasses results in each eye seeing a slightly different picture. In a red-blue anaglyph, for instance, the eye covered by the red filter sees the red parts of the image as "white", and the blue parts as "black"; the eye covered by the blue filter perceives the opposite effect. True white or true black areas are perceived the same by each eye. The brain blends together the image it receives from each eye, and interprets the differences as being the result of different distances. This creates a normal stereograph image without requiring viewers to cross their eyes.

Autostereoscopic Three-dimensional Displays

Autostereoscopy is a method of displaying three-dimensional images that can be viewed without the use of special headgear or glasses on the part of the user. These methods produce depth perception in the viewer even though the image is produced by a flat device.

Several technologies exist for autostereoscopic three-dimensional displays. Currently most of such flat-panel solutions use lenticular lenses or parallax barrier. If viewers position their head in certain viewing positions, they will perceive a different image with each eye, giving a stereo image. Eye strain and headaches are usual side effects of long viewing exposure to autostereoscopic displays that use lenticular lens or parallax barriers. These displays can have multiple viewing zones allowing multiple users to view the image at the same time. Other displays use eye-tracking systems to automatically adjust the two displayed images to follow viewers' eyes as they move their head.

5.2.2 Our System

In our research, we have integrated a FaceSpace large scale stereo projector and an Intersense ultra sonic tracker for human anatomy learning in a virtual environment.

5.3 Surgery Simulation

5.3.1 Haptic Interaction

We use a SensAble Phantom Omni haptic device to provide haptics interaction in virtual surgery simulation. We use the SensAble OpenHaptics toolkit for haptics support. The OpenHaptics toolkit includes two major modules: 1) the Haptics Device API (HDAPI) and 2) the Haptic Library API (HLAPI). The HDAPI enable haptics programmers to render forces directly by providing low-level access to the haptic device. The HDAPI also controls the runtime behavior of the drivers. The HLAPI provides high-level haptic rendering. The design of the HLAPI is familiar to OpenGL API programmers since HLAPI is patterned after OpenGL syntax.

A common problem when using haptics devices is the refresh rate. A typical graphics application refreshes 30 - 60 times a second which means 30 - 60 frames per second, to provide continuous motion on the screen for human eyes. But a haptics application normally refreshes the forces rendered by the haptic device at approximately 1,000 times a second to provide the kinesthetic sense of stiff contact. Consequentially haptics and graphics rendering have to be performed in separate threads so that each thread can run at its respective refresh rate. When using the HDAPI, people need to create different threads for haptics and graphics and handle threading in their programs. We use the HLAPI in VHASS since, when using the HLAPI, threading will be handled by the HLAPI rendering engine.

The typical structure to enable haptics in an OpenGL application using HLAPI is first to set up OpenGL by creating a graphics rendering context. Then the program initializes the HLAPI by creating a haptics rendering context and connect it to a haptic device. The program also needs to specify how the physical coordinates of the haptic device should be mapped into the coordinate space used by the graphics rendering context. The HLAPI uses the mapping to map geometry specified in the graphics space to the physical workspace of the haptic device. After the scene graph is rendered using OpenGL, the program renders the haptics while capturing the geometry as a depth or feedback buffer shape. Finally the rendering loop continues by rendering the graphics again.

5.3.2 Drilling

The drilling burr used in a real dissection is a small tungsten-carbide or diamond burr attached to a drill. We use a polygonal geometry shaped like the real drilling burr to represent the virtual drilling burr. The shape of the geometry defines the region for modification.

Drilling operation is simulated by interactively positioning the virtual drilling burr in the volume of human parts. At each frame, voxels in human parts are processed sequentially to find out if they intersect the virtual drilling burr. In the segmentation index volume, voxels intersecting with the virtual drilling burr are marked to be negative while keeping the absolute value of the voxel unchanged. Then neighbors of the voxel will be updated to

