THE UNIVERSITY OF CALGARY

Interactive Volume Manipulation

by

Hung-Li Jason Chen

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

January, 2008

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Interactive Volume Manipulation" submitted by Hung-Li Jason Chen in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Supervisor,
Dr. Faramarz F. Samavati
Department of Computer Science

External Examiner
Dr. Steven Boyd
Department of Mechanical and Manufacturing Engineering

Co-Supervisor,
Dr. Mario Costa Sousa
Department of Computer Science

Dr. Przemyslaw Prusinkiewicz
Department of Computer Science

Date

Abstract

Interactive volume manipulation and exploration is an essential and important process in a number of application domains such as medicine, geophysics, computational fluid dynamics. Conventional interactive methods typically demand significant amounts of time and do not lend to natural interaction scheme with the 3D volume. In addition, for most of these methods, having a real-time feedback is a challenge.

In this thesis, we present a framework for real-time 3D volume manipulation and exploration. We propose sketch-based interfaces and introduce tools for direct drilling, peeling, cutting/pasting, and segmenting the 3D volume. In our approach, the user first selects a volume tool and sketches a region to manipulate on the volumetric data. This process can be repeated as many times as necessary until a desired result is obtained. Next, the user places strokes directly on the visible surface to perform region growing segmentation. To prevent unexpected segmentation, the sculpted volume is used to constrain the region growing process. In addition, we provide a set of playback functions for fine-tuning the segmentation result.

In our method, we utilize a point radiation technique as a fundamental sculpting primitive. This technique is used for (1) a new mask-based sweep volume that is further developed into a collection of volume manipulation tools and (2) interactive seeded region segmentation. The proposed tools were inspired by traditional medical illustrations and allow for direct drilling, lasering, peeling, and cutting/pasting the 3D volume. Our sketch-based system utilizes GPU programming to achieve real-time processing for both rendering and volumetric sculpting.

Acknowledgement

I would like to thank Dr. Faramarz Samavati and Dr. Costa Sousa for accepting me as one of their graduate student. They are outstanding in both teaching and supervising. Without their help, I would not be able to publish the paper, Sketch-Based Volumetric Seeded Region Growing. They helped me a lot in the writing and provided me with great support for the travel. The trip to Vienna was amazing and lots fun. I could not forget that I helped Dr. Samavati to get lost at midnight in the Vienna city. But it was not all that bad after the Danube dinner; at least we got the chance to uncover the breathtaking St. Stephen's cathedral at night. See, there is also a benefit without taking a taxi! I like Dr. Costa Sousa's interpretation of getting off the air plane from the back—it's a doubly linked list!

I would also like to thank my parents. Without their support, I would not be able to accomplish anything. I also want to thank my brother; he gave me a lot of suggestions, and I also gain great knowledge from him; he has always been my raw model. In addition, I need to thank my aunt, uncle, and Tony, they really helped me a lot. Most of all, I really appreciate the great support and encouragement from my girlfriend, Wendy, who has been and will always be an important part of my living forward.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In computer graphics, volumetric data is referred as a collection of elements organized in a rectilinear 3D grid. Each element on the grid point represents a certain property of the underlying data source. Its applications often appear in the context of medicine, geophysics, physics, and engineering. The complexity of volumetric data often involves iteratively scanning and processing through the entire input 3D grid. The increasingly large amount of information to be processed has been a great challenge for modern computing systems.

For medical imaging systems, such as computerized tomography (CT), magnetic resonance imaging (MRI) and ultrasound, the volumes produced can easily reach the size of $512 \times 512 \times 512$; and as a consequence, an iteration of over 134 million elements is required. Moreover, when interaction with the data is desired, the processing can be quickly multiplied to 4027 million computations for a target speed of 30 frames per second to show a smooth animated result. In addition, interactive manipulation of such complex models are often essential for helping users explore and understand the shapes, features and relationships among the obscured internal materials. However, as most operations require sophisticated analysis of the volume data, this incurs a greater burden on the originally large processing count. Therefore, development of new processing and visualization techniques is needed to help solve these critical issues.

1.1 Motivation

In this research, we were inspired by traditional medical illustrations depicting clinical procedures using cutaways, peeling, drilling to abstract complex structures and visualize effectively the targeted internal structures (or features). An example is given in Figure 1.1 illustrating a brain surgery. The surgeon first incises the scalp and locates the injury site. Next, the left frontal bone is removed, and the dural flap is cut open to reveal the contused and lacerated brain tissue. At last, the surgeon resects portions of the left frontal lobe of the brain and removes it from the skull. After observing a number of similar medical illustrations, we realize the same importance of manipulation for medical volume data sets. Therefore, question arises: How does one create intuitive tools to facilitate operations like opening the skull or segmenting the gray and white matter?

An ideal volume manipulation environment would allow users to intuitively operate directly in 3D, using a variety of tools for manipulating the volume, with real-time feedback and high-quality rendering. Creating such interactive volume tools is not an easy task. Existing interactive volume manipulation techniques usually do not lend to natural interaction scheme with the 3D volume. In addition, for most of these methods, having a real-time feedback with high-quality rendering is a challenge. Due to their natures, volumetric data sets have large amount of features but usually obscured and hidden. Revealing, visualizing and extracting these features require appropriate tools. Rather than introducing the abstract tools, it would be more natural to create tools that emerge in our daily life. This brings up the aspect of interactive implementation of the tools that is a challenge for the volumetric data

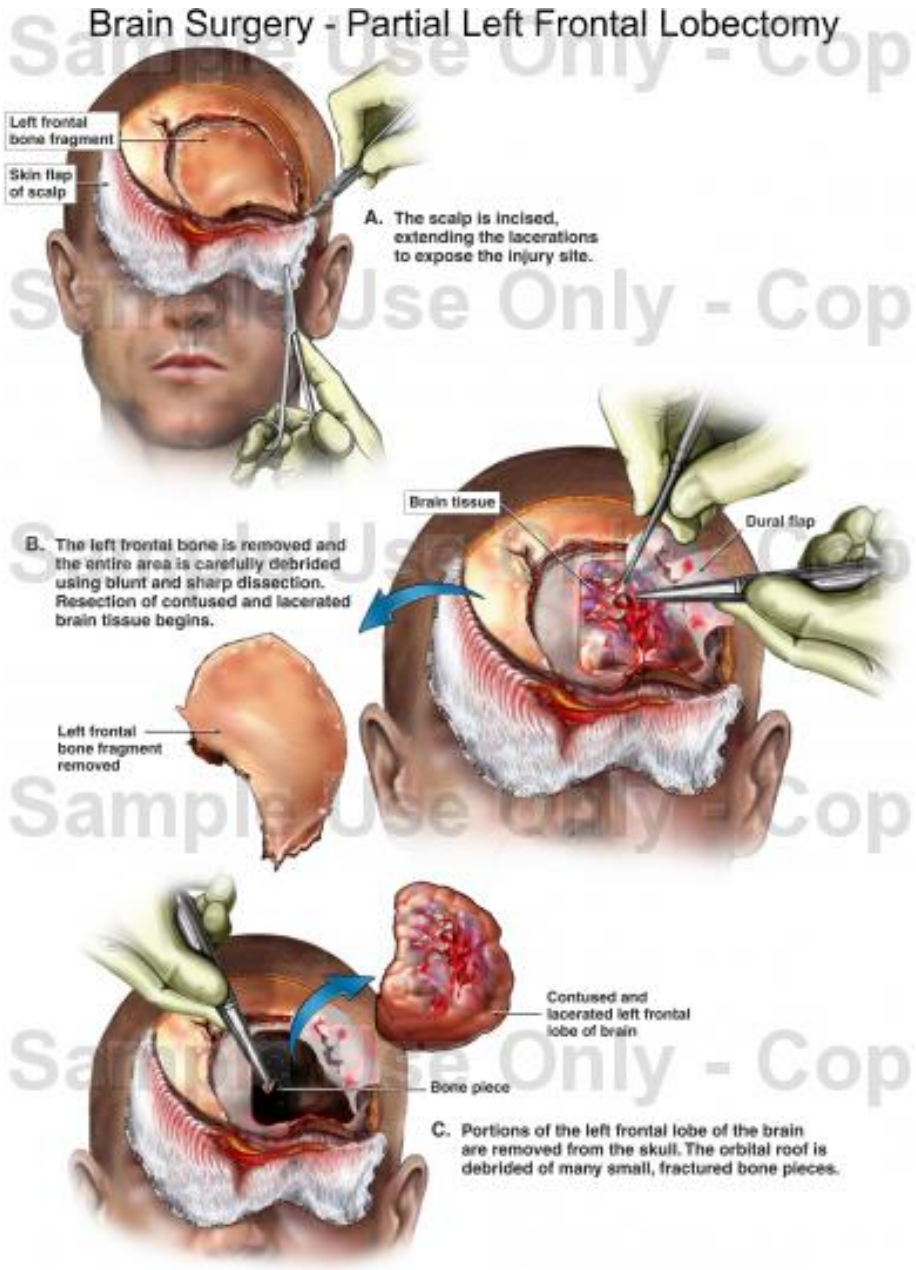

Figure 1.1: Medical illustration of brain surgery—partial left frontal lobectomy. The exhibit shows the exposure of the front of the skull, debridement of the fracture site, and removal of a portion of the frontal lobe. (Copyright ©2007 Nucleus Medical Art. All rights reserved. www.nucleusinc.com)

sets due to their large sizes. In order to provide ease-of-use and suitability for a majority of the tasks, intuitiveness and flexibility should be taken into consideration when creating such tools.

To satisfy the requirement of intuitiveness, one possibility is to utilize sketch-based interfaces. However, the development of sketch-based systems for volumetric data is a challenging problem. One issue as other sketch-based systems is the mapping from a 2D sketch to a 3D volume. If the specification of the third dimension requires much user intervention, the advantage of adapting a sketch-based approach can be forfeited. The other challenge is achieving interactive responses in order for the tools to be effective. Due to the size of volumetric data sets, which often requires computationally expensive operations such as feature look-up in neighboring voxels and blending with additional volumes, real-time feedbacks can be difficult to obtain.

To allow flexible volume manipulation, it would be suitable to create tools that can be freely moved in the 3D space in order to explore any portion of the volume. One strategy is to exploit volume sweeping, in which the tool is presented by a 2D or 3D object in space and swept along an arbitrary path. This approach provides flexible tool displacement. However, as discussed in the general context of swept volumes [1], interactivity has been a major obstacle for sweeping in large volumetric data sets. This is because the large amount of elementary comparisons that are performed between the tool object and the data volume.

In order to effectively visualize the manipulation results, high-quality rendering is necessary especially in the applications where precise validation is required, such as medical imaging. Voxel splatting presents one possibility for volume rendering. In splatting, only the voxels that are visible are rendered and can largely reduce storage

on the graphics hardware. However, the quality and speed of splatting depend on the number of displayed voxels and can create artifacts in the blended image. As an alternative volume rendering technique, ray-casting provides high-quality rendering however with the storage requirement of the entire data volume. To minimize the usage of graphics resources, binary volumes can be stored but may result in aliasing artifacts.

1.2 Approach

In this work, we propose novel interfaces for interactive volume manipulation and sculpting. The user sketches directly over the displayed 3D volume with a pen tablet instead of manipulating with a 3D device. We also present a GPU-based system that processes the user interaction in real-time. Figure 1.2 illustrates the key stages of the volume manipulation/exploration capabilities of our system. In this example, the goal is to segment specific parts of a brain data set. Upon loading the volume data, the user directly sketches over the displayed image to mark the region for uncovering (Figure 1.2, a). The system applies the peeling mask to cut through surface layers with user-defined depths (Figure 1.2, b). The left frontal portion of the skull is cut out, and the user directly sketches seeds on the visible cerebrum (Figure 1.2, c). The region starts to grow (Figure 1.2, d), and finally the complete segmentation, if it is desired, is obtained and moved out of the skull (Figure 1.2, e).

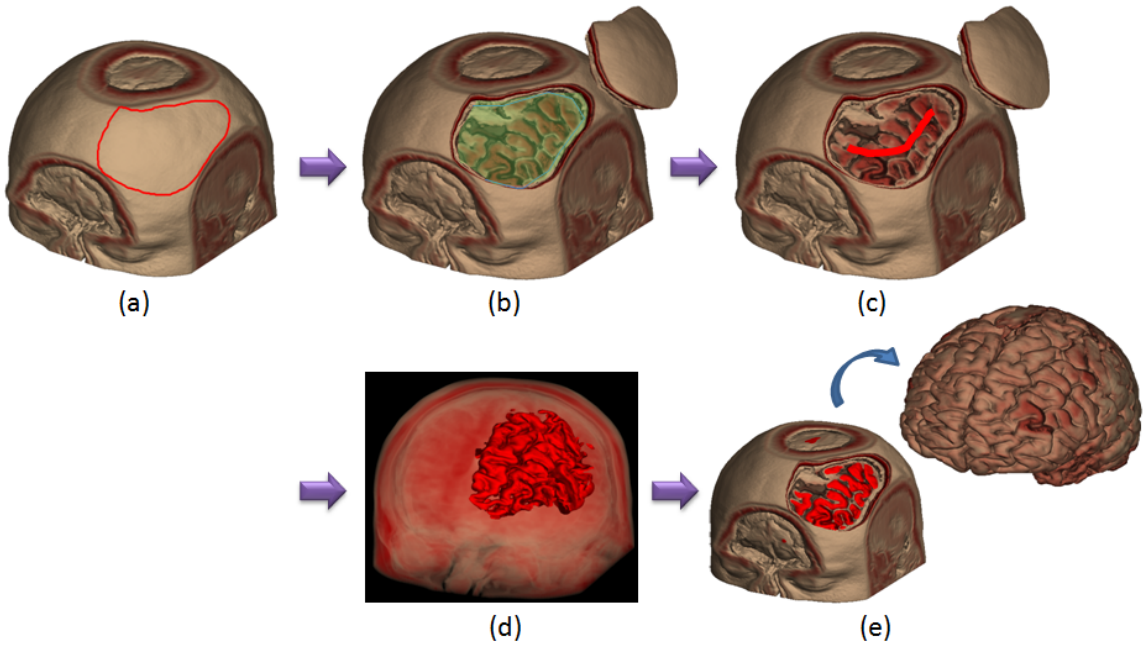

Figure 1.2: An example of our sketch-based interactive volume manipulation: user sketches directly over the data to indicate a region for opening (a), the system performs a surface-based peeling operation with user specifying the depth (b), the skull is removed and the user sketches seeds for segmentation (c), the region grows (d), and the grey and white matter are segmented and removed from the brain (e).

1.3 Methodology

Due to the large volume data size to be processed, it is challenging to fulfill the various manipulation requirements while achieving the desired interaction speed. In this work, we propose a GPU-based point radiation technique for interactive volume manipulation with ray-casting. We address the aliasing problem by using a higher quality volume space. The point radiation algorithm takes a point in space or on the tool object and radially distributes energy into volumetric grid cells. The result is recorded in a radiation volume (i.e. a 3D texture organized in the form of arrays of values in the graphics memory) and referenced by the ray-casting engine

to distinguish between the visible and occluding parts of the volumetric data. The point radiation method completely removes aliasing rendering artifacts and provides an extensible real-time framework for volume manipulation.

Figure 1.3 depicts an overview of our interactive volume manipulation system. We store the entire volumetric data as a 3D texture on the GPU and adapt a real-time ray-casting method for rendering. We introduce a point radiation technique that uses points as data primitives (Figure 1.3, a). We extend based on the concept of 2D point splatting and utilize the geometry shader to radiate points directly to any 3D location in the volume space. To provide flexible sculpting tools that can be easily constructed with 2D strokes, we utilize a mask-based sweep volume method (Figure 1.3, b). In our approach, points inside the sketched mask are radiated, and the mask is swept along an arbitrary path. As the mask sweeps, the intersecting elements from the volume are removed in a real-time fashion. With variations from the mask-based sweeping scheme, we further develop a collection of volume tools (Figure 1.3, c) in our system: (1) the drilling tool, which cuts through the volume with a constant direction from the surface points; (2) the laser tool, which orients and positions automatically on the volume surface and cuts along the negative gradient direction; and (3) the peeling tool, which detects surface points covered by the mask and allows individual mask points to radiate along the detected normals. Furthermore, we introduce the cut and paste tool (4), which is used to illustrate the cut-away portion by dual-rendering the radiation volumes.

Based on the point radiation technique, we also propose a framework of volume segmentation tools (Figure 1.3, d). Our main methodology is the seeded region growing algorithm. We enable interactive seeded region segmentation by utilizing the

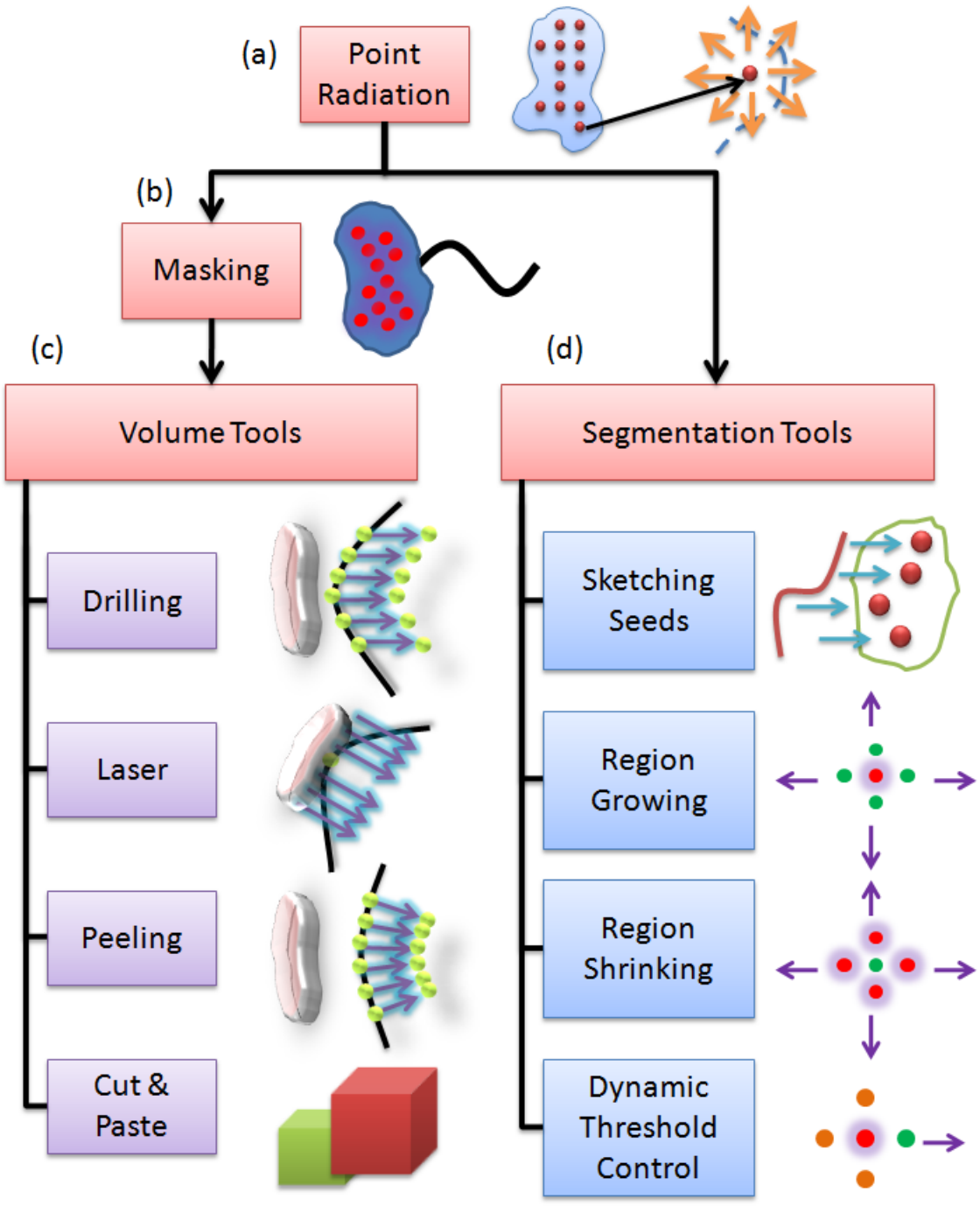

Figure 1.3: Overview of our interactive volume manipulation system. (a) the point radiation technique is the fundamental building block of a masking scheme along a path that is used in two applications: a collection of interactive volume tools for (c) sculpting and (d) segmenting.

geometry shader on the GPU for direct access to the neighbors. The core components include: (1) sketching seeds, which allows the user to place multiple sketches directly over the displayed volume as a simple indication of the desired segment, and then the system detects the seeds via raycasting; (2) region growing, which is performed and processed directly on the GPU to allow interactive control (e.g. pause, play, resume, and stop); (3) region shrinking, which reverses the region growing process when an undesired segmentation result is observed; and (4) dynamic threshold control, which enables the user to fine-tune threshold values while simultaneously performing the region growing operation.