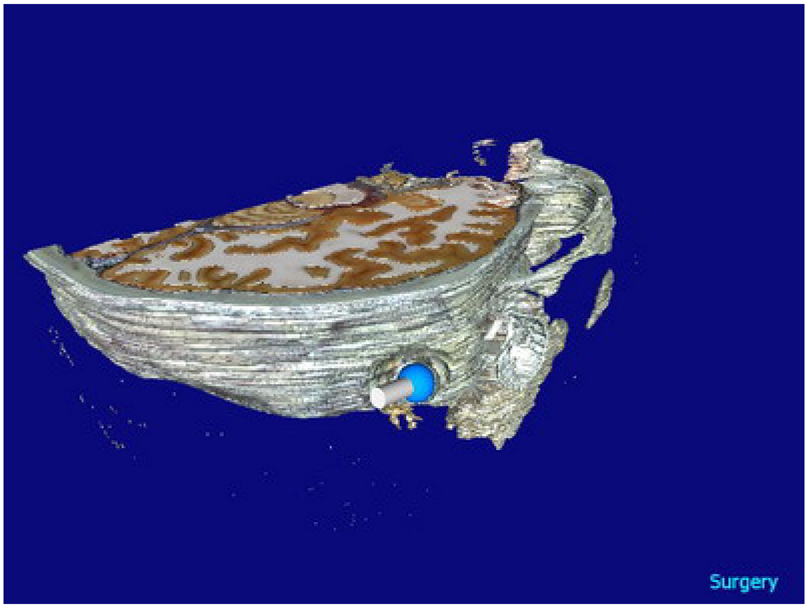

Figure 5.6: Drilling on Human Parts

reflect the change. After completion, users can restore the original human parts by making negative voxels to be positive voxels again.

To accelerate processing speed, an octree-based space partition is used to group voxels into sub-regions. When looking for intersecting voxels, we skip sub-regions that do not intersect the virtual drill burr. At each frame, sub-regions having one or more voxels intersected with the virtual burr are processed to regenerate surface. The tissue colors and textures of the newly generated surface are defined by associated color volumes. Figure 5.6 shows the intermediate results of the drilling operation on the human temporal bone structure.

Chapter 6: Future Work

Future work on our research are:

- **Improvement on direct volume rendering on cryosection volumes**. We need to do more on designing efficient transfer functions on photographic volume. One possible way is to analysis and compare existing color image edge detection methods for opacity estimation on cryosection images. To increase reality in volume rendering, we need to implement advanced rendering techniques, such as shading, shadow, and global illumination.

- **Improvement on neighbor-based surface reconstruction method**. Our new algorithms overcome limitations of the well-known Marching Cubes method while maintaining the same performance with the original Marching Cubes method. Future works on the new algorithm include smoothness improvement on generated surfaces and dynamic decimation to improvement run-time performance.

- **More tools for human parts exploration and interaction**. Our new models combine volume data and human parts segmentation together to provide accurate appearance of human parts in real-time deforming interaction such as cutting and drilling. We need to do more on virtual surgery simulation. Currently only one surgery operation, drilling, is simulated. Future tasks include more surgery operation simulation and more complete scenario in surgery training. To increase verisimilitude in surgery simulation, future tasks include but are not limited to, soft tissue simulation, deformation as well as fluid simulation such as blood.

- **New interaction and simulation based on our 3D models**. Volumes have advantages in better support to rigid interaction such as slicing and drilling. On the other hand, surfaces have advantages in better support to deformable object simulation such as soft tissue simulation. Combining volume and surface together provides a new way for interaction and simulation on virtual humans: implement rigid interaction on volumes and map back to surfaces; implement soft tissue simulation on surfaces and map back to volumes. We have finished framework to map rigid interaction result from volume to surface. Next step is to map soft tissue simulation from surface to volume.

- **Integration with virtual reality peripheral devices**. Better integration with virtual reality peripheral devices in another important step in future research. Currently our system supports haptic device and tracking devices. More devices, such as force feedback gloves, can increase reality in virtual human anatomy education.

Chapter 7: Summary and Conclusions

This dissertation intends to create a new platform for human anatomy learning. To overcome limitation and weakness of existing systems, we start from cryosection images to create new three-dimensional models of anatomical structures. Our new models support both actual cadaver appearance as well as free manipulation.