1.4 Contributions

The key contribution of our work is a framework of interactive tools for real-time 3D volume manipulation, sculpting, and segmentation in the context of high-quality ray-cast rendering. Specific contributions include (1) a novel GPU-based point radiation technique as a fundamental volume sculpting primitive, (2) a new mask-based sweep volume, (3) a suite of interative volume tools for volume sculpting, probing, clipping, and cutting including view-dependent tools for drilling, lasering, peeling, cutting/pasting, (4) sketch-based seeded region growing method and playback functions for interactive volume segmentation; and (5) a GPU framework to achieve real-time feedbacks in a 3D environment.

1.5 Thesis Overview

The rest of the thesis is organized as follows. In Chapter 2, we provide the fundamental concept of medical imaging data and review key papers in the research of volume manipulation. In Chapter 3, we present our core technique—point radiation—which is fundamental in building all of our volume manipulation tools. In Chapter 4, we explain our suite of volume tools that can be used for interactive sculpting and cutting large volumetric data sets. In Chapter 5, we describe our sketch-based region growing algorithm as well as the set of VCR-like playback functions. In Chapter 6, we provide experiments using real-world medical volumes and discuss the limitations. Finally, we give conclusion and future work in Chapter 7.

Chapter 2

Background

Volumetric data consists of a discrete collection of samples, or voxels, at 3D locations. This can be expressed as:

$$f(i, j, k) = v \tag{2.1}$$

where v represents all possible values for the voxels and i, j, k are indices for the discrete 3D volume space. Voxels represent volumetric units and are analogous to pixels in a raster image. A voxel v may show properties, such as density, color, material or pressure. The property can also be a form of vector representing, for example, velocity at each location. The function f is usually defined on a regular grid or 3D array and it is isotropic if the samples are taken at regularly spaced intervals along the three orthogonal axes. When the spacing differs along the three axes, f becomes anisotropic. Three types of grid settings are possible: rectilinear, curvilinear (structured), and unstructured. Rectilinear grid has an axis-aligned structure with varying spacing constants along the main axes. Curvilinear grid is a transformed version of the rectilinear grid while preserving topology between voxels. Finally, the unstructured grid type requires the connectivity information to be explicitly specified (e.g. tetrahedra, hexahedra, or prisms) [31].

Volumetric data can be obtained from simulation or sampling. Simulation utilizes mathematical or modeling techniques such as computational fluid dynamics (CFD). The results of simulations are usually captured as volumetric data sets and used for

analysis and verification purposes. The sampling process is commonly carried out in medical diagnostic devices. In addition, industrial Computerized Axial Tomography (CAT) employs a similar sampling technology for inspecting composite materials or mechanical parts. The main distinction among the different sources of acquisition comes from the shape of the volume element. In medical imaging, the voxels are rectangular or cubic, but other applications may not have an identical setting; for example, the volume elements in CFD data are wedge shaped [56].

In this work, we focus on the sampled data acquired from medical imaging. However, our system also works with simulated and rectilinearly structured data. In the context of volumetric medical data, volume manipulation plays an important role for revealing hidden structures and obtaining regions of interest. Different approaches have been explored such as volume sculpting, volume clipping, and volume segmentation. For rendering of the manipulation results, object-order and image-order visualization strategies help to capture finer details and higher fidelity. Nevertheless, interactive volume techniques still remain as a challenging task as the size of datasets increases in a cubic order. In the following sections, we review a number of key papers for volume manipulation (Section 2.1) and outline the state-of-the-art rendering techniques (Section 2.2).

2.1 Volume Manipulation

Volume manipulation is a broad term and often appears in different contexts, such as volume modeling, volume sculpting, illustrative manipulation, volume segmentation, and volume deformation. In this work, we aim at a powerful interface by bringing

together a collection of volume sculpting and segmentation tools. Our goal is to provide tools to uncover hidden features, reveal the spatial relationship between different elements, and isolate the volume of interest. In general our system relates to the following volume manipulation categories: volume sculpting, volume clipping, volume sweeping, and sketch-based interfaces for volume segmentation. We review essential papers in each of the categories.

2.1.1 Volume Sculpting

The notion of volume sculpting originally refers to a modeling technique for sculpting a solid material with a tool, which modifies values in the voxel array. Sculpting tools are used to add, remove, paint, and smooth material. In 1991, Galyean and Hughes [22] present a modeling technique for sculpting a solid material with a tool. During the process of sculpting, they start with a block of material and remove it bit-by-bit. The shape of the sculpted object (clay) is described by a characteristic function; whose value is 1.0 where there is clay, and 0.0 elsewhere. To cut away the object, they propose three types of tools: Additive Tool (Toothpaste Tool), Heat Gun, and Sandpaper. The Additive Tool left a trail of material wherever it moved. The Heat Gun tool is used to melt away material like a heat gun melts styrofoam. The Sandpaper tool alters the object by wearing away the ridges and filling the valleys. In their work, the object (original solid) is directly represented by a data array, which they call a *voxmap*. The values in the data array ranges between 0 and 1. To render the solid, they extract the iso-surface of the object with the threshold defined at level 0.5. In addition, the data is low-pass-filtered to avoid aliasing effect. The sculpting tool is also represented by a *voxmap* and anti-aliased with a $2 \times 2 \times 2$ box filter.

While sculpting, however, the tool must align with the same axes of the object voxel data. To apply the tool to the object *voxmap*, the object's sub-voxel location is first determined; then a particular value from the tool *voxmap* is selected to combine with the object voxel value.

In 1995, Wang and Kaufman [55] present another modeling technique based on the metaphor of interactively sculpting complex 3D objects from a solid material. In their approach, the voxel data is represented as a 3D raster volume and reconstructed by a trilinear interpolation function. They propose two sculpting operations: carving and sawing. Carving is described as a process of taking a pre-existing tool to chip or chisel the object bits by bits. In their system, carving tools are pre-generated using a volume sampling technique and stored in a volume raster of $20 \times 20 \times 20$. The process of carving involves positioning the tool volume to the object in 3D space and performing a boolean subtraction between the two volumes. To subtract the object volume by the tool volume, their algorithm first finds the sub-volume of the object that is overlapped by the tool volume; then for each grid point within the object sub-volume, the boolean diff operator is performed. For the sawing operation, the user is allowed to draw any size circle, polygon, and Bezier curve to form a 2D sawing region. This region is then extruded (with a slider to adjust the depth) to form the tool volume on the fly. In order to prevent object space aliasing while generating the tool volume, they propose a 3D splatting method in substitution of the sampling process via expensive 3D convolution. In this method, the density of each point of the tool is splatted in 3D space to the affected neighboring sample points using a hypercone filter [55]. The hypercone filter has a spherical filter support with a maximum weight at the center and attenuated linearly to zero at a distance equal

to the radius of the filter. To accelerate the splatting process, the neighborhood sample points are pre-computed and stored in a 3D lookup table. In our approach, the density points are splatted in parallel on the GPU. Our method increases the interaction speed by a factor of 128 compared to serially processing data points on the CPU. In addition, our point-based design scales with the increasing number of stream processors on the GPU (e.g. ATI has reported 320 processors on their latest graphics card). Instead of pre-constructing 3D lookup tables, we minimize storage space and combine high-quality 1D and 2D Gaussian filters in coherent with the GPU pipeline.

In 1996, Avila and Sobierajski [3] introduce virtual tools that can be simulated by applying three-dimensional filters to some properties of the data within the extent of the tool. Similar to [55], they also represent the object volume as a 3D rectilinear array and use an interpolation function to produce a continuous scalar field. Each voxel in the object volume stores properties including material density, color, and an index value. The density and color are modifiable values, and the index value is used to access material stiffness, classification, and shading parameters. In the data modification (sculpting) process, a small tool volume is used to *melt* or *construct* the object volume. In addition to melting and constructing operations, they also experiment with a number of sculpting tools (e.g. burn, squirt, stamp, paint, and airbrush) by varying the constant value and data properties.

In 2000, Ferley et al. [21] present a sculpting metaphor for rapid shape prototyping. Sculpting tools are used to add, remove, paint, and smooth material in space. They allow the use of free-form tools that could be designed inside the application and propose a stamp tool to make imprints on an existing shape. They store discrete

potential field values in a 3D volume and adapt a balanced binary tree to update the data. For the sculpting tools, they directly use a continuous tool potential field without any filtering pass and store the discrete version in a volume. Free-form tool shapes are composed by duplicating the structure stored in the discrete volume. To apply a tool, a point sample is taken to reconstruct a continuous potential field by trilinearly interpolating the tool volume.

The majority of previous works ([22], [3], and [21]) have adapted 3D input device (e.g. 3D mouse) to control the sculpting tool. In particular, Avila and Sobierajski [3] incorporate 3D haptic input device into volume sculpting. However, sculpting with a 3D device can be a challenging task as parts of the volume can be occluding the tool itself. Moreover, it can be difficult to visualize the 3D location of the virtual tool relative to the target volume. Therefore, similar to Wang and Kaufman [55], we utilize 2D device for volume sculpting, In addition, we incorporate a sketch-based approach to provide more intuitive user interfaces.

2.1.2 Volume Clipping

Volume clipping provides a means to remove parts of the volume with cutting planes or more complicated geometry. Weiskopf et al. [57] propose interactive clipping techniques that exploit per-fragment operations on the graphics hardware. They combine data-driven transfer functions and geometry-guided clipping methods to effectively visualize a volumetric data set. With texture-based volume rendering, they allow interactive selection and exploration for the regions of interest. Two approaches are presented for volume clipping: a depth-based clipping technique and clipping via voxelized clip object. In the first approach, a clip object is represented

by a tessellated boundary surface, and its depth structure is stored in a 2D texture. This depth structure is then used by a fragment shader to clip away parts of the volume. Meanwhile, they introduce the terms: volume probing (i.e. where the volume is clipped away outside the clip geometry) and volume cutting, which is the opposite of volume probing (i.e. the volume inside the geometry becomes invisible). In the second approach, the clip geometry is voxelized and stored in a volumetric data set (3D texture). Clip regions are specified by marking the corresponding voxels in the volume. To avoid a common aliasing problem, each voxel stores the Euclidean distance (normalized to the interval [0, 1]) to the closest point on the clip object. During rendering, a trilinear interpolation is applied on the clipping texture with 0.5 as the visibility threshold.

Manssour et al. [36] describe a unified approach for clipping techniques in the context of the raycasting algorithm. Several tools are presented to explore different ways of incorporating seeing-through capabilities. The unified algorithm takes into account the individual samples examined by each emitted ray from the projection plane. When no clipping tools are specified, the ray enters and exits the volumetric data on the cubical boundaries. In this case, the sampling interval starts and ends on the data boundaries. In their work, they introduce a user-defined sampling interval in which any clipping/cutting tools can be specified by the interval. When a geometry constrain is used to specify the interval, clipping planes or clipping objects can be interactively configured. While using a confocal volume rendering technique, voxel properties can be combined with the geometry information provided by the user.

Recently, Huff et al. [28] exploit the graphics hardware and propose erasing, digging, and clipping operations to uncover hidden structures in the volume data.

The eraser tool is used to eliminate the voxels inside a virtual cylinder crossing the entire volume. The cylindrical region is calculated by using a projection plane, a 2D point on the plane, and the radius of the tool. The digger tool is developed to eliminate voxels inside a virtual sphere. The spherical region is generated by a 3D point and a pre-defined radius. The clipper tool defines a convex region inside the original 3D volume. An approach similar to the cutting planes is adapted to repeatedly carve and slice the current convex volume.

Some volume clipping approaches are inspired by traditional illustration techniques of cutaway views and peeling. Existing techniques perform automatic volume cutting by using importance functions associated with internal structures of interest [54], [10]. McGuffin et al. [37] present a method for browsing the volume with interactive manipulation widgets and assigning individual voxels as the fundamental primitive. More recently, Correa et al. [20] propose a set of procedural operators (peeler, retractors, pliers, dilators) that are placed anywhere on or within the volume. They define their manipulation operator as a transformation mapping from the original volumetric space to a dissected or deformed volumetric space. However, these operators are needed to be defined procedurally.

2.1.3 Volume Sweeping

Swept volume is defined as the volume generated by the motion of an arbitrary object along an arbitrary path (or even a surface) possibly with arbitrary rotations [1]. A difficult problem in swept volume has been the identification and visualization for the volumetric boundary of complex geometries. Despite the great challenges, the study in this field results in numerous applications, such as numerically controlled

machining verification, robot analysis, solid modeling, ergonomics, collision detection, and pen-stroke modeling [1]. In the context of volume manipulation, swept volume can be utilized as a tool for removing regions of the volume.

In order to construct the swept volume, the sweeping definition needs to be transformed into a volume data. Winter and Chen [59] exploit image templates for constructing swept volumes. In their work, images are loaded from files and swept along a pre-defined trajectory. Their underlying data type is scalar field. Two approaches are adopted for composing the resulting volume: voxelized approximations and original sweeping specification. In the first approach, the sweeping definition is voxelized and converted to a volumetric data set by walking along the trajectory in discrete intervals. The trajectory is subdivided into small segments with a distance criteria. For each segment, two neighboring image templates are located; and for each voxel bounded by the two templates, it is projected onto each of the templates along the sweeping direction. Each template is sampled at the projection point, and the two sampled values are linearly interpolated to yield the voxel value [59]. However, this process requires a lengthy amount of time to generate the entire swept volume, and hence interactivity could not be achieved. In the second approach, the swept volume is directly ray-traced with simple sweeping specifications such as rotational sweep and Bezier trajectory sweep. During the rendering process, sweeping equations are directly evaluated for each sample on the casted ray. The evaluation process involves algebraically or numerically computing the inverse mapping for the sweeping function. Hence, the specification of the sweeping trajectory could only be procedurally defined. Both of these approaches are capable of producing smooth swept-volumes. However, an intuitive interface for guiding the volume sweeping is

not presented. More importantly, their technique would not suitable for real-time environment as it may take minutes to generate a swept volume that could not be dynamically modified.

2.1.4 Sketch-based Interfaces for Volume Segmentation

Many segmentation approaches have been proposed for 2D image segmentation including thresholding, k-means clustering, watershed segmentation, and level-set methods (see the survey conducted by Pham et. al. [44]). For segmenting 3D medical data sets, these techniques could also be applied and adapted easily by re-using the 2D image techniques. While it is straight-forward to interact with 2D images, it is a difficult task to develop interfaces to guide 3D segmentation. In the effort of providing interfaces that can quickly translate 2D instructions to 3D guidance, we focus on using sketch-based interfaces for volume segmentation within a 3D environment.

To specify region of interest (ROI) in the volume using sketches, a number of works focus on painting over cross-sectional images of the volume data. In 2003, Sherbondy et al. [50] present a seeded region growing segmentation method using the parallel computational power of commodity graphics hardware. In their seed selection stage, the user is allowed to paint seeds by drawing on the cross-sectional views of the volume (i.e. axial and coronal image sections). The brush is defined as a sphere with adjustable radius. Tzeng et al. [53] propose an intuitive user interface for specifying high-dimensional classification functions by painting directly on sample slices of the volume. Besides the intensity and gradient information that are used in conventional classification, they consider additional properties such as textures and positions. In their approach, the user is allowed to paint on a few

volume slices to partition the data into different material classes. Painting in one color specifies the example regions that belong to the material class of interest and another color defines the regions that are outside. Not all slices of the volume have to be painted, but the user must paint in some areas of a couple of slices to classify the volumetric data set. After each painting, their system generates a high-dimensional classification function by training an artificial neural network. The neural network takes the samples (painted regions) as input to obtain a set of weights. A classifier is used to map voxels into uncertainty. As a result, the classified volume is passed to the renderer for immediate visual feedback. The high-dimensional function takes a number of voxel properties as input. These properties include the voxel's scalar value (fundamental information), gradient magnitude, neighboring values (provide information that can be incorporated for texture), and position (i.e. taken into account a material's structural properties). However, their approach has limited flexibility in which the user can only paint on cross-sectional slices.

More recently, new techniques have emerged that allow users to sketch directly over a 3D displayed volume for specifying ROI. This brings forward an era of more intuitive and interactive volume segmentation. Owada et al. [41] develop a system called volume catcher to segment volumetric data and obtain ROI. They propose an intuitive user interface in such a way that the user traces the contour of the target region by drawing a 2D free form stroke over the displayed volume. Assuming the user has traced the silhouette of the ROI, their system will instantly return a plausible 3D region inside the stroke after running a statistical region merging algorithm. Their main contribution is that the system inferred the depth information of the ROI automatically by analyzing the data whereas conventional methods mostly rely

on the user to specify this information. In their system, a single stroke (open or closed) is used to indicate foreground and background constraints depending on the orientation of the drawing. The algorithm that deduces a 2D freeform stroke to a 3D path first sweeps the 2D freeform stroke along the depth direction to create a 3D curved surface. This sweep surface is then parameterized to produce a set of sampling points. On each lattice point, a silhouette coefficient is computed to indicate the closeness of a point to a silhouette from the current viewpoint. An optimal path on the parameter space is then converted to a 3D path by assembling the set of 3D lattice points [41]. Finally, the constraints are generated by offsetting the 3D path with a direction and displacement. Despite the straight-forward interface they have adapted, their method does not work for all data sets. In our approach, segmentation seeds are grown from the visible surface points sketched by the user and can work for both connected and disconnected data sets.