We have implemented two different visualization methods, direct volume rendering and surface reconstruction. For direct volume rendering of human parts, we have created a series of new opacity transfer functions. Direct volume rendering on photographic volumes (color volumes created from cryosection images) relies on the human eye to distinguish different human parts. Thus, the efficiency of highlighting human part boundaries directly affects the recognition of different anatomical structures in photographic volumes. Our new opacity transfer functions are more efficient than existing ones to highlight human part boundaries in photographic volumes.

We have combined together volume data and human parts segmentation to create new three-dimensional models. Our new models store boundary information implicitly and support deformable interactions. At run-time, we perform real-time surface reconstruction on the models for interaction results. We have observed the limitation of the Marching Cubs method in our environment. The step filter used in the Marching Cubes method makes it unable to extract a complete surface for human parts with specified scalar values because the filter picks all scalar values greater than a given threshold t. The iso-surfaces extracted by the Marching Cubes method can include human parts other than the current human part. Although it is acceptable for viewing iso-surfaces on radiological CT and MRI images, this limitation is not acceptable for human parts surface reconstruction. We have modified the original Marching Cubes method to use a box filter, which picks scalar values in between two threshold, t_{low} and t_{high}. The modified Double Tresholding Marching Cubes is able to extract complete and accurate boundary surface for each human part. We also have developed a new neighbor-based surface reconstruction algorithm. Our new method runs on the new models natively and performs like the Marching Cubes method, one of the fastest surface reconstruction methods.

Based on the newly created virtual humans, we have developed a series of human anatomy learning and exploration tools. These tools provides real-time manipulation on realistic virtual human parts. We also have implemented drilling simulation based on a SensAble Omni haptic device. Finally, we have finished a framework for virtual human anatomy learning in a virtual environment. With a large scale stereo projector and ultrasonic tracker, we provide immersive rendering and interaction on virtual humans.

Bibliography

Bibliography

[1] M. J. Ackerman, "The visible human project: A resource for anatomical visualization," in *Proc. of MEDINFO'98*, vol. 52, Studies in Health Technology and Informatics. Amsterdan: IOS Press, 1999, pp. 1030–1032.

[2] V. M. Spitzer and D. G. Whitlock, "The visible human data set: The anatomical platform for human simulation," *Anatomical Record*, vol. 253, no. 2, pp. 49–57, 1998.

[3] K. H. Höhne, M. Bomans, M. Riemer, R. Schubert, U. Tiede, and W. Lierse, "A 3d anatomical atlas based on a volume model," *IEEE Computer Graphics and Applications*, vol. 12, no. 4, pp. 72–78, 1992.

[4] Y. Lin, J. X. Chen, and Y. Liu, "Virtual human anatomy," *IEEE/AIP Computing in Science and Engineering*, vol. 7, no. 5, pp. 71–73, 2005.

[5] Y. Liu, J. X. Chen, and Y. Lin, "Real-time photo-realistic virtual human anatomy," *IEEE/AIP Computing in Science and Engineering*, vol. 10, no. 2, pp. 41–47, 2008.

[6] S. Spitzer, M. J. Ackerman, A. L. Scherzinger, and D. Whitlock, "The visible human male: A technical report," *Journal of the American Medical Informatics Association*, vol. 3, no. 2, pp. 118–130, 1996.

[7] K. H. Höhne, B. Pflesser, A. Pommert, M. Riemer, T. Schiemann, R. Schubert, and U. Tiede, "A new representation of knowledge concerning human anatomy and function," *Nature Medicine*, vol. 1, no. 6, pp. 506–511, 1995.

[8] A. Pommert, R. Schubert, M. Riemer, T. Schiemann, U. Tiede, and K. H. Höhne, "Symbolic modeling of human anatomy for visualization and simulation," in *Visualization in Biomedical Computing 1994, Proc. of SPIE*, vol. 2359, 1994, pp. 412–423.

[9] R. Schubert, K. H. Höhne, A. Pommert, M. Riemer, T. Schiemann, U. Tiede, and W. Lierse, "A new method for practicing exploration, dissection, and simulation with a complete computerized three-dimensional model of the brain and skull," *Acta Anatomica*, vol. 150, no. 1, pp. 69–74, 1994.