In our previous work [13], we have developed a sketch-based interface for seeded region growing volume segmentation. In that approach, the user can rotate the data in any angle and freely sketches the ROI directly over the 3D volume. The strokes are captured in its original form for maintaining precision while projecting into the volume space. After the strokes have been laid, parts of the volume outside the ROIs are then automatically cut out in real-time. The user repeats this process as many times as necessary until he/she decides to specify the seed point 3D location directly at the ROI. To prevent unexpected segmentations, the region growing is restricted to the specified ROI. The system utilizes GPU programming to achieve real-time processing for both rendering and volumetric cutting independent form the size and shape of the sketched strokes. Figure 2.1 shows an outline of the segmentation

process. The results in our previous work are depicted in Figure 2.2. In the current approach, we develop real-time tools in the context of ray-casting to provide higher rendering fidelity. This allows the entire volume to be manipulated and rendered interactively instead of caching only a small subset of voxels on the GPU. Moreover, the user can sketch seeds on multiple surfaces simultaneously to join the disconnected regions and gain flexibility in variety of data sets.

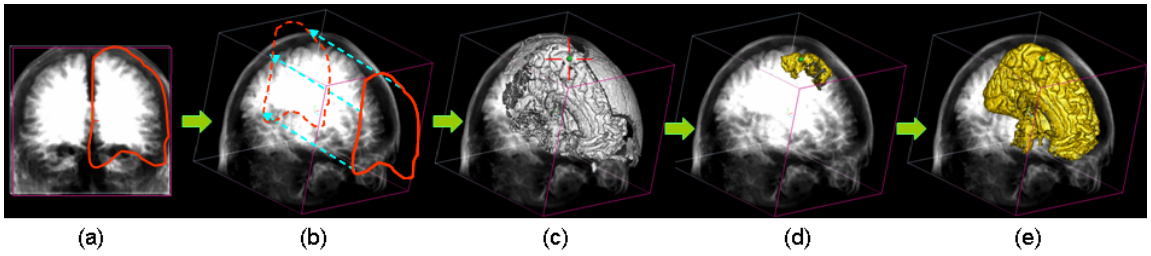

Figure 2.1: Our previous sketch-based volume segmentation method: user sketches a ROI directly over the data (a), the ROI is extruded (b), volume outside is cut out and user plants the seed point (c), region grows (d) and segments volume portions within the extruded ROI (e). [13]

2.2 Volume Rendering

There exists a variety of techniques for rendering a volumetric data set. However, two main approaches have emerged and become the mainstream of research in recent years. One direction is along the lines of point-based rendering. Although we do not adapt point-based rendering directly in our system, our GPU-based point radiation technique (described in Chapter 3) is solely founded by the conceptualization of 2D splatting. The other direction is raycasting, which we use as a core rendering engine in our system. In this section, we first provide an overview of the popular techniques in volume rendering. And then we review papers and describe, in more details, the

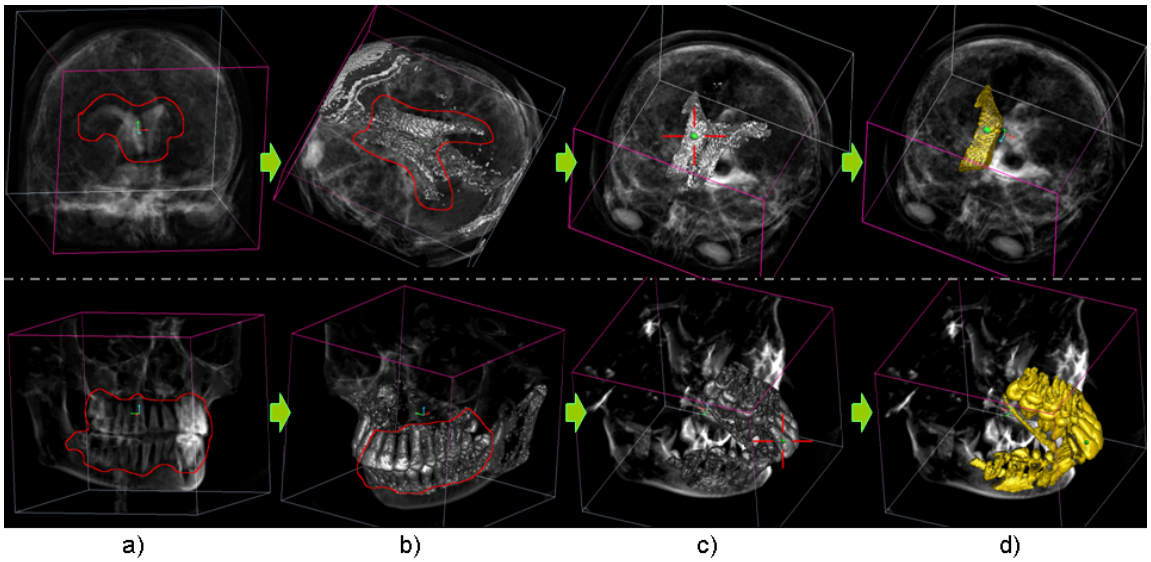

Figure 2.2: Our previous results showing segmentation of right ventricle (top) and partial teeth (bottom). (a) Raw volume, X-ray, with sketched-region. (b) Resulting cut, rotating the view, new sketch. (c) Resulting cut, rotating the view, plating the seed. (d) Region growing contained within sketched/resulting volume from (c). [13]

two fundamentally different approaches: point-based rendering and raycasting.

2.2.1 Overview

The standard graphics pipeline has been well established and specializes in rendering geometric entities such as triangles. A number of early attempts to render volumetric data are to approximate a surface using geometric primitives. A famous method for this process would be the marching cubes algorithm [35]. In this algorithm, an iso-value is defined to distinguish between inside and outside of the volume. A surface is constructed by marching along the boundary cubes of the data volume and collecting intersecting triangles based on a given iso-value. The resulting surface is also called an iso-surface since all surface elements are constructed around this iso-value. There are 256 ways of how a surface can pass through a grid cell, and they are further broken

down into 15 base topologies. For each of the 15 cases, a look-up table is used to store the set of generic triangles, and the actual triangle vertex locations are determined by using linear interpolation [31]. An approach to accelerate the marching cubes process using programmable hardware on commodity GPUs is presented by Johansson and Carr [29]. They pre-compute topology for each cell and store the information in a so-called—span space—data structure on the GPU. However, this technique suffers from several limitations when visualizing, for example, a CT scanned data. During the conversion process, the context and the solidity aspect of the volume are lost, and there may be an excessive quantity of geometric primitives produced.

Recent approaches focus on directly visualizing the volumetric data without generating an intermediate surface representation. This process is also referred to as *direct volume rendering*. In direct volume rendering, the data can be visualized with object-order or image-order techniques.

In object-order volume rendering, a forward mapping approach is used to map the volume data onto the image plane. This approach decomposes the volume into a set of basis functions, or kernels, that are individually projected to the screen and blended into the final image. This process is also called *splatting* as the idea is analogous to throwing eggs onto a wall [58]. The eggs can be seen as the basis functions, with the size of the egg relating to the width of the kernel. Splatting is categorized into three types: composite-only, axis-aligned sheet-buffered, and image-aligned sheet-buffered splatting [31]. Composite-only splatting accumulates footprint on the screen. Axis-aligned sheet-buffered splatting sums up the entire kernel within the current sheet, whereas the image-aligned version adds only slices of the kernels intersected by the current sheet layer. Another splatting approach called Elliptical

Weighted Average (EWA) allows screen-space shapes other than circles to be defined.

In image-order volume rendering, a backward mapping approach is employed and usually referred as raycasting ([58], [30]). In our system, we use raycasting as a primary rendering method as it provides higher quality results over object-order techniques. Raycasting starts by emitting rays from pixels on the image plane. Intervals are taken along each ray to sample the values within the volumetric data set. Sampling is commonly performed by trilinear interpolation, which is supported by most graphics hardware today. At last, the samples are combined to form the final pixel values. Different rendering modes can be selected to combine these sampled values, and they include: X-ray rendering, Maximum Intensity Projection (MIP), and iso-surface rendering. The X-ray mode (Figure 2.3, a) sums up the sampled values along each instanced ray. The MIP rendering mode (Figure 2.3, b) records the sample that has the largest intensity (or attribute) value. Iso-surface rendering (Figure 2.3, c) uses an iso-value to determine the surface points and applies standard shading techniques (e.g. Phong shading) to color the pixel values. Surface gradients are required to model the reflection of lights. To transform voxel properties to colors, transfer functions can be applied to map density or gradient magnitude to color values. In addition, transfer functions can also be configured to show multiple nested surfaces. To achieve this, front surfaces are rendered as semi-transparent layers and the result is blended with the colors behind.

One important advantage of object-order volume rendering is that only the required points have to stored, dramatically eliminating the empty space storage overhead for large data sets. However, the drawbacks of these methods include longer rasterization time and expensive blending operations when compared to the image-

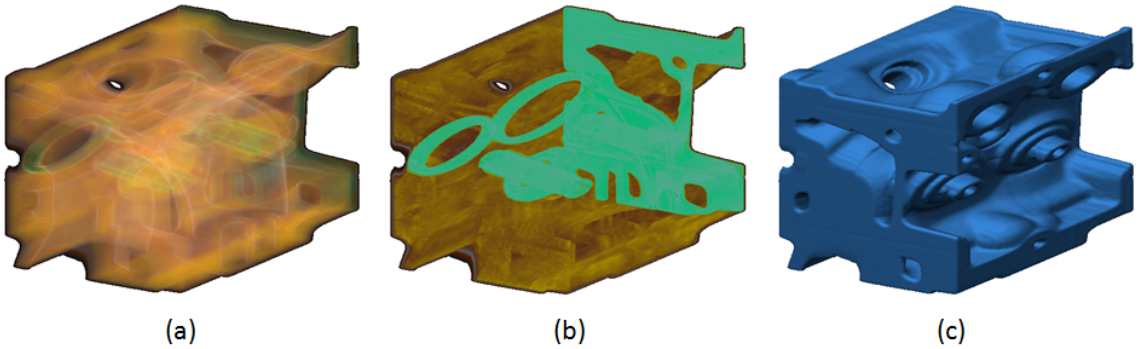

Figure 2.3: Engine block rendered with different image-order volume rendering modes. (a) X-ray rendering. (b) Maximum intensity projection (MIP). (c) Iso–surface rendering.

order techniques. Other methods such as domain-based or hybrid techniques have also been explored. Domain-based methods [31] convert the spatial volume data into other domains, such as compression, frequency (e.g. Fourier domain), and wavelet, and project the transformed data onto the screen directly. Hybrid strategies seek to combine the object-order and image-order techniques to find the optimal data representation and rendering quality.

For volumes produced by medical imaging devices, the data typically contains a density-scaled grid of values. When rendered with the standard techniques described above, the result is a gray-scaled image. To enhance the image and allow distinction between the various materials, transfer functions which map volume density values to different colors and transparencies can be utilized. The mapping can be achieved by analyzing a number of volume histograms: density, gradient magnitude, and two-dimensional histogram [34]. Figure 2.4 illustrates the use of a one-dimensional density histogram that accumulates the quantity of voxels (vertical axis) for each density value (horizontal axis). A fuzzy classification function can be applied to

interpret intervals of density values as material properties, such as air, fat, soft tissue, and bone. Each material property can then be used to assign opacity and color values to the voxels in the volume. Kniss et al. [34] exploit transfer functions in higher dimensions introducing both 1D and 2D transfer functions. For 2D transfer functions, edges or darker areas in the 2D histogram reveal material boundaries.

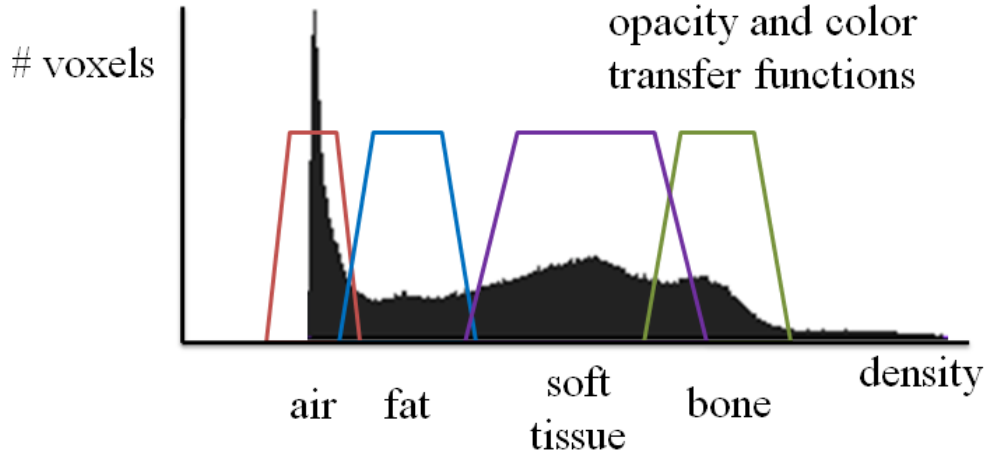

Figure 2.4: Density histogram with vertical axis indicating the number of voxels and the horizontal axis indicating the density values. The peaks in the histogram represent the different material properties as air, fat, soft tissue, and bone respectively. Fuzzy classification is utilized to map ranges of density values into colors and opacities. [31]

2.2.2 Point-Based Rendering

Botsch and Kobbelt [8] propose a point-based rendering framework on modern GPUs. Their approach is based on a two-pass splatting technique with a smoothing filter. They use programmable graphics hardware to render high quality splats while minimizing data transfer between CPU and GPU. They are able to achieve up to 28M surface splats per second on a GeForceFX 5800 graphics card. In their work, they

propose to use splats as rendering primitives. In the first rendering step, splat size and shape have to be determined in order to reconstruct a continuous surface without holes. The splat size is calculated based on the current viewing parameters and the splat's position and radius. The shape of the splat is defined by a small disk in object-space. To render the splat in an efficient way, they utilize the hardware accelerated function—point sprites. Splats are also filtered and blended to achieve high-quality anti-aliased rendering. The filtering process is governed by a radially decreasing Gaussian weight function. The weight of each pixel in the image-space is computed by a function of the 3D distance of its corresponding object-space point to the splat's center. This function is implemented with a 1D Gaussian texture look-up directly performed in the fragment shader. For blending the result, a per-pixel normalization is used to average the contributions of overlapping splats.

2.2.3 Raycasting on GPU

Scharsach [48] present advanced raycasting techniques using the graphics hardware. He leverages the fragment shader of a shader model 3 compatible graphics card and enables orthogonal projection, perspective projection, and fly-through applications for volumetric data. He also introduces a cached blocking scheme to bypass the scarce GPU memory problem. In his hardware raycasting implementation, the volumetric dataset is loaded into the GPU as a three-dimensional texture with the support of hardware trilinear interpolation for sampling. After loading the data, the volume is represented and rendered by a simple bounding box. The position inside the volumetric data set is encoded as the color channel and stored as the texture coordinates of the boundary box.

To perform the actual ray-casting algorithm, Scharsach outlined the procedure in the following steps [48]:

1. Draw front faces of the color cube into an intermediate texture.

2. Draw back faces, subtract the color value of the front faces, normalize the outcome and store this vector together with its initial length in a separate direction texture.

3. Draw front faces again, taking the colors as an input parameter for the fragment program, and cast along the viewing vector (that is stored in the direction texture). Store the intermediate steps in a separate compositing texture. Terminate the ray if we leave the bounding box or as soon as the opacity has reached a certain threshold (early ray termination).

4. Blend the result back to screen. It would be possible to composite to the screen directly, but a separate blending step makes the approach more flexible and does not impose a significant speed penalty.

To accelerate the rendering process, he also incorporates empty space skipping into the raycasting algorithm. With empty space skipping, a blocking scheme is used with equally sized blocks representing collections of voxels from the original data set. Each block is labeled as active if it passes the iso-value test of a corresponding transfer function, and inactive otherwise. This way, unnecessary examination of voxels can be quickly bypassed. To remove artifacts due to insufficient sampling rates, he proposes a hit-point refinement method to better estimate the real intersection point with the iso-surface. The basic idea is to recursively step back or forward (i.e. multiply the

step size by -0.5 or 0.5) whenever the density is greater or smaller than the pre-defined threshold. In every refinement process, six bisection steps are performed in order to obtain a close approximation.

In addition to the common hardware support for trilinear sampling in raycasting, Sigg and Hadwiger [52] describe a fast GPU implementation to perform third-order texture filtering. They enable filtering with a cubic B-spline kernel and successfully reconstruct continuous first-order and second-order derivatives based on the same method. In their approach, the number of input samples required for cubic filtering is dramatically reduced by utilizing linear texture fetches instead. As a result, they are able to evaluate a tri-cubic filter with 64 summands using just eight trilinear texture fetches.

Chapter 3

GPU-Based Point Radiation

In this work, we develop a set of real-time volume manipulation tools based on a fundamental concept—point radiation. We enable interactive processing speed by harnessing the parallel computational power in commodity graphics hardware. Both the manipulation and rendering are taking advantage of the programmable graphics hardware (see Appendix B). In this chapter, we describe the GPU-based point radiation technique in detail and provide a proof-of-concept to support our arguments. In Section 3.1, we first introduce the volume sculpting problem that we are trying to solve. In Section 3.2, we discuss our previous approach and related work in swept volume. In Section 3.3, we provide an overview of our suite of solutions as well as the motivation behind. In Section 3.4, we explain our fundamental sculpting primitive that is based on a point radiation method implemented in the graphics hardware. In Section 3.5, we extend the point radiation technique to work with user-defined masks. Finally, we show results and discuss related issues in Section 3.6.

3.1 The Volume Sculpting Problem

Volumetric data acquired from medical diagnostic systems often involve complex structures and a mixed of different materials, such as bones, flesh, veins, arteries, soft tissues, fluids, and air. Visualization methods and classification techniques have been

developed to exhibit the general information embedded within the data sets. Most approaches only apply global operations on the data but do not take consideration for local point of interest. However, in professional radiology practice, clinicians, surgeons, and neuroradiologists frequently explore the inner structures of the sampled data and quantify isolated region of interest within the volume. The results are then used for diagnosing cancerous cells, measuring tumor sizes, and assessing carotid artery stenosis.

Volume sculpting is an interactive approach that aims to isolate individual components within the volume and to locate region of interest. It is a process to carve away bits of a solid material. Volume sculpting presents an important modeling technique in computer graphics. Similar to a 2D painting program, the user employs an eraser to remove paint from the canvas. In the case of 3D, a virtual tool can be adapted to gradually cut or clip away pre-existing parts of the volume. Volume sculpting targets in the voxel level and removes any occluding region that the user has no interests in while retaining only important sections of the material. Volume sculpting is an essential operation to uncover hidden details and reveal inner structures of a data set. When combined with segmentation and data driven transfer functions, it becomes an unprecedented tool which unrolls many possibilities for volume manipulation and volume visualization.

Interactive volume sculpting is constrained due to the following reasons. First, medical volumes are large (i.e. anywhere from 256^3 to 512^3 and increasing) and demand a high performance computational equipment and algorithms to be merely effective. Second, manipulation on such data sets sometimes requires multiple times the storage space of the original data. Clearly, the limitations on processing power

and storage availability are restricting the development of interactive tools for volume sculpting. In addition, the lack of intuitive tools for defining volume of interest also prevents users from achieving desirable operations.

Over the last decade or so, different algorithms have been proposed to solve the volume sculpting problem. A number of approaches utilize 3D interaction devices to control the sculpting tool, such as [22], [3], [21], and [12]. Sculpting a volume with 3D devices seems to be a direct approach and a straightforward metaphor for operating on 3D data sets. But with a 2D display (as what most workstations are equipped with), the user is always presented with a 2D image at any moment in time regardless of the rendering methods selected (i.e. marching cubes with standard pipeline, raycasting, or 3D texturing). Therefore in our system, we resort to a two-dimensional interaction scheme incorporating a pen and a tablet. In addition, we use advantages of the latest graphics hardware capability to achieve real-time volume sculpting operations.

3.2 Related Work

In our previous work [13], we used a closed free-form stroke to depict a cutting tool that crops out the extruded volume of the stroke's region along main axes with an orthographic projection using the current view. To achieve real-time processing, we adapted a point caching mechanism, with the visible portion of the volume represented as points and stored in the graphics memory. Various index buffers were used to assist with the cutting operations. The result generated by the cutting tool was constrained to a forward cutting scheme which was necessary to provide a direct con-

nection from 2D free-form sketch to 3D volumes. However, this approach would not be suitable for problems involving drilling or cutting on a targeted area of interest (e.g. removing or opening the skull without affecting surrounding materials). Therefore, an extension for building flexible cutting paths over the forward-only method is desired. In this work, we extend our previous approach in [13] to build an interactive sculpting framework that is flexible, extensible, scalable, fast, and simple to compute. Instead of caching the volume as points, we store the entire volumetric grid in the graphics memory and handle all manipulation requirements directly on the GPU. We propose a point-based approach and a novel masking mechanism. We combine the benefits of swept volume [1] as it has volume clipping potential allowing flexible cutting paths.

The definition of swept volume is to move an arbitrary object along an arbitrary path [1]. Winter and Chen [59] presented two approaches for constructing swept volumes using image templates. In the first approach, they employed a volume raycasting technique for rendering swept volumes in their original sweeping specifications. In the second approach, they approximated the swept volume by voxelizing the set of image templates swept along a pre-defined trajectory. To build the swept volume, they adapted a projection-based approach that traverses every voxel between a pair of templates. However, this becomes a complicated process in the situation when two templates intersect. Two other limitations were also observed. First, the swept volume approximation depended on the resolution of the volume and the threshold of the subdivision path. Second, the processing speed depended on the complexity of the trajectory and the size of the volumetric data set. In their example, a complete model was generated as long as two minutes for a volumetric data that had a

resolution of $150 \times 150 \times 300$.

3.3 Our Approach

In our work, we apply a similar swept volume concept in volume sculpting. We sculpt or clip a volumetric data set by erasing the portions swept by a sketched mask. We adapt a point radiation method (Figure 3.1) as the granularity of removing voxels with a mask. A radially decreasing function is attached to the point for distributing the field of influence to its surrounding voxels.

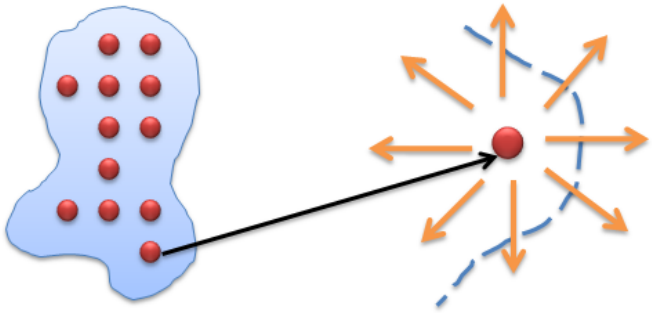

Figure 3.1: The point radiation technique. Left: elements on a sketched mask. Right: point radiation for a particular element on the mask.

Our process of constructing the volume (swept by a user-defined mask) is identical to using image templates as in [59]. But instead of using a projection-based approach, we resort to a method similar to the 3D splatting process described by Wang and Kaufman [55] because of its simplicity and extensibility. Rather than pre-computing the neighborhood distributions of a splatting point as in [55], we adapt a GPU programming paradigm to achieve precise calculation of neighborhood distributions and obtain real-time anti-aliased results.

The GPU-based point radiation strategy is the foundation of our system. Using a mask-swept volume approach further confirms this claim (the point-based sculpting primitive is both effective and flexible). In mask-swept volume, a user-defined mask is drawn and swept along a trajectory. The resulting swept volume is modeled and computed by adapting the point radiation method (shown in red dots). Then we extend the same concept, that uses a set of flexible points as sculpting primitives, and introduce different trajectories to build the drilling tool, the laser tool, and the peeling tool (described in Chapter 4). We also build a sketch-based segmentation tool based on the point radiation method (described in Chapter 5). In the following sections, we present the GPU-based point radiation algorithm and the masking process in more details.

3.4 GPU-Based Point Radiation

In our approach, the concept of the point radiation method is a fundamental building block. This technique enables a number of applications in volume manipulation due to its simple design, flexibility, extensibility, scalability, and parallel computability via the programmable graphics hardware. The term *point radiation* extends naturally from a hardware supported function—point sprite. Point sprite is a two-dimensional billboard method for rendering a textured image from a single point in space; whereas point radiation establishes a three-dimensional metaphor for rendering a filtered volume from a point sample. The basic idea of point radiation is simple and borrows from the notion of 2D splatting.

3.4.1 Concept

2D splatting was first described by Westover in [58] as a footprint evaluation process for object-order volume rendering (more details are found in Chapter 2). In splatting, voxels are projected onto the image plane with a filtering operation. Figure 3.2 shows two strategies for filtering voxels. The first method (Figure 3.2, left) samples in the

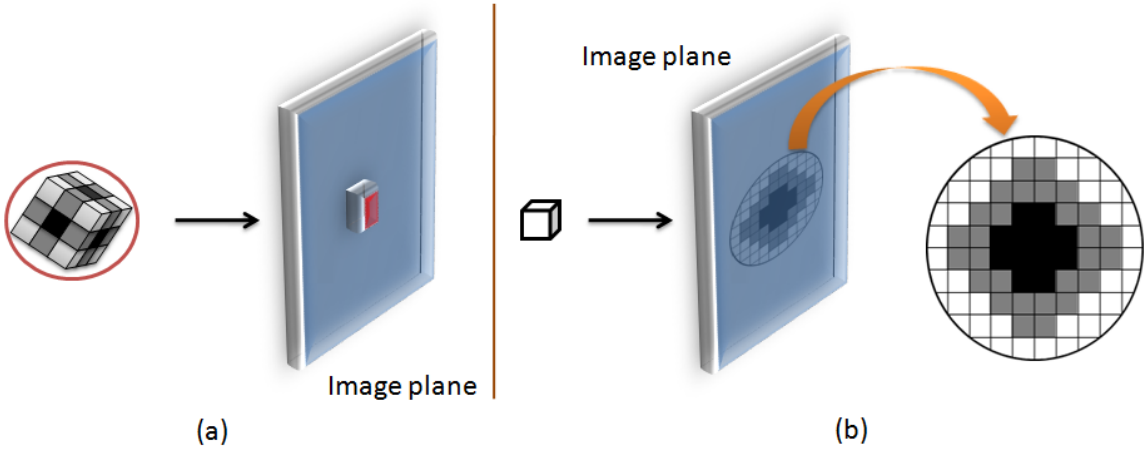

Figure 3.2: (a) shows that one voxel contributes values to many pixels in a weighted footprint. (b) shows many voxels are filtered over a spherical region to provide a single value for a pixel. [56]

volume space and computes a pixel value on the image plane. To determine the pixel value, a reconstruction filter is used to gather contributions from the surrounding sampling voxels. The second scheme (Figure 3.2, right), first proposed by Westover [58], projects each voxel (sample) onto the image plane and uses a convolution filter to distribute energy (intensity values) from the input sample. A footprint in the image space is computed and covers the affected pixels on the image plane. This scheme allows every input sample to be treated individually and is therefore suitable for parallel execution.

To select a convolution kernel for splatting, a number of recent works (e.g. [8], [7], [43], [13], [9], [40], [39], [25], [62], and [14]) have chosen the Gaussian distribution function. The Gaussian kernel smoothes neighborhood elements and provides a high-quality anti-aliased rendition. Figure 3.3 show a discrete approximation to Gaussian function as a weighted average mask. During the kernel composition step, the samples are splatted onto the image plane and combined by using a simple summation operation. The final image results in a smooth blending of a collection of contributions from the input samples.

$$\frac{1}{115}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Figure 3.3: Discrete approximation to Gaussian function with $\sigma = 1.4$ [23].

Similar to 2D splatting, our point radiation method is also an energy distribution process but produces a three-dimensional footprint. This is also analogous to the process commonly found as in the context of implicit volume modeling ([4], [5], and [49]). Implicit modeling defines an implicit volume using a continuous scalar function with an iso-value. The implicit volume is bounded by the iso-surface formed by setting the scalar function to the iso-value. Multiple implicit volumes can be combined by defining different scalar functions and blended with compositional operations (e.g union, addition, or difference). Furthermore, complicated primitive volumes can be

pre-defined and blended with a tree-like structure [61]. In our approach, we do not rely on solving complicated implicit functions and sorting hierarchical blending operators. We only need point primitives as it provides a simple and flexible interface for interacting with sampled volumes. Most importantly, we implement the point-based strategy on the programmable hardware to achieve parallel performance gain (e.g. a minimum factor of 128, as determined by the number of stream processors on the GPU) compared to the time required for solving and ordering implicit equations on the CPU.

Our input can by any point sample $\mathbf{p}$ in the continuous 3D space with position $(x, y, z)\ \varepsilon\ [-0.5, 0.5]^3$ that falls within a normalized unit cube. We use Gaussian as the filtering kernel for spreading energy radially into the volume space. The extent of the kernel filter is defined by a radius R, in terms of number of voxels. The $2R \times 2R \times 2R$ region forms a 3D footprint in the volume space. Figure 3.4 shows an example of a 3D kernel filter with radius 2 and its corresponding 3D footprint.

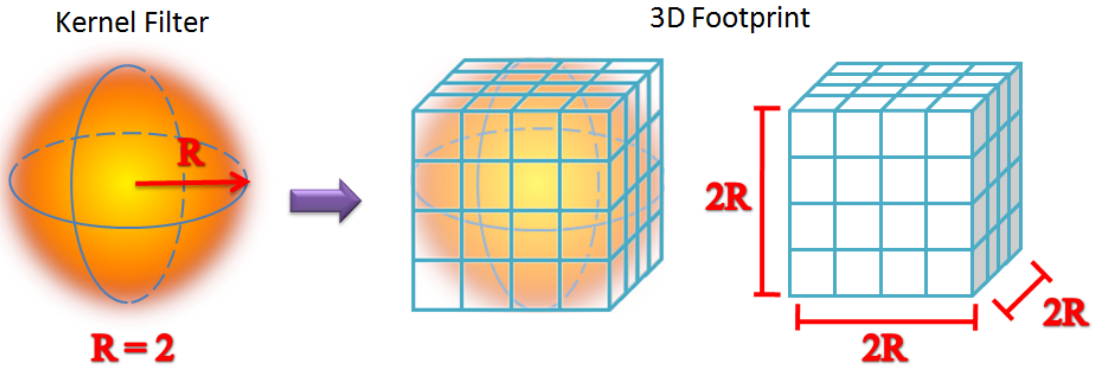

Figure 3.4: The 3D kernel filter with radius 2 (left) and a 3D footprint in volume space (right) that expands a 4 by 4 by 4 region.

To compute the weight ω at a particular voxel $\mathbf{V}$ of the footprint, we map $\mathbf{V}$ to

$\mathbf{v}$ the Gaussian kernel space using viewport transformation. Then, we only need to use 3D Gaussian function:

$$\omega = Gaussian3D(\mathbf{v}).$$

To compute the energy released by the set of footprint voxels, we combine the input sample value and the filter weight as:

$$\alpha_i = s \cdot w_i, \tag{3.1}$$

where α_i is the output energy at i^{th} footprint voxel; s holds the input sample value; and w_i is the weight at i^{th} footprint voxel. Figure 3.5 shows this smoothing scheme as well as its 1D and 2D analogy for combining the energy contributions from input samples. The 3D kernel composition step (Figure 3.5, right) is similar to the blending operation as in implicit modeling. In this work, we utilize a simple summation to accumulate the energy contributions from all input points.

3.4.2 Methodology

The challenges associated with the point radiation concept include the followings: (1) the ability to quickly generate 3D footprints and blend with a separate volume, which is used to hold the radiation result; (2) the computation should be fast; (3) the system should stay interactive even with a large number of input points; and (4) the rendering speed should not be compromised during intense user interaction. A CPU-based implementation would not satisfy the set of requirements because of serial executions. Thanks to the recent advancement in the programmable graphics hardware, dramatic increase in processing parallelism has become available to accelerate and optimize point processing. The addition of the geometry shader and a

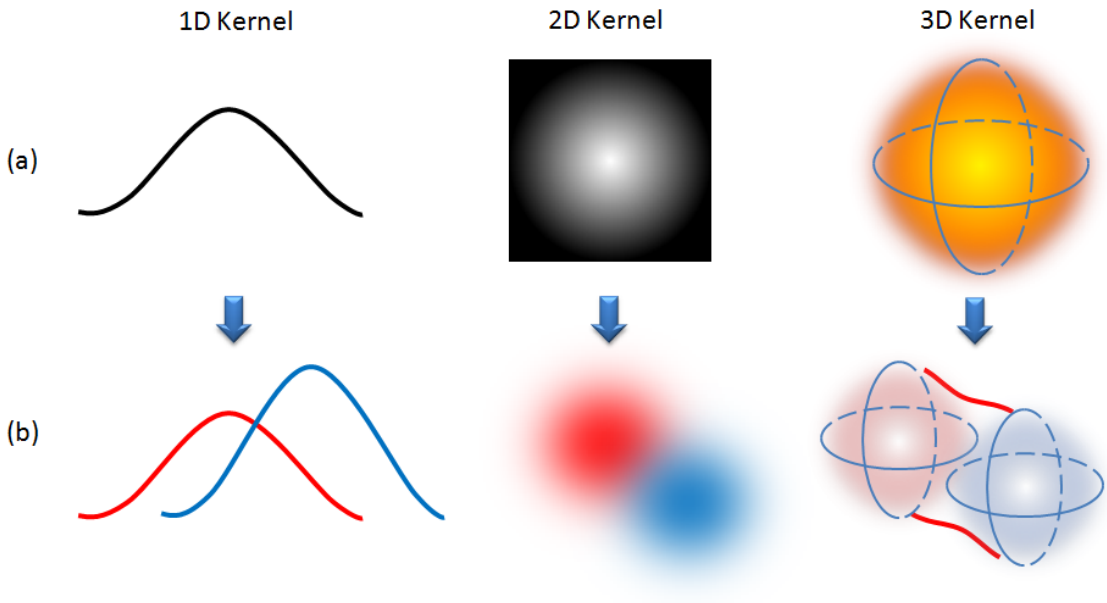

Figure 3.5: (a) shows the filtering kernels in its 1D, 2D and 3D form. (b) shows the blending process for the corresponding filtering kernels in (a).

unified architecture also enables the aforementioned concept of point radiation to be fully realized.

We implement the point radiation technique utilizing the geometry shader and its ability to render to 3D texture (see Appendix B). Geometry shader allows point primitives to be amplified (e.g. dynamically constructing primitives during the pipeline processing) and can redirect the input point to any location in the 3D output texture. One possibility in the point radiation algorithm is to use these features as we need to create a 3D footprint from the input point and blend it with the 3D output volume. However, if we directly instruct the geometry shader to generate the set of all footprint points, the geometry shader may become overloaded. Consequently, this prevents stream processors on the GPU to be executed for other

tasks and possibly impairs parallelism. For example, when the kernel has a radius of 4, the geometry shader is required to output 512 point primitives for constructing a complete 3D footprint, and this is against the design rule of building compact shaders on the GPU [42]. Moreover, the geometry shader can only emit up to 1024 vertices [18]. Kernels with radius greater than 5 produce more than 1728 points and therefore are no longer supported by the GPU.

Instead, we utilize the point sprite hardware acceleration function to assist with this task. Point sprite is commonly used in most well-known splatting techniques to achieve real-time rendering. It allows a point to be rasterized into a square region consisting of fragments containing relative coordinates with respect to the input point (e.g. fragment at the lower-right corner has coordinate $(1, 1)$, refer to hardware manual for exact specification). To reconstruct the set of 3D footprint voxels, we use a stack of 2D point sprites to largely reduce loads on the geometry shader. We call this 3D point sprite as this can be entirely accomplished by a single geometry program. Figure 3.6 illustrates our 3D point sprite technique. For the previous example, now the geometry shader only needs to output 8 points (instead of 512 points) with each point configured as a 2D point sprite.