[10] A. Pommert, K. H. Höhne, B. Pflesser, M. Riemer, T. Schiemann, R. Schubert, U. Tiede, and U. Schumacher, "A highly realistic volume model derived from the visible human male," in *Proc. of The Third Visible Human Project Conference*, 2000.

[11] R. Schubert, B. Pflesser, A. Pommert, K. Priesmeyer, M. Riemer, T. Schiemann, U. Tiede, P. Steiner, and K. H. Höhne, "Interactive volume visualization using "intelligent movies"," in *Proc. of Medicine meets Virtual Reality (MMVR'99)*, vol. 62, Health Technology and Informatics. Amsterdan: IOS Press, 1999, pp. 321–327.

[12] J. Qualter, M. M. Triola, M. J. Weiner, M. A. Hopkins, M. Kirov, and M. S. Nachbar, “The virtual surgery patient: Development of a digital, three-dimensional model of human anatomy designed for surgical,” in *Proc. of the 17th IEEE Symposium on Computer-Based Medical Systems (CBMS’04)*, 2004.

[13] H. D. Chen, X. H. Jiang, Y. Sun, and J. Wang, “Color image segmentation: advances and prospects,” *Pattern Recognition*, vol. 34, no. 12, 2001.

[14] C. K. Yang and W. H. Tsai, “Reduction of color space dimensionality by moment-preserving thresholding and its application for edge detection in color images,” *Pattern Recognition Letters*, vol. 17, no. 5, pp. 481–490, 1996.

[15] B. Preim and D. Bartz, *Visualization in Medicine: Theory, Algorithms, and Applications.* Morgan Kaufmann, 2007.

[16] J. C. Russ, *The Image Processing Handbook, Fifth Edition.* CRC Press, 2006.

[17] H. J. Trussell, “Comments on ”picture thresholding using an iterative selection method”,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 9, no. 5, p. 311, 1979.

[18] R. Lukac and K. N. Plataniotis, *Color Image Processing: Methods and Applications.* CRC Press, 2006.

[19] J. M. Queen, “Some methods for classification and analysis of multivariate observations,” in *Proc. of the Fifth Berkeley Symposium on Mathematics, Statistics, and Probabilities*, vol. I, 1967, pp. 281–297.

[20] C. R. Brice and C. L. Fennema, “Scene analysis using regions,” *Artifical Intelligence*, vol. 1, pp. 205–226, 1970.

[21] R. Adams and L. Bischof, “Seeded region growing,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, pp. 641–647, June 1994.

[22] A. Mehnert and P. Jackway, “An improved seeded region growing algorithm,” *Pattern Recognition Letters*, vol. 18, no. 10, pp. 1065–1071, October 1997.

[23] S. W. Zucker, “Region growing: Childhood and adolescence,” *Computer Graphics and Image Process*, vol. 5, pp. 382–399, September 1976.

[24] D. Sinclair, “Voronoi seeded colour image segmentation,” AT&T Laboratories, Cambridge, United Kingdom, Technical Report TR99-4, 1999.

[25] H. Palus and D. Bereska, “Region-based colour image segmentation,” in *Proc. of the Fifth Workshop Farbbildverarbeitung*, 1999, pp. 67–74.

[26] J. Serra, *Image Analysis and Mathematical Morphology.* London, UK: Academic Press, 1982.

[27] H. Digabel and C. Lantuejoul, “Interative algorithms,” in *Proc. of 2nd European Symposium on Quantitative Analysis of Microstructures in Material Science*, 1978, pp. 85–99.

[28] L. Vincent and P. Soille, "Watersheds in digital spaces: an efficient algorithm based on immersion simulations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 6, pp. 583–598, 1991.

[29] E. N. Mortensen, B. S. Morse, W. A. Barrett, and J. K. Upuda, "Adaptive boundary detection using livewire two-dimensional dynamic programming," *IEEE Computer in Cardiology*, pp. 635–638, 1992.

[30] J. K. Udupa, S. Samarasekera, and W. A. Barrett, "Boundary detection via dynamic programming," in *Proc. of Visualization in Biomedical Computing*, 1992, pp. 33–39.

[31] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathe-matik*, vol. 1, pp. 269–271, 1959.