To compute the energy α_i at a footprint voxel, we need to sample the Gaussian kernel filter. A 1D Gaussian function is a function of the form:

$$f(x) = ae^{-\frac{(x-x_O)^2}{2b^2}}, \tag{3.2}$$

for some real constants $a > 0$, b, and x_o (center); and a 3D Gaussian function is:

$$f(x, y, z) = a^3 e^{-\frac{(x-x_O)^2+(y-y_O)^2+(z-z_O)^2}{2b^2}}. \tag{3.3}$$

Theoretically, we could sample the 3D Gaussian function by using a 3D coordinate

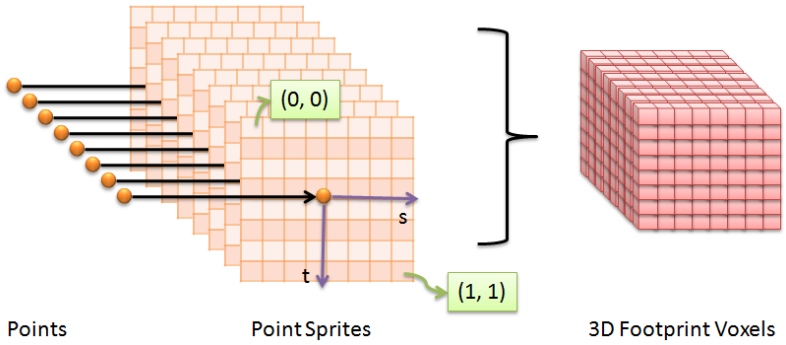

Figure 3.6: The 3D point sprite. Points are emitted as 2D point sprites by a single geometry program and reconstructed into 3D footprint voxels. Each 2D point sprite contains automatically generated texture coordinates ranging from (0, 0) to (1, 1).

that consists of the 2D coordinate generated by 2D point sprite and its corresponding depth value. But this requires us to store a 3D texture in the GPU memory to represent the Gaussian kernel. When composing a 3D Gaussian texture in high resolution, a large amount of memory has to be allocated for storing the 3D texture. In fact, the 3D Gaussian kernel can be decomposed into a 2D Gaussian function in the XY-plane and a 1D Gaussian function in the Z-axis. A 2D Gaussian function is of the form:

$$f(x,y) = a^2 e^{-\frac{(x-x_o)^2+(y-y_o)^2}{2b^2}}. \tag{3.4}$$

If we multiply the 2D and 1D Gaussian function, we have:

$$\begin{aligned} f(x,y)\cdot f(z) &= a^2 e^{-\frac{(x-x_o)^2+(y-y_o)^2}{2c^2}} \cdot a e^{-\frac{(z-z_o)^2}{2b^2}}, \\ f(x,y)\cdot f(z) &= a^3 e^{-\frac{(x-x_o)^2+(y-y_o)^2+(z-z_o)^2}{2b^2}}, \\ f(x,y)\cdot f(z) &= f(x,y,z). \end{aligned}$$

Figure 3.7 demonstrates the multiplication of the two Gaussian functions for reconstructing the 3D Gaussian filter which has a spherical support. This scheme reduces

memory consumption since we only need to store a 2D texture and a 1D texture on the GPU. The coordinates generated by the 2D point sprite can now be used in its original form to look up the 2D Gaussian texture without an extra coordinate reconstruction step. To further enhance our algorithm, we perform the 1D Gaussian texture lookup in the geometry shader and attach it to the output 2D point sprite. This allows a single texture lookup to be used for all fragments within the same 2D point sprite.

Figure 3.7: Combining 2D Gaussian function in the XY-plane and 1D Gaussian function in the Z-axis to form a spherical filter that reconstructs a 3D Gaussian function.

Figure 3.8 shows the work flow of our point radiation method. First, we store the discretized versions of the 1D and 2D Gaussian kernels in the texture memory. Then, the Gaussian functions are reconstructed using programmable graphics hardware. To begin the radiation process, the geometry shader receives the position and attributes of an input point $\mathbf{p}$ fetched via the vertex shader. Next, we compute the point radiation by first transforming the position of $\mathbf{p}$ into screen-coordinate to prepare it for rasterization. Then, we compute the first layer of the 3D footprint with a

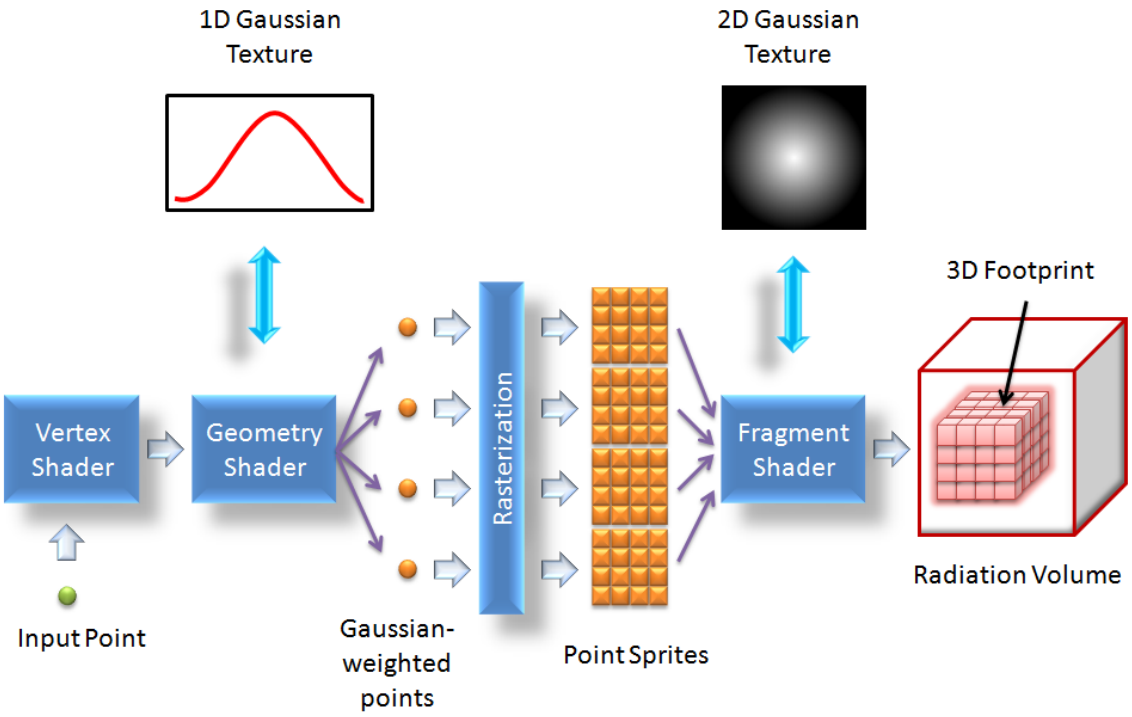

Figure 3.8: The work flow of our GPU-based point radiation technique.

viewport transformation scheme using the following equation:

$$layer0 = round\left(\frac{\mathbf{p}_z}{\zeta}\right) - R, \tag{3.5}$$

where $\mathbf{p}_z$ is a normalized depth value of $\mathbf{p}$ and ζ is the depth of the voxel dimensions. Next, we iterate through $layer0$ to $layer(2R-1)$ and duplicate a point primitive for each layer to reconstruct the 3D footprint. Each point should be associated with two attributes: (1) the $ZWeight$ sampled from the 1D Gaussian texture computed from the normalized layer coordinate and (2) the sample value s of point $\mathbf{p}$ (e.g. intensity value).

After the rasterization step, the set of points emitted by the geometry shader are converted into 2D point sprite fragments. Next in the fragment shader, we look

up the 2D Gaussian texture with the 2D point sprite coordinate and combine the result with $ZWeight$ to form a 3D energy value ω corresponding to a 3D footprint location. The final fragment value is then combined with the sample value s and blended into the radiation volume [1].

3.5 Masking

In this work, we employ a novel sketch-based interface for sculpting volumes. The user draws a closed free-form sketch on the screen to define the shape of the sculpting tool or specify a region of interest for sculpting. The information of this region is retrieved by our system to construct a computational mask. The mask is moved along a trajectory to sculpt or clip away parts of the volume. The masking process extends the basic point radiation method to create a mask radiation scheme for cutting volumes. This approach is intuitive and flexible in which the user is allowed to design a free-form sculpting tool (by using a pen and a tablet) similar to how an artist would paint on the canvas.

3.5.1 Mask Generation

The first step in the masking process is to generate a binary computational mask, which is composed of 1s and 0s, where 1 indicates that the pixel is covered by the sketched area and 0 means that the pixel is outside the area. Figure 3.9 illustrates the process for generating the mask. At first, the user places a stroke on the screen and a closed curve is obtained. Next, we fill the enclosing area using the stencil

[1] For blending, we use glBlendFunc(GL_SRC_ALPHA, GL_ONE) to perform the accumulation.

buffer with a 1-bit color [60]. Then we save the content of the stencil buffer as a texture as demonstrated in Figure 3.9.

Figure 3.9: Generating the computational mask using the stencil buffer.

3.5.2 Mask Filtering

A binary computational mask can introduce aliasing effect due to an insufficient sampling rate. To avoid aliasing in the image space, we modify the mask generation process and perform a post-filtering step. Instead of using a mask with a binary format, we use floating-points to represent intermediate values between areas covered by the sketch and outside the sketch. This method produces a smooth blending on the boundary of the sketched region and further prevents aliasing in the volume space.

Post-filtering techniques are often used for approximating a continuous image during the discretization process. Post-filtering can be classified into two categories: uniform sampling and stochastic sampling. Uniform sampling involves rendering a virtual image at n times the resolution of the final screen image. The final image is then produced by averaging down the virtual image using a convolution operation [56]. Stochastic sampling is similar to the uniform version except that it jitters the

samples according to some schemes. In our experiments, we found better results in using stochastic methods over the uniform scheme.

With stochastic sampling, we apply jittering on a regular grid. New pixel positions are sampled within a sub-pixel (i.e. grid cell) — the final color is then reconstructed with different schemes. Jittering is a method that trades aliases with noise and can be used to approximate a Poisson disk distribution [15]. To apply jittering, each pixel on a 2D image is subdivided into sub-pixels. A center is located in each of the sub-pixel. Then a randomized position (i.e. an X and Y offset) with respect to the center is sampled from the original geometry (i.e. in the continuous space) within the sub-pixel. The collection of offset samples are therefore the jittered samples. Figure 3.10 illustrates jittering with a 2-times super-sampling resolution.

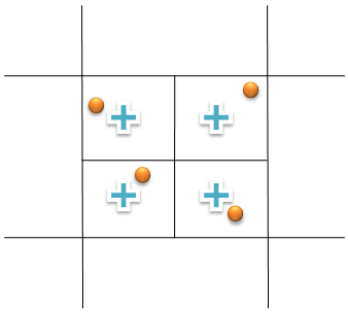

Figure 3.10: Stochastic sampling via jittering of a regular grid.

The next step is to apply a reconstruction filter over the set of jittered samples. There exist several choices for the reconstruction filter, including the box (constant function), tent (linear function), Gaussian, cubic B-spline, cubic Catmull-Rom (i.e. spline), and cubic Mitchell-Netravali filter [51] [38]. Both the box and the tent filter do not provide enough smoothness. Gaussian provides very smooth result, but details are diminished. The cubic Mitchell-Netravali filter is a weighted combination of the

cubic B-spline and the cubic Catmull-Rom filter. In our experiments, we found that the cubic B-spline filter provides the best result (i.e. providing smooth results while maintaining sharpness and details) when compared to the aforementioned filters (Figure 3.15). The cubic B-spline function[2] is a piecewise polynomials and its second derivatives is continuous (C^2). The 1D uniform cubic B-spline filter can be described by the following equation [38]:

$$f_B(x) = \frac{1}{6}\begin{cases} 3|x|^3 - 6|x|^2 + 4 & |x|<1 \\ -|x|^3 + 6|x|^2 - 12|x|^3 + 8 & 1 \leq |x|<2 \\ 0 & \text{otherwise.} \end{cases}$$

Notice that x can be the local sample coordinate relative to the center of the filter with the radius of two. To construct a 2D cubic B-spline filter, we first discretize the 1D function into an n-dimensional vector b. Then we compute $b \cdot b^T$ to yield a n x n matrix containing discretized entries of a 2D cubic B-spline filter. For example, when $n = 3$, we can derive the matrix as the following:

$$b = \frac{1}{6}\begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix}, \quad b \cdot b^T = \frac{1}{6}\begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix} \cdot \frac{1}{6}\begin{pmatrix} 1 & 4 & 1 \end{pmatrix} = \frac{1}{36}\begin{pmatrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{pmatrix}. \tag{3.6}$$

Figure 3.11 illustrates the effect after applying stochastic sampling with a cubic B-spline reconstruction filter.

3.5.3 Mask Sweeping

In the last stage of masking, we move the mask along a trajectory and build a swept-volume that can be used for volume sculpting or volume clipping. Figure 3.12 depicts

[2]Cubic B-spline forms a basis for the space of cubic splines.

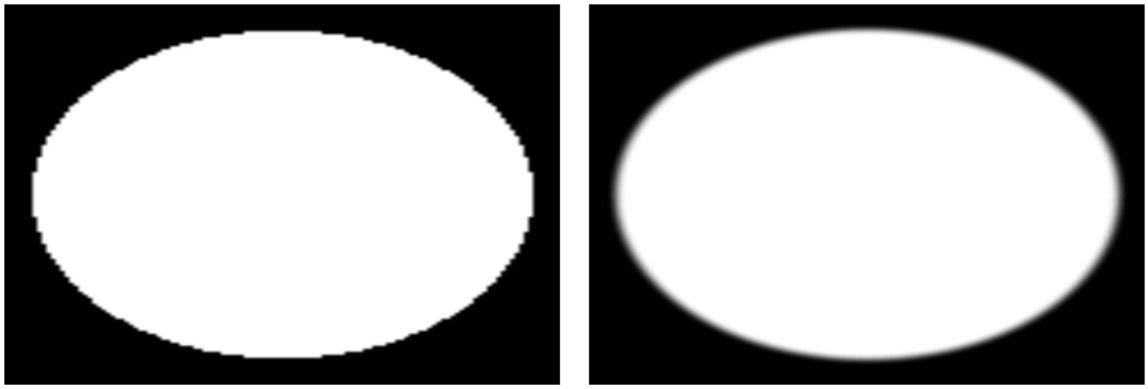

Figure 3.11: Sketching an elliptical mask. (a) shows the original and un-filtered mask. (b) shows the mask filtered with stochastic sampling and reconstructed with a cubic B-spline filter.

the mask sweeping process. The mask is positioned on the sweeping path by using a

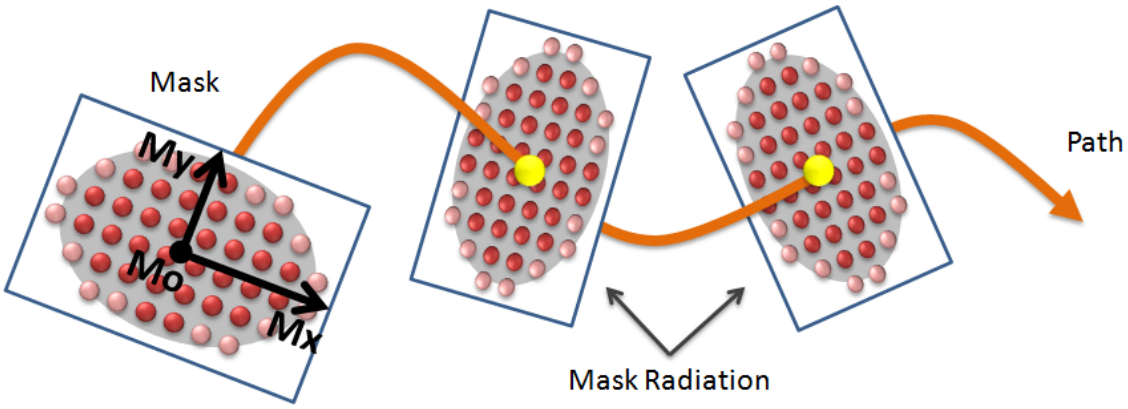

Figure 3.12: Sweeping a mask along a trajectory and distributing energy with mask radiation.

local coordinate system defined by the center of the mask $\mathbf{M_o}$, $\mathbf{M_x}$ and $\mathbf{M_y}$, the unit vectors along the x-axis and y-axis respectively. The orientation of the mask (i.e. determined by the $\mathbf{M_x}$ and $\mathbf{M_y}$ vector) can be computed by using a Frenet frame [45] or parallel transportation frame [26]. To build the swept-volume, we augment the

point radiation strategy to include mask as a radiation unit. For each pixel on the mask, we combine the corresponding mask value with point radiation to distribute energy (or weights on the mask) in the volume space. The result is recorded in a volume accumulating the set of point radiation operations along the mask sweeping path. To compute the object-space positions $\mathbf{M_{ij}}$ for the pixels on the mask, we offset the center of the mask with relative coordinates (ξ_i, ξ_j) in the $\mathbf{M_x}$ and $\mathbf{M_y}$ direction:

$$\mathbf{M_{ij}} = \mathbf{M_o} + \xi_i \mathbf{M_x} + \xi_j \mathbf{M_y} \tag{3.7}$$

where i and j range from 0 to the mask resolution in the X and Y direction respectively; and ξ_i and ξ_j are offsets calculated by converting mask indices from image-space coordinates (e.g. $0, 1, 2, ...$) to object-space coordinates (e.g. $0.0, 0.1, 0.2, ...$) based on the size and resolution of the mask. To accelerate the mask radiation process, we store the set of mask offset values (i.e. the set of ξ_is and ξ_js) in a vertex buffer to minimize data updating and optimize performance on the GPU. This way, we only need to specify the mask local frame $\mathbf{M_o}$, $\mathbf{M_x}$, $\mathbf{M_y}$ for every mask radiation operation.

3.6 Results and Analysis

In this section, we present volume modeling results to demonstrate our point radiation technique and discuss possible effects that can be generated by using different radiation field sizes, anti-aliased filters, and precision in the volume data. We also provide a comparison for building swept volumes between a projection-based approach and our point radiation algorithm. Finally, we illustrate our idea of using

mask-swept volume for sculpting a real volumetric data.

We render the volumetric data with a real-time ray-casting technique implemented in the graphics hardware. We adapt the same method as described by Scharsach [48] (more information can be found in Chapter 2). In the rendering process, the fragment shader is used to perform tri-linear interpolation of the volumetric data. We use an iso-value of 0.5 to determine the visibility of a voxel in the radiation volume (Section 3.4). In volume modeling (i.e. building a swept volume from an empty cube), voxels with radiation values greater than or equal to 0.5 are rendered, and not rendered otherwise. In volume sculpting, voxels with radiation values less than 0.5 are rendered instead.

3.6.1 Radiation Field Size

The field size of point radiation is defined by the radius of the 3D kernel. Kernel size is a major factor for determining the smoothness and performance of a potential sculpting operation. Figure 3.13 illustrates mask radiation of two elliptical planes crossing each other with kernel radius 2, 3, and 4 respectively. Different thickness of the plane can be observed with increasing kernel radius. A small kernel size in (a) produced a thin sweeping layer that preserves sharpness but introduces artifacts due to insufficient sampling of the kernel filter. A larger kernel size in (c) generated thick slices which guarantees smoothness but fails to produce sharp edges. In our experiment, we found that a kernel radius of 3 (as seen in Figure 3.13, b) produces a prominent result with the best performance for data ranging from 128^3 to 512^3. A large kernel size produced more smoothing effect but created a dramatic impact on performance and sharpness.

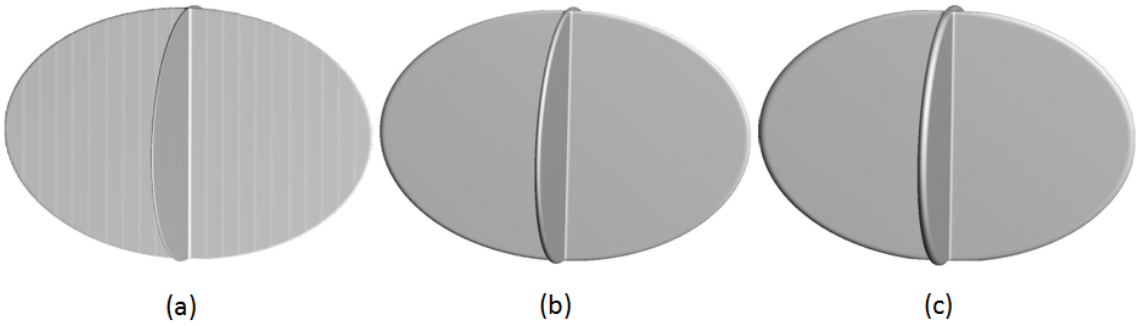

Figure 3.13: Mask radiation of two elliptical planes crossing each other. Kernel radius of (a) 2, (b) 3, and (c) 4 is shown. Different radiation field strength produced varying thickness. A trade-off between sharpness and smoothness can also be seen from left to right.

3.6.2 Anti-Aliased Filters

Proper mask filtering ensures a smooth result for volume sweeping when a mask is radiated along a trajectory. We applied various anti-aliasing filters on the sketched mask and obtained intrinsic difference in the results. Figure 3.14 compares results from applying uniform and stochastic sampling techniques. An elliptical sketch was used as the mask and swept along a pre-defined path to form a helix. Figure 3.14 (a) reveals a severe aliasing effect while directly using a binary computation mask. In (b), a simple OpenGL polygon smooth function (i.e. glEnable(GL_POLYGON_SMOOTH)) was used but did not provide evident improvement. For (c), (d), and (e), we applied uniform sampling by utilizing different subdivision filters. In (c) and (d), we applied a cubic B-spline filter with one and three levels of subdivision respectively. In (e), we used a three-level Chaikin subdivision filter. Although a three-level cubic B-spline filter smoothed the resulting volume somewhat, but overall the aliasing effect was still visible. However with a stochastic sampling approach, a dramatic difference emerged, as seen in (e).

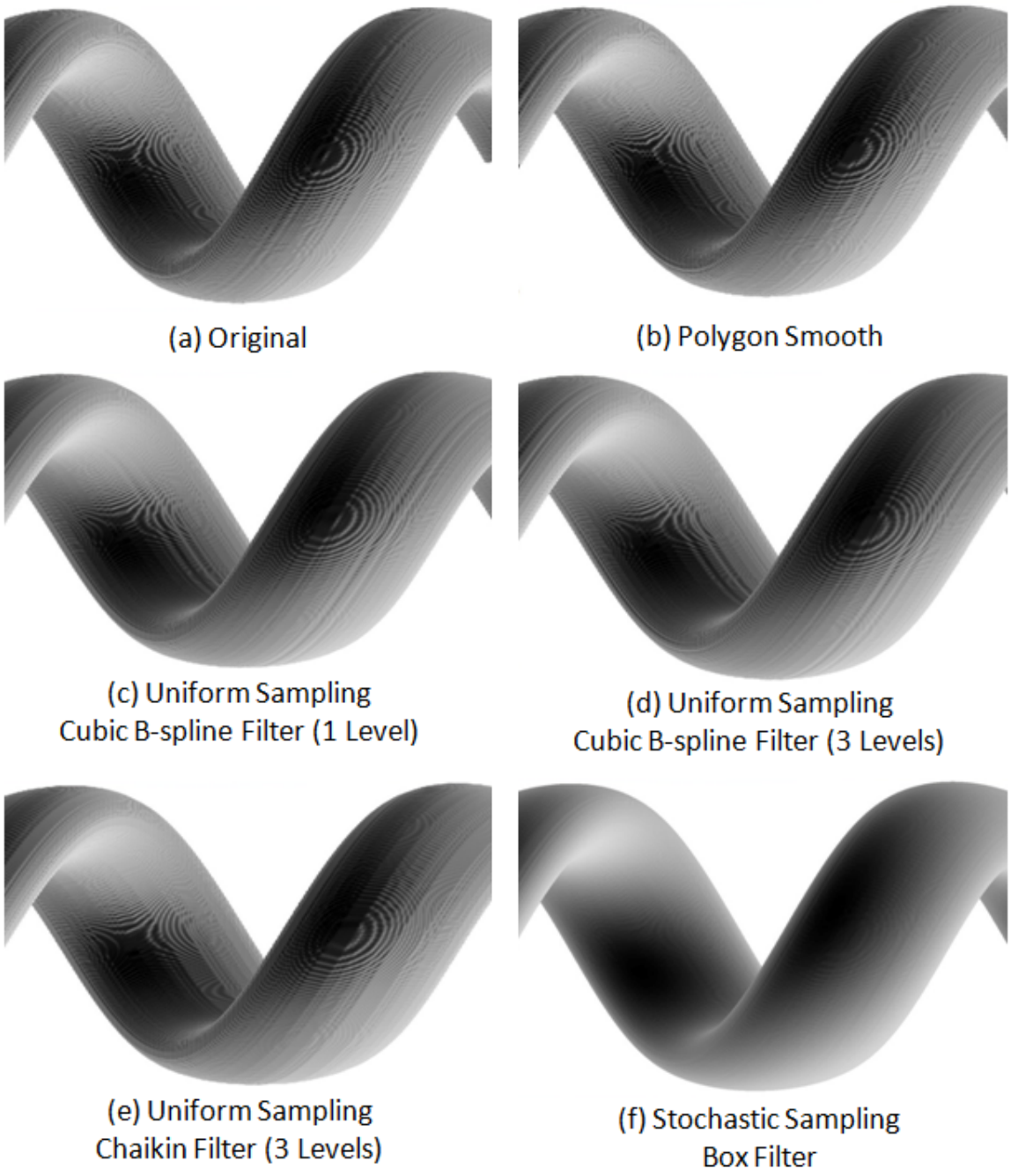

Figure 3.14: Helix swept by an elliptical mask anti-aliased with various schemes: (a) the original mask; (b) the OpenGL polygon smooth function; (c) uniform sampling with a cubic B-spline subdivision filter (one-level); (d) uniform sampling with a cubic B-spline subdivision filter (three-level); (e) uniform sampling with a one-level Chaikin filter; and (f) stochastic sampling with a box reconstruction filter.

With a vast improvement in stochastic sampling, we further experimented with different reconstruction filters. Figure 3.15 compares three reconstruction filters as well as the original result. Figure 3.15 (b), (c), and (d) show the swept volumes obtained by using the box filter, the cubic Mitchell-Netravali filter, and the cubic B-spline filter respectively. Overall, the cubic B-spline reconstruction filter produced a best smoothing effect on the original data.

Although the cubic B-spline filter outperformed the other types (Figure 3.15, d), a small amount of circular pattern could still be observed on the surface of the helix. To further reduce the pattern, we varied the filter width of the cubic B-spline function. Figure 3.16 compares swept volumes generated by using a filter width of (b) 10, (c) 20 , (d) 40, (e) 50, and (f) 100. We obtained a smooth surface after the filter size of 40 but noticed no further improvement at 50 and 100. Therefore, we concluded that a filter size of 40 is adequate to produce a close-to-perfect swept volume boundary.

In Figure 3.17, we illustrate two swept volumes anti-aliased by jittering and reconstructed with a cubic B-spline filter of size 40. Masks with a square and a maple leaf shape were used to sweep out a helix.

3.6.3 Volume Precision

The precision in the recording radiation volume plays an important role for rendering a smooth swept volume. Figure 3.18 compares the use of an 8-bit and a 16-bit floating-point volume texture. With 8-bit precision, some aliasing effect appears as under-sampling in the volume space. By using a 16-bit volume texture, a smooth swept volume could be faithfully reproduced. We also experimented with 32-bit

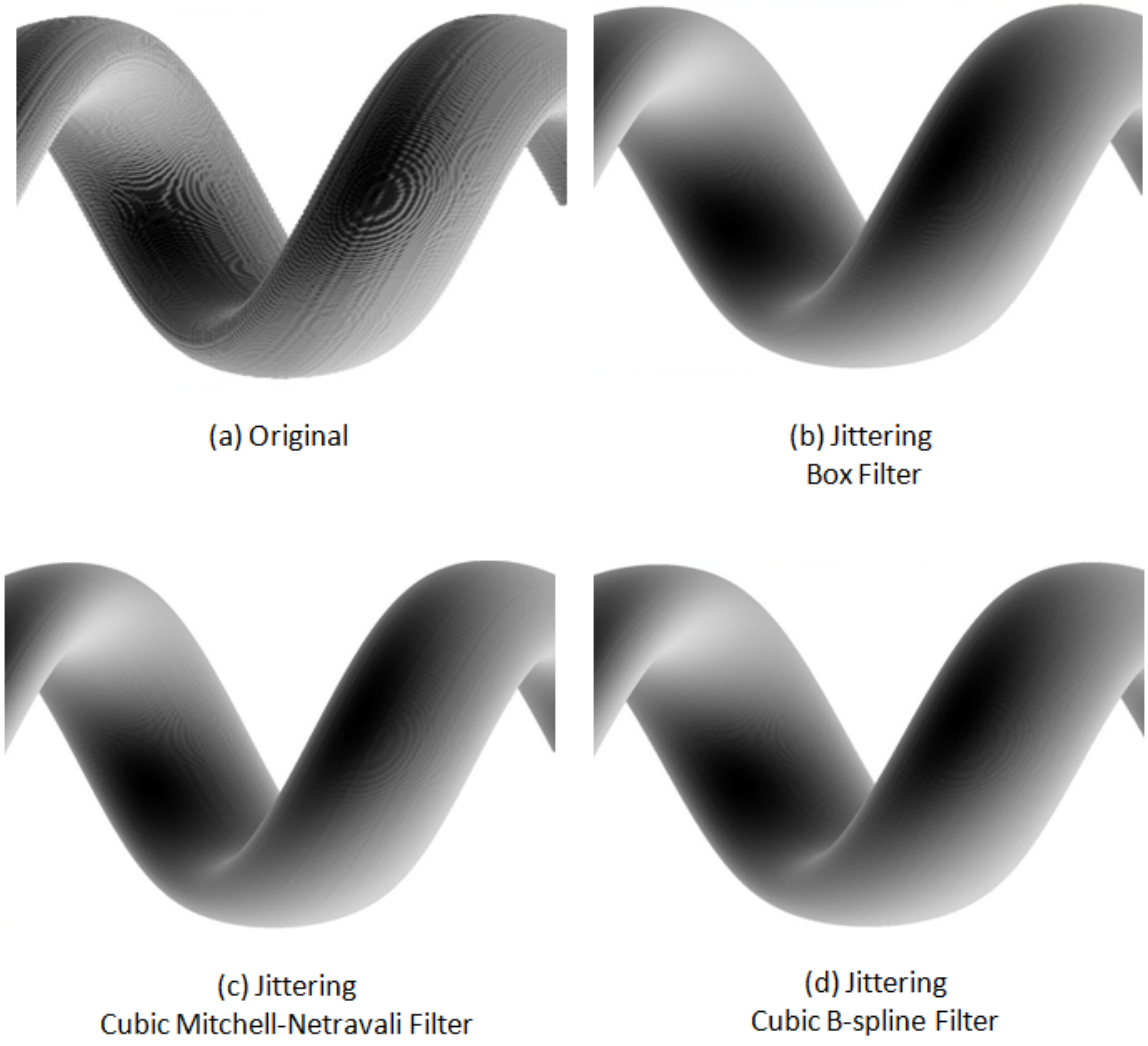

Figure 3.15: Helix swept by an elliptical mask anti-aliased by jittering with various reconstruction filters: (a) the original mask; (b) the box filter; (c) the cubic Mitchell-Netravali filter; and (d) the cubic B-spline filter.

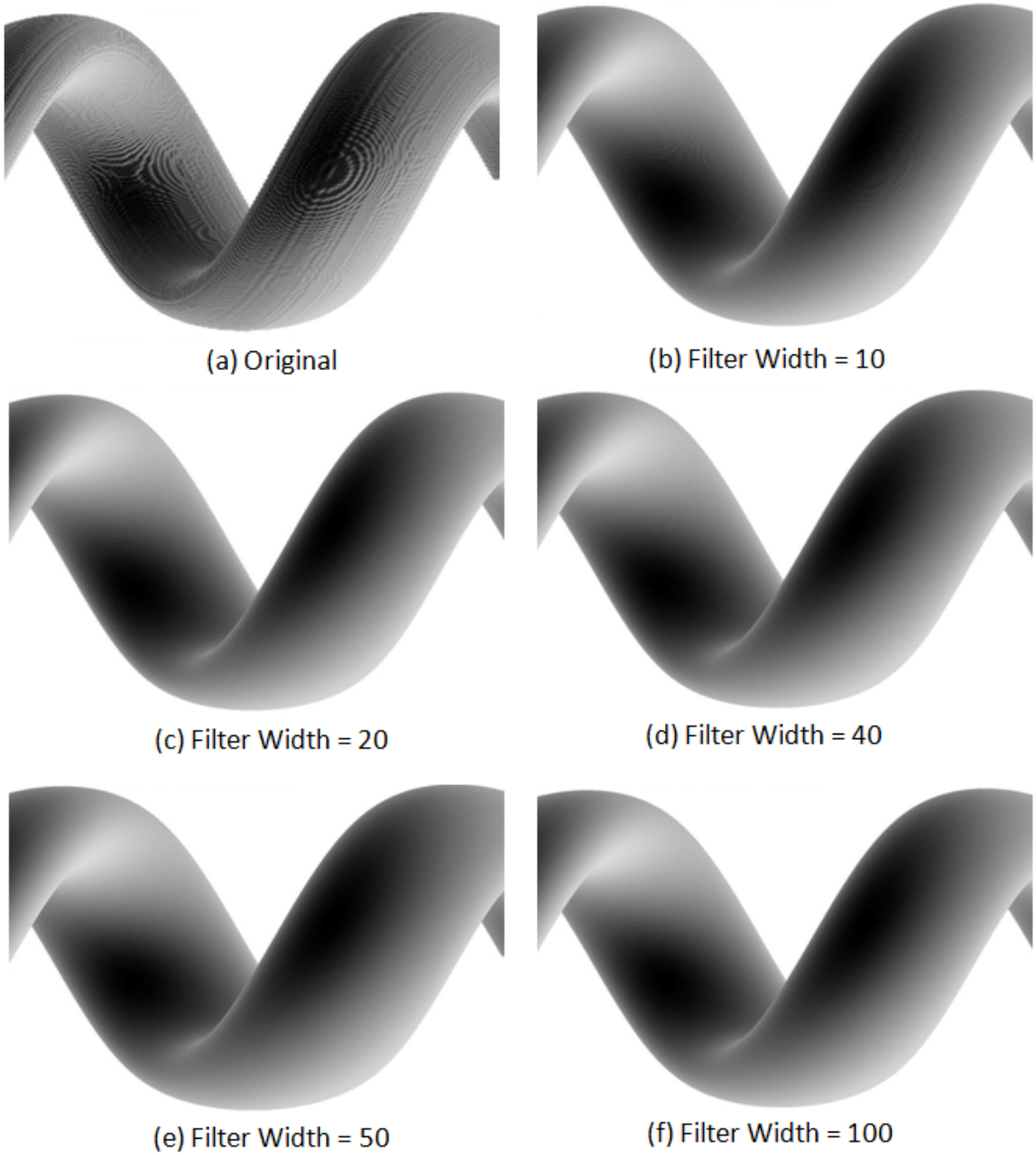

Figure 3.16: Helix swept by an elliptical mask anti-aliased by jittering with different cubic B-spline reconstruction filter sizes: (a) the original mask; (b) filter width = 10; (c) filter width = 20; (d) filter width = 40; (e) filter width = 50; and (f) filter width = 100;

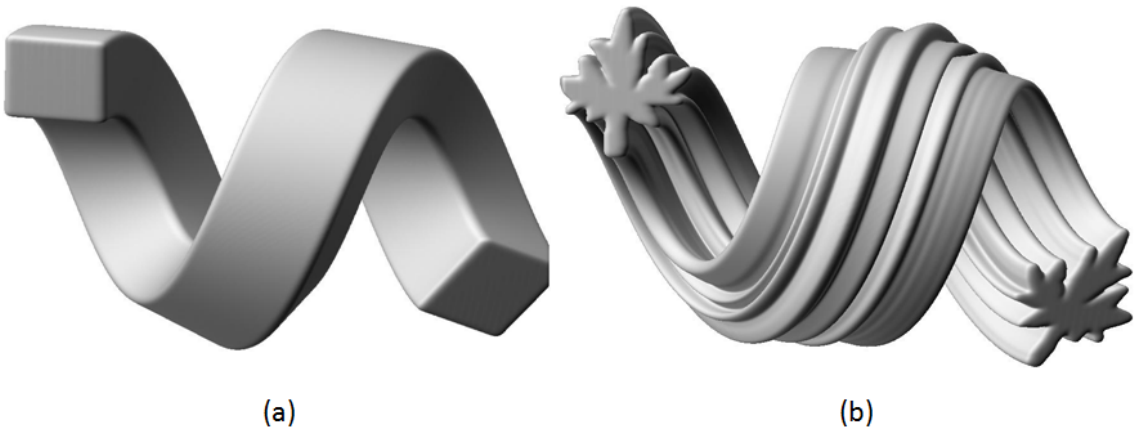

Figure 3.17: Swept volumes produced by sweeping (a) a square mask and (b) a sketched mask with a maple leaf shape. Both results were anti-aliased by jittering and reconstructed through a cubic B-spline filter with size = 40.

precision with the expense of extensive memory consumption, but the result has no dramatic improvement over the 16-bit texture.

3.6.4 Projection-Based Vs. Point Radiation

Here we compare our point radiation approach with a projection-based technique (as suggested by Winter and Chen in [59]) for generating swept volumes. Figure 3.19 depicts the joining point of two cylindrical pipes swept by an elliptical mask. From using a projection-based technique (Figure 3.19, a), aliasing effect appeared at the end point of the left pipe due to a slanted orientation. In Figure 3.19 (b), we tried applying a tri-cubic interpolation instead of tri-linear interpolation to render the scene in Figure 3.19 (a). A slight improvement on the end point of the pipe could be observed due to a global smoothing operation with a cubic filter, but the staircase effect was still visible. Figure 3.19 (c) illustrates our point radiation technique applied in the same situation. A smooth end-cutting on the left pipe could be observed. Point

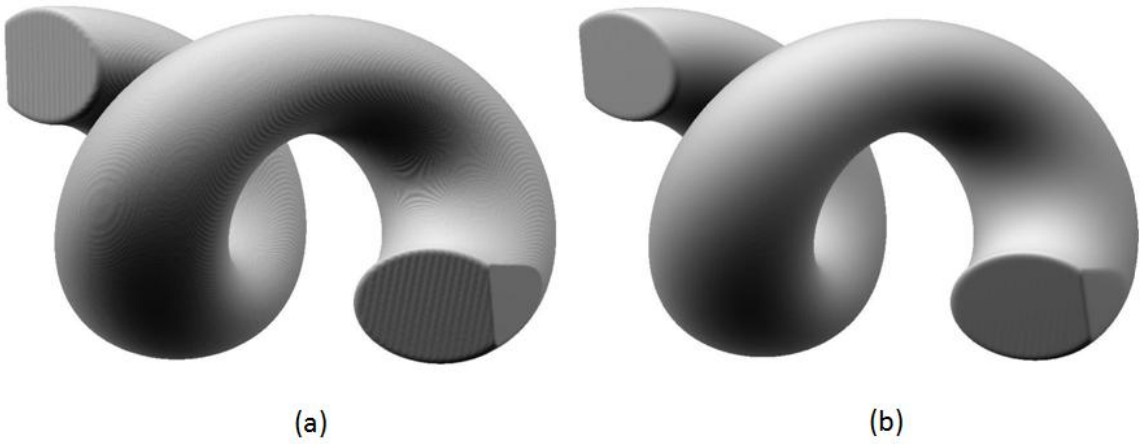

Figure 3.18: Helix swept by an elliptical mask. The recording radiation volume was represented by using (a) 8-bit and (b) 16-bit floating-point precision respectively.

radiation enabled us to achieve global anti-aliasing and allowed us to generate swept volumes that are free of artifacts.