[32] Y. Kim and S. C. Horli, *Handbook of Medical Imaging: Display and PACS.* SPIE Press, 2000, vol. 3.

[33] J. A. Sethian, *Level Set Methods and Fast Marching Methods.* Cambridge, UK: Cambridge University Press, 1999.

[34] B. B. Avants and J. P. Williams, "An adaptive minimal path generation technique for vessel tracking in cta/ce-mra volume images," in *Proc. of Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 1935. Lecture Notes in Computer Science, 2000, pp. 707–716.

[35] C. M. van Bemmel, L. Spreeuwers, M. Viergever, and W. Niessen, "Level-set based carotid artery segmentation for stenosis grading," in *Proc. of Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 2489. Lecture Notes in Computer Science, 2002, pp. 36–43.

[36] X. Zeng, L. Staib, R. Schultz, and J. Duncan, "Survey: interpolation methods in medical image processing," *IEEE Transactions on Medical Imaging*, vol. 18, no. 10, pp. 100–111, 1999.

[37] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, no. 6, pp. 311–317, June 1975.

[38] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf, "Real-time volume graphics," Course Notes 28, 2004.

[39] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, May 1988.

[40] R. A. Drebin, L. Carpenter, and P.Hanrahan, "Volume rendering," in *Proc. of SIGGRAPH '88*, 1988, pp. 65–74.

[41] D. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling: A Procedural Approach.* Academic Press, July 1998.

[42] A. Kaufman, "Voxels as a computational representation of geometry," in *The Computational Representation of Geometry. SIGGRAPH '94 Course Notes*, 1994.

[43] N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, June 1995.

[44] C. M. Wittenbrink, T. Malzbender, and M. E. Goss, "Opacity-weighted color interpolation for volume sampling," in *Proc. of IEEE Symposium on Volume Visualization*, 1998, pp. 135–142.

[45] J. F. Blinn, "Jim blinn's corner: Image compositing theory," *IEEE Computer Graphics and Applications*, vol. 14, no. 5, 1994.

[46] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *Proc. of IEEE Symposium on Volume Visualization*, 1994, pp. 91–98.

[47] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization," in *Proc. of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[48] E. LaMar, B. Hamann, and K. I. Joy, "Multiresolution techniques for interactive texture-based volume visualization," in *Proc. of IEEE Visualization '99*, 1999.

[49] K. Engel, M. Kraus, and T. Ertl, "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading," in *Proc. of Graphics Hardware*, 2001.

[50] S. Fang, T. Biddlecome, and M. Tuceryan, "Image-based transfer function design for data exploration in volume visualization," in *Proc. of IEEE Visualization '98*, October 1998, pp. 319–326.

[51] I. Fujishiro, T. Azuma, and Y. Takeshima, "Automating transfer function design for comperhensible volume rendering based on 3d field topology analysis," in *Proc. of IEEE Visualization '99*, October 1998, pp. 467–470.

[52] G. Kindlmann and J. W. Durkin, "Semi-automatic generation of transfer functions for direct volume rendering," in *Proc. of 1998 Volume Visualization Symposium*, Oct. 1998, pp. 79–86.

[53] K. Kwon and B. Shin, "Visualization of segmented color volume data using gpu," in *Proc. of ICAT 2006*, 2006.

[54] D. Ebert, P. Rheingans, and T. Yoo, "Designing effective transfer functions for volume rendering from photographic volume," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 2, pp. 183–197, 2002.

[55] S. Wang and H. Li, "Gvf-based transfer functions for volume rendering," in *Computer Graphics International*, 2006, pp. 727–734.

[56] C. Xu and J. L. Prince, "Snakes, shapes, and gradient vector flow," *IEEE Transactions on Imaging Processing*, vol. 7, no. 3, pp. 359–369, March 1998.

[57] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active contour models," *International Journal of Computer Vision*, vol. 1, no. 4, pp. 321–331, January 1988.

[58] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, November 1986.

[59] P. Perona and J. Malik, “Scale-space and edge detection using anisotropic diffusion,” *IEEE Transactions on Pattern Analysis Machine Intelligence*, vol. 12, pp. 629–639, 1990.