3.6.5 Volume Sculpting

The mask sweeping approach is not only used for modeling a volume, but it can also be utilized as a volume sculpting or volume clipping tool. Figure 3.20 illustrates a clipping operation applied on the statue leg data set with a helical swept volume.

3.7 Chapter Summary

In this chapter, we presented our solution to the volume sculpting problem by implementing a novel point radiation technique leveraging recent features in the programmable graphics hardware. For direct splatting in the volume space, we utilized an array of hardware-accelerated point sprites to obtain dramatic performance improvement (at a minimum of 128-times faster than the convention approach). We

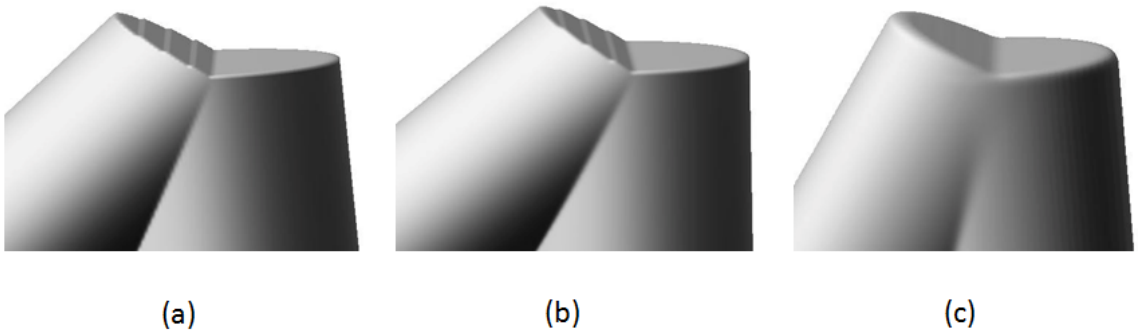

Figure 3.19: The joining point of two cylindrical pipes swept by an elliptical mask. (a) shows the swept volume produced by using a projection-based technique and rendered with tri-linear interpolation. (b) shows the same swept volume as in (a) but rendered with tri-cubic interpolation to provide a global smoothing effect. (c) shows the swept volume generated by using our point radiation technique and rendered with tri-linear interpolation.

also extended the point radiation technique to include mask operations, such as mask generation, mask filtering, and mask sweeping for producing a mask-swept volume system. At last, we explained our rendering strategy which utilizes a GPU programming paradigm. We also presented results and compared our technique with previous approaches.

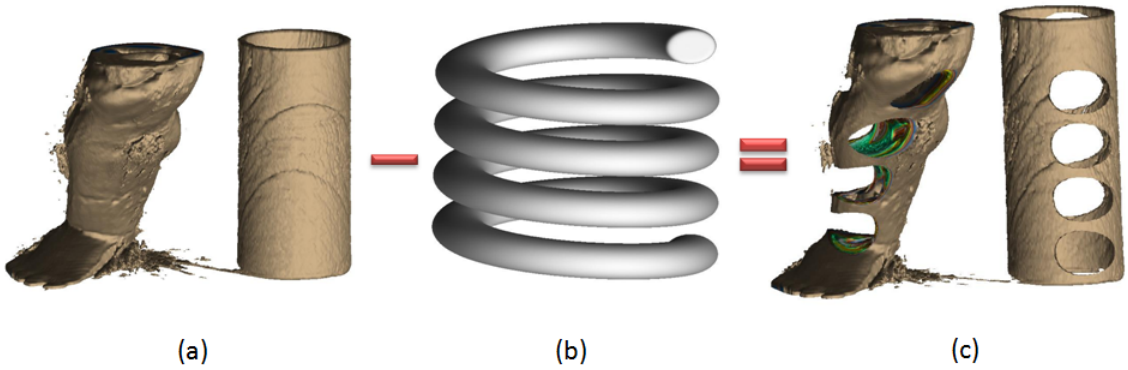

Figure 3.20: Clipping the statue leg data set with a helical swept volume. (a) shows the original statue leg (341x341x93) rendered in iso-surface mode. (b) shows a helix swept by a circular mask. (c) shows the effect of subtracting the original status leg by the helical swept volume.

Chapter 4

Sketch-Based Volume Tools

In this chapter, we present a suite of volume sculpting tools, mostly sketch-based, utilizing the point-radiation technique (Chapter 3). We employ the mask sweeping approach (Chapter 3, Section 3.5) to create the tool set. We also utilize GPU programming to achieve real-time frame rates. In Section 4.1, we first introduce the interface problem, and then we present our motivation and approach. In Section 4.2, we explain the fundamental building blocks which establishes our interactive environment. In Section 4.3, we discuss the details in building two kinds of drilling tools: view-dependent drilling and the laser tool. In Section 4.4, we describe the peeling tool that is used for examining internal materials. In Section 4.5, we propose the cut and paste tool which provides a dual-view in volume sculpting.

4.1 Introduction

In Chapter 3, we proposed a technique to model a swept volume by moving a computational mask along a 3D trajectory. The mask-swept volume could then be used for sculpting or clipping a volumetric data. However, the way in which the trajectory is defined was not discussed. For examples in Chapter 3, we have used pre-defined curves like the helix equation which could not be dynamically modified. In order to enable interactive sculpting operations, we need to incorporate a user-definable sweeping path for controlling the mask. One possible way would be to allow the

user to place a second stroke that specifies the trajectory. However, with a 2D input device (i.e. a pen tablet), we are limited to an x-y input coordinate system, which only allows the user to sketch a planar curve on the screen. To specify the third-dimension, we either have to map or unproject the 2D stroke into 3D (using a reference plane in space) or resort to other 3D sketching techniques. Instead of pre-defined or directly sketched trajectories, we derive a set of volume tools that has a natural candidate for the trajectory. Our solution contains tools like the view-dependent drilling, laser, and the peeling tool. The drilling and laser tools are special cases of sweeping along a straight line. The view-dependent drilling tool uses the view-direction as the sweeping trajectory. The laser tool allows the tool to be placed automatically on the surface and adapts the inverse surface normal as the sweeping trajectory. Although it is possible to manually specify the drilling direction with any angle, but this may require complicated device for this task to become effective. When the input involves only a simple device, such as the mouse, the view and the normal directions are best choices for drilling volumes and surface-based objects. The peeling tool assigns individual tool element with different trajectories using the surface information. In addition, we propose a cut and paste tool that allows the cut-away region to be rendered back to the scene. In our work, we are inspired by traditional medical illustrations (Figures 1.1 and 4.1) depicting specific surgical operations and procedures. Figure 4.1 (a) is a illustration of a traumatic crush injuries of the left hand. Skin tissues were removed in several places of the illustration to reveal the hidden fracture. Figure 4.1 (b) shows the removal of a bone flap from the skull during a craniotomy. Before the cutting operation was performed, the surgeon placed sketches on the skull to mark the opening section. For our volume tools, we

borrow the ideas from the skin removal illustration (Figure 4.1, a), the drilling and the surgical operations (Figure 4.1, b).

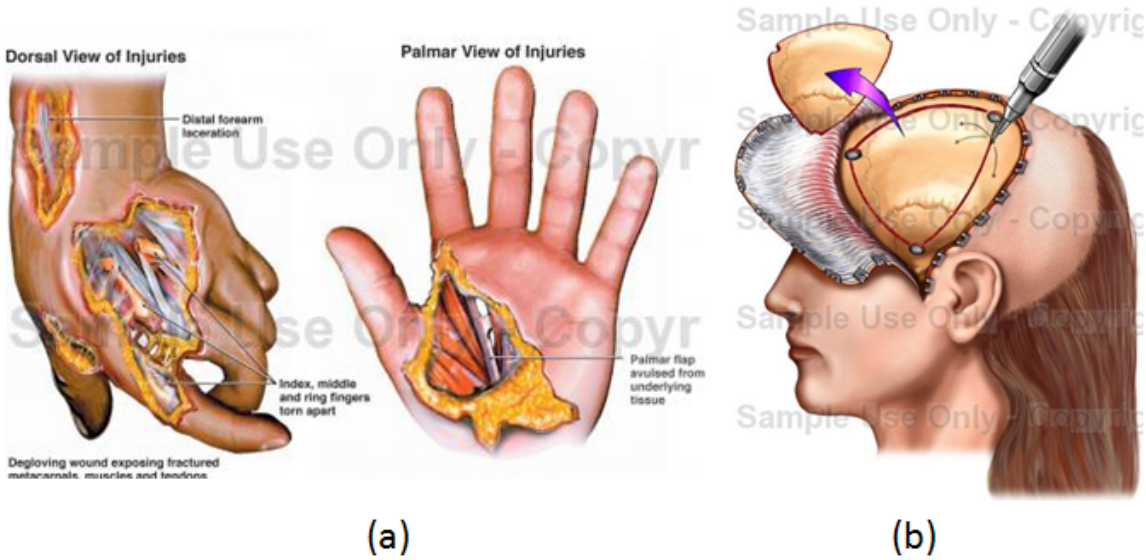

Figure 4.1: Example of traditional medical illustrations. (a) A traumatic crush injuries of the left hand. (b) The creation and removal of a bone flap from the skull during a craniotomy. (Copyright ©2007 Nucleus Medical Art. All rights reserved. www.nucleusinc.com)

Our system utilizes the sketch-based interface for constructing the shape of the volume tools. Each tool is operated by the sketched mask and derived from the basic mask sweeping scheme (as discussed in Chapter 3). To satisfy the diverse sculpting conditions, we create precise cutting tools by controlling the granularity of mask sweeping. We assign independent sweeping path and starting position to each element of the mask. Our approach not only eliminates the need for specifying a 3D sweeping path for volume sculpting, but it is also very intuitive as the process only involves a quick sketch and simple 2D movement. Although we adapt simple sweeping paths in the proposed tool set, possibilities of incorporating complex trajectories also

exist but remain as future work.

4.2 Tool Framework

In this section, we explain the framework components that are essential to building our suite of volume tools. There are four common tasks performed in our system. First, the input volumetric data is processed and displayed. Second, the sketch is rendered over the displayed volume. Third, we employ a surface detection process that constrains our set of sculpting tools. We find surface elements based on ray-casting and compute surface normals using geometric information in surrounding voxels. Finally, we adapt a two-pass rendering scheme for tool processing and the actual rendering step. In our work, we exploit advanced features in the graphics hardware to assist in designing most of the framework components to provide a fully integrated, real-time 3D environment.

4.2.1 Displaying Data

To display the data for volume sculpting, we first load the source file to collect optical and geometrical information. In our system, we focus on processing the raw (unsegmented) volumetric data as it is the most common form in clinical applications and represents a generic data type in other applications (e.g. geophysics). Raw data consists of voxel intensity values organized in a 3D array. Due to the large size and computational requirement of processing the raw data, we propose using GPU as a solution for storing and manipulating the data. To load the input data, we traverse the 3D array and save as a 3D texture on the GPU (Appendix B) for real-

time rendering. As we loop through the input data array, we construct an intensity histogram (by accumulating a voxel count for each intensity level) at the same time (see Section 2.2 for more details on histograms). A data range from the intensity histogram is then highlighted by the user to depict the desired classification for display. Figure 4.2 depicts the histogram selection at a low intensity range and the resulting image with X-ray and surface rendering.

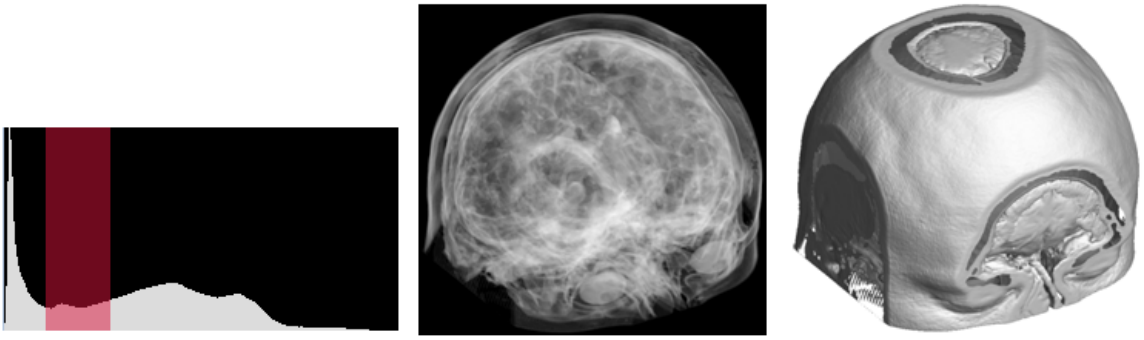

Figure 4.2: (a) Selecting a data range from the intensity histogram. (b) and (c) show the resulting images rendered in the X-ray mode and the surface mode respectively.

Direct rendering of the raw volumetric data produces results in gray-scaled images. It is sometimes difficult to visualize the details and distinguish the various materials from the displayed data. This may hinter the user from choosing a correct intensity range to work with. To improve rendering, we apply transfer functions that transform data attributes to color values (Section 2.2). The mapping in the transfer functions is usually done by analyzing various histograms. Commonly-used histograms include the intensity histogram, the gradient magnitude histogram, and the 2D histogram (gradient magnitude vs. intensity). However, two questions arise for implementing the transfer functions: (1) how do we compute the additional his-

tograms efficiently? (2) Is it possible to dynamically assign colors to individual voxels while rendering? To compute for the additional histograms, we need to traverse the 3D data array with three more passes: one pass for evaluating gradients of the entire volume, a second pass that accumulates voxel counts for constructing the gradient magnitude histogram, and a third pass for combining the previous histograms to create a 2D histogram. For a CPU-based implementation, a lengthy processing time may be required for iterating a large data set, such as 512^3. This can produce an overhead every time a new volumetric data set is loaded into the system as extra time is spent in the array iteration. To avoid large iterations on the CPU, we adapt a GPU-based approach for processing a 3D grid of voxels in parallel.

To process voxels on the GPU, we decompose the task of processing a 3D grid into a stack of 2D grids. For each 2D grid, we store the positions of the grid points in a vertex buffer. The vertex buffer is then rendered to the z^{th} volume layer. We use the instancing feature on the graphics hardware (Appendix B) to quickly render the vertex buffer d times (d is the volume depth) for processing the entire 3D data set. In this approach, we are only required to issue one draw call to perform computation for the entire volume and the result can be returned promptly. Figure 4.3 gives an overview of our GPU processing pipelines for computing the gradient volume as well as the histograms that are used in the transfer functions. In the first pipeline, the intensity volume is used to generate the gradient volume. In the second pipeline, the gradient volume is referenced for computing the gradient magnitude histogram. In the last pipeline, both the intensity and gradient volume are referenced for computing the 2D histogram.

To compute the gradient volume, we utilize the instanced vertex buffer (Appendix

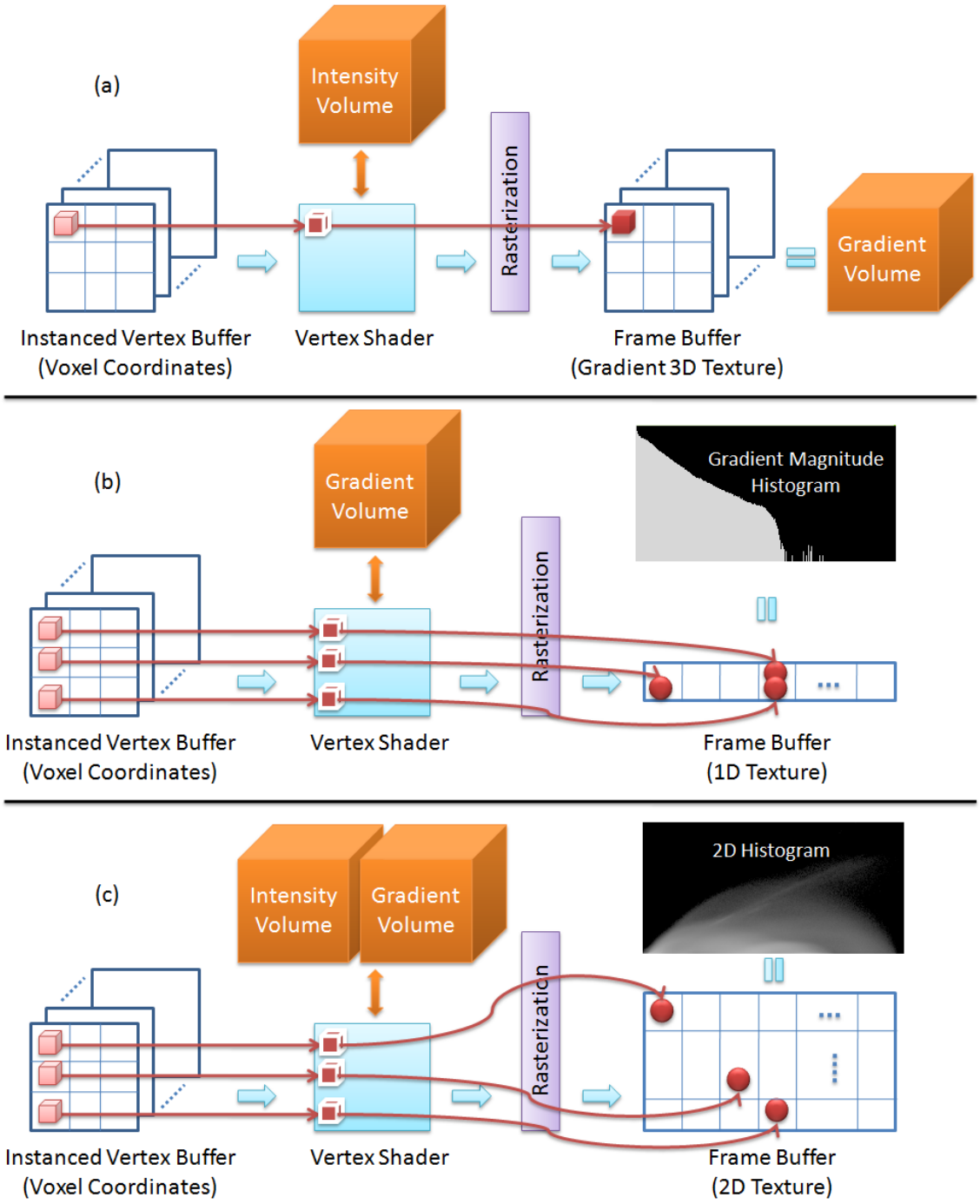

Figure 4.3: GPU processing pipelines for computing (a) the gradient volume, (b) the gradient magnitude histogram, and (c) the 2D histogram.