[60] S. Grossberg, “Neural dynamic of brightness perception: Features, boundaries, diffusion, and resonance,” *Perception and Psychophysics*, vol. 36, no. 5, pp. 428–456, 1984.

[61] L. Ibanez and W. Schroeder, *The ITK Software Guide 2.4*. Kitware, Inc., 2005.

[62] J. Krüger and R. Westermann, “Acceleratoin techniques for gpu-based volume rendering,” in *Proc. of IEEE Visualization 2003*, 2003, pp. 287–292.

[63] H. Scharsach, “Advanced gpu raycasting,” in *Proc. of CESCG 2005*, 2005, pp. 69–76.

[64] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3d surface construction algorithm,” *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 193–169, July 1987.

[65] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill, “Octree-based decimation of marching cubes surfaces,” in *Proc. of the 7th conference on Visualization’96*. IEEE Visualization, 1996.

[66] G. M. Nielson and B. Hamann, “The asymptotic decider: Resolving the ambiguity in marching cubes,” in *Proc. of the 2nd conference on Visualization’91*, 1991, pp. 83–91.

[67] G. M. Nielson, “On marching cubes,” *IEEE Transations on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 283–297, 2003.

[68] ——, “Dual marching cubes,” *IEEE Visualization 2004*, pp. 489–496, 2004.

[69] A. Lopes and K. Brodlie, “Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 1, pp. 16–29, 2003.

[70] N. Amenta, M. Bern, and M. Kamvysselis, “A new voronoi-based surface reconstruction algorithm,” in *Proc. of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, 1998, pp. 415–421.

[71] T. K. Dey, J. Giesen, and J. Hudson, “Delaunay based shape reconstruction from large data,” in *Proc. of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*, 2001, pp. 19–27.

[72] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, “Surface reconstruction from unorganized points,” in *Proc. of the 19th Annul Conference on Computer Graphics and Interactive Techniques*, 1992, pp. 71–78.

[73] J. Wang, M. M. Oliveira, and A. E. Kaufman, “Reconstructing manifold and nonmanifold surfaces from point clouds,” *IEEE Visualization 2005*, pp. 415–422, 2005.

[74] G. T. Herman and H. K. Liu, "Three-dimensional display of human organs from computed tomograms," *Computer Graphics and Image Processing*, vol. 9, no. 1, pp. 1–21, 1979.

[75] L. S. Chen, G. T. Herman, R. A. Reynolds, and J. K. Udupa, "Surface shading in the cuberille environment," *Computer Graphics and Applications*, vol. 5, no. 12, pp. 33–43, 1985.

[76] L. Wang, C. Tu, W. Wang, X. Meng, B. Chan, and D. Yan, "Silhouette smoothing for real-time rendering of mesh surfaces," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 3, pp. 640–652, 2008.

[77] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker, "A streaming narrow-band algorithm: Interactive computation and visualization of level sets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 4, pp. 422–433, 2004.

[78] J. Kruger, P. Kipfer, P. Kondratieva, and R. Westermann, "A particle system for interactive visualization of 3d flows," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 6, pp. 744–756, 2005.

[79] S. P. Callahan, M. Ikits, J. L. Comba, and C. T. Silva, "Hardware-assisted visibility sorting for unstructured volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 3, pp. 285–295, 2005.

[80] H. Zhang and A. Kaufman, "Interactive point-based isosurface exploration and high-quality rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1267–1274, 2006.

[81] G. Taubin, T. Zhang, and G. H. Golub, "Optimal surface smoothing as filter design," in *Proc. of the 4th European Conference on Computer Vision*, vol. 1, 1996, pp. 283–292.

[82] D. Mueller, A. Maeder, and P. O'Shea, "Implementing direct volume visualization with spatial classification," in *Proc. of of the Australian Pattern Recognition Society (APRS) Workshop on Digital Image Computing (WDIC 2005)*, 2005, pp. 49–53.

Curriculum Vitae

Yanling Liu was born on August 27, 1976. He received his Bachelor of Science from Southwest Jiaotong University, China in Computer Science in 1994. He received his Master of Science from Southwest Jiaotong University, China in Communication Engineering in 2001. His research interests include computer graphics, virtual reality, and medical imaging and visualization.