B, Section B.4.3) to iterate through the grid of voxels. For each voxel, the vertex shader looks up intensity values of the neighborhood from the intensity volume and apply the central difference operator to compute the gradient vector. For a voxel with location (x_i, y_i, z_i), the central difference operator is defined as:

$$\begin{aligned}\Delta g(x_i, y_i, z_i) = \quad \{ \quad & f(x_{i-1}, y_i, z_i) - f(x_{i+1}, y_i, z_i), \\ & f(x_i, y_{i-1}, z_i) - f(x_i, y_{i+1}, z_i), \\ & f(x_i, y_i, z_{i-1}) - f(x_i, y_i, z_{i+1})\}.\end{aligned}$$

The magnitude of the gradient vector $||\Delta g||$ is then output to the corresponding voxel location in the gradient volume (Figure 4.3, a).

With gradient information available for the entire volume, we now compute the gradient magnitude histogram by rendering a second pass on the GPU (Figure 4.3, b). The frame buffer pixels are now organized as a 1D array storing the voxel counts of the histogram chart (i.e. the heights of the histogram bars). For each voxel, the vertex shader looks up the gradient magnitude from the gradient volume and redirects the voxel to the fragment location that represents the corresponding gradient magnitude value. We accumulate the voxel count of the histogram with alpha blending (i.e. adding the source and destination color) and store the result in a 1D texture. The computation for the 2D histogram (gradient magnitude vs. intensity, Figure 4.3, c) is similar to the 1D histogram. For each voxel, the vertex shader looks up the intensity and gradient magnitude from the respective volume. Then the voxel is redirected to a 2D texture accumulating the voxel count for the 2D histogram (gradient magnitude vs. intensity). To render the resulting histograms, we simply apply texture mapping from the computed 1D and 2D textures.

To assign colors to individual voxels, we could apply the transfer function for each voxel on the CPU and store a color volume on the GPU for rendering. However, this may create a potential memory-overflow problem. In addition, updating the color volume would require an off-line data transfer over the PCI bus. Instead of storing a static color volume on the GPU, we dynamically compute the sample colors during the rendering process. First, the user defines a color table (i.e. color mapping) over the displayed histogram. Then, the color table is stored as a 1D or a 2D texture on the GPU. During the rendering, the intensity or the gradient magnitude of the sample is used to look up the color table for dynamic shading computation.

Figure 4.4 demonstrates the results of applying transfer functions by using the intensity histogram, gradient magnitude histogram, and the 2D histogram. Colors are assigned to the various histograms and the results are rendered in X-ray and surface mode. For the examples in Figure 4.4, color updates could be performed simultaneously while the data is being rendered. This is because that we only need to update a simple color table to the GPU rather than updating an entire color volume.

4.2.2 Sketching on the Volume

Our system allows the user to place strokes directly on the displayed volume to define the tool mask. In order to maintain precision of the sketch, the input points from the stylus is stored as a point list and rendered as line segments without filtering. This ensures that every pixel from the raw stroke matches the volume features for mask modeling. Nevertheless, sketching a fine curve on the screen requires fast processing of the stylus input when complex rendering is involved. Since rendering

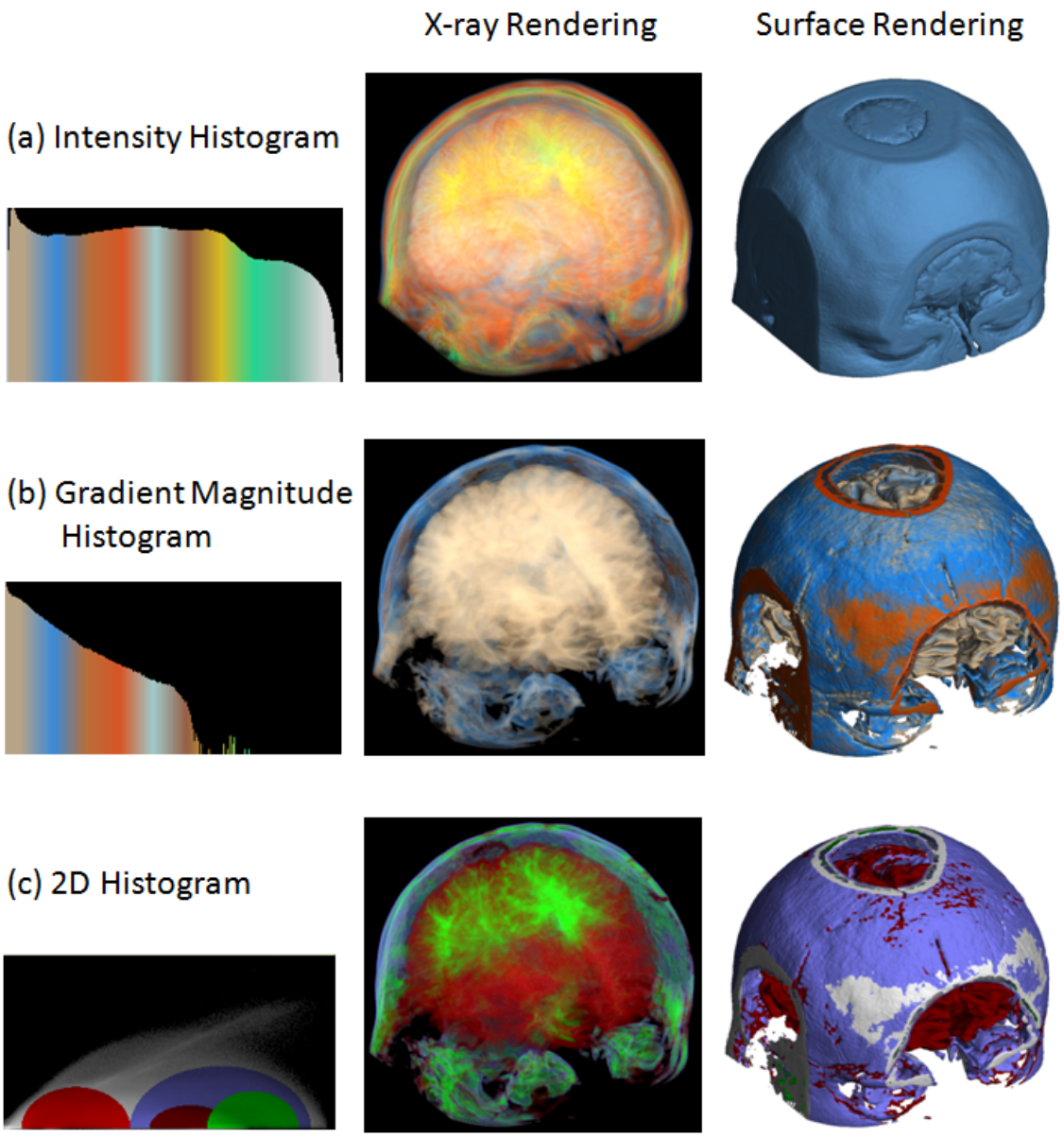

Figure 4.4: Applying transfer functions by assigning colors to (a) the intensity histogram, (b) the gradient magnitude histogram, and (c) the 2D (gradient magnitude vs. intensity) histogram. Results are rendered in the X-ray and surface mode.

the volume data is a costly operation, simultaneously sketching and rendering the volume is deemed to degrade the curve quality. In order to obtain smooth sketching, we freeze the background rendering (i.e. the volume raycasting) by saving the entire scene to a texture. Thus when the user places strokes on the screen, we render the screen-sized texture first followed by the input strokes. This avoids delays caused by the concurrent rendering of both the sketch and the volume data. Note that we do not process or filter the sketch so that the precision and originality of strokes are maintained. All of our tools start with a closed stroke. As the user finishes with a stroke, the start and end point are connected to form a closed curve. To determine the inside/outside of the closed stroke, we fill the enclosing area utilizing the stencil buffer with a 1-bit color [60]. The inside of the stroke is filled with 1s in the stencil buffer and 0s elsewhere.

4.2.3 Surface Detection

When using a 2D input device, such as a pen tablet, there exists ambiguity for sketching directly in 3D. To address this issue, we constrain our virtual tools to the visible surface of the displayed volume. This eliminates the need to specify a third dimension as the tool automatically adheres to the visible surface of the volume. In order to maximize performance and achieve interactive responses in our system, we adapt a GPU-based surface detection method based on ray-casting [56].

We start from the location of the tool (mask) plane and shoot rays along the view-direction. Next, the algorithm captures surface points and records the position and normal information. The surface points are defined by the rendering parameters (i.e. the thresholds selected from the intensity histogram) [48]. The positions and

normals of the detected surface points are then saved in multiple 2D textures for use in the volume tools. If the ray exits the bounding volume without hitting a surface, then we write a zero vector into the frame buffers (2D textures) to indicate the no-hit condition. Figure 4.5 depicts the raycasting process (right) with the result recorded into separate surface points textures (i.e. position and normal).

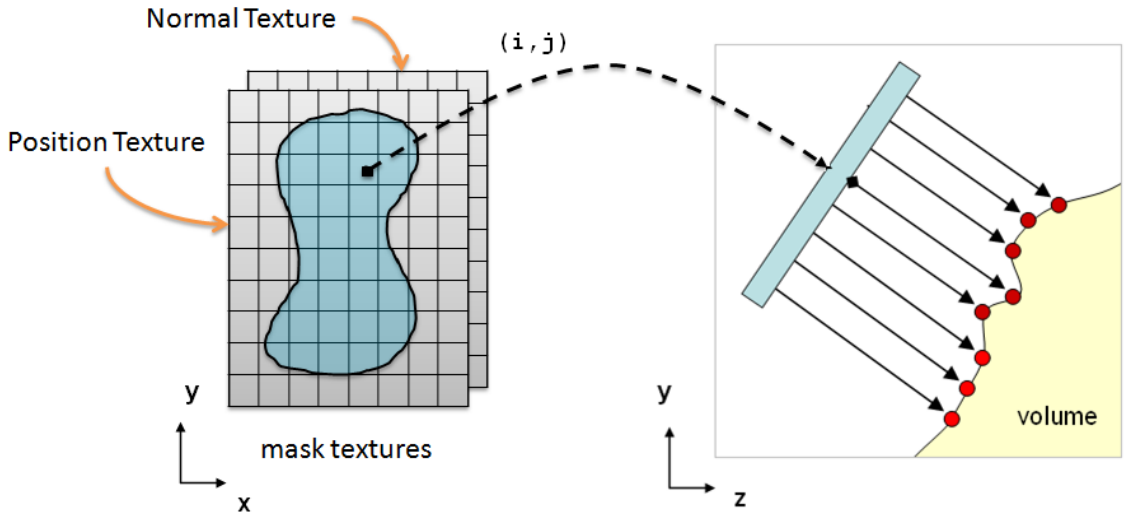

Figure 4.5: Surface detection via ray casting. The positions and normals of the surface points (red) are recorded into multiple 2D mask surface textures. (left) 2D textures associated with sketched mask. Each cell entry (r,g,b) can be used to store the surface points (red) position or normal. (right) Side view of the mask placed in front of the volume. Rays from each mask cell hit surface points in the volume surface. A zero vector is written with a missed hit.

4.2.4 Two-Pass Rendering

In our system, we use an extra rendering pass for processing and rendering the volume manipulation tools. When user interacts with the system, we construct a new volume, called sculpture volume, which is also saved on GPU in our implementation to achieve real-time performance. To process the tools, an entire rendering pass is

required for loading the user input, executing specific tool operation, and saving the sculpting result back into the sculpture volume (Figure 4.6, top). We reconstruct the tools utilizing the point radiation method (Sections 4.3 and 4.4) and combine 3D footprints in the sculpture volume (i.e. a radiation volume as described in the point radiation process in Section 3.4).

For the actual rendering of the sculpting results, we use a real-time raycasting algorithm utilizing the programmable shaders [48] in a second rendering pass. The sculpture volume from the first rendering pass is used for clipping voxels after rasterization (Figure 4.6, bottom). Voxels with radiation values in the sculpture volume less than 0.5 are visible, and otherwise invisible. We use intensity and gradient values to add colors onto the final pixel values.

4.3 The Mining Tools

Two types of mining operation can be performed in our system. The first type is view-dependent drilling. This tool allows the user to drill into the volume based on the current viewing parameters and is useful for removing regions that are occluding the volumes behind. The second type is the laser tool. The laser tool allows dynamic tool displacement and mines along the inverse surface normal direction. It is useful for automatically cleaning and removing surface regions to reveal hidden materials. In both tools, we take into consideration the pen pressure from the input stylus.

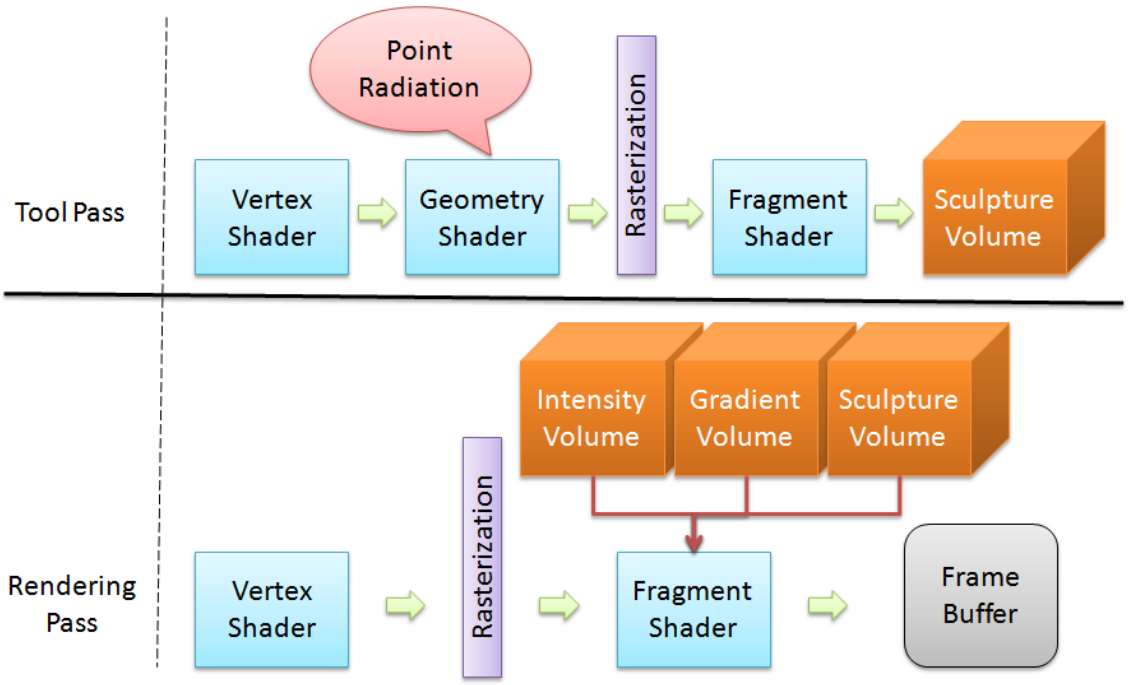

Figure 4.6: Two pass rendering showing the tool pass (top) and the rendering pass (bottom). The tool pass implements the point radiation algorithm, and the rendering pass serves a real-time raycasting algorithm.

4.3.1 View-Dependent Drilling

In view-dependent drilling, the user first draws a closed free-form stroke on the screen to define the shape of the tip (Figure 4.7, a). Next, a transparent region depicting the drilling mask is presented (Figure 4.7, b). When the pen is moved around, the displayed mask also follows the movement. As pressure is exerted on the pen tip, the drilling operation is performed with a drilling direction that is parallel to the current viewing direction (Figure 4.7, c). Note that other drilling directions are also possible, but subject to the limitation on the input device (e.g. the tilting angle of the pen can be utilized for specifying the desired drilling direction).

Each element of the mask has individual starting position but all sweep along

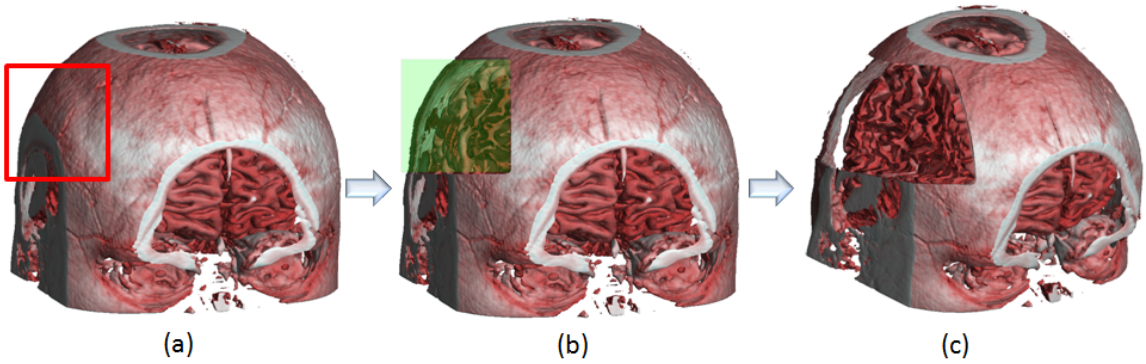

Figure 4.7: Our view-dependent drilling tool applied over the raw super-brain data: user sketches the tool shape on the screen (a), user applies pressure to drill on the skull (b), and a rectangular region is drilled (c).

the same direction (Figure 4.8, left). The mask follows the pen movement. Upon receiving a pressure change or slight pen movement, our system finds the surface points and responds with a drilling operation. To perform drilling, we attach the mask elements to the surface points and execute the mask radiation operation. We repeat this process according to the depth of drilling:

$$depth = depth_{max} * pressure_n, \tag{4.1}$$

where $depth_{max}$ is a maximum drilling depth allowance for each surface detection operation; $pressure_n$ is the instantaneous pressure supplied by the stylus and normalized to [0, 1]; and $depth$ is an integer defining the succession for the mask radiation process. Based on our experiment, setting the $depth_{max}$ value to the range of 8 to 12 is suitable for a volume of size 256^3. Having a maximum drilling depth of 12 prevents excessive removal of volume materials for every stylus input and help maintain stylus responsiveness. Successive mask radiation operations offset the positions of the mask elements with a small amount (i.e. relative to the size of a voxel) along the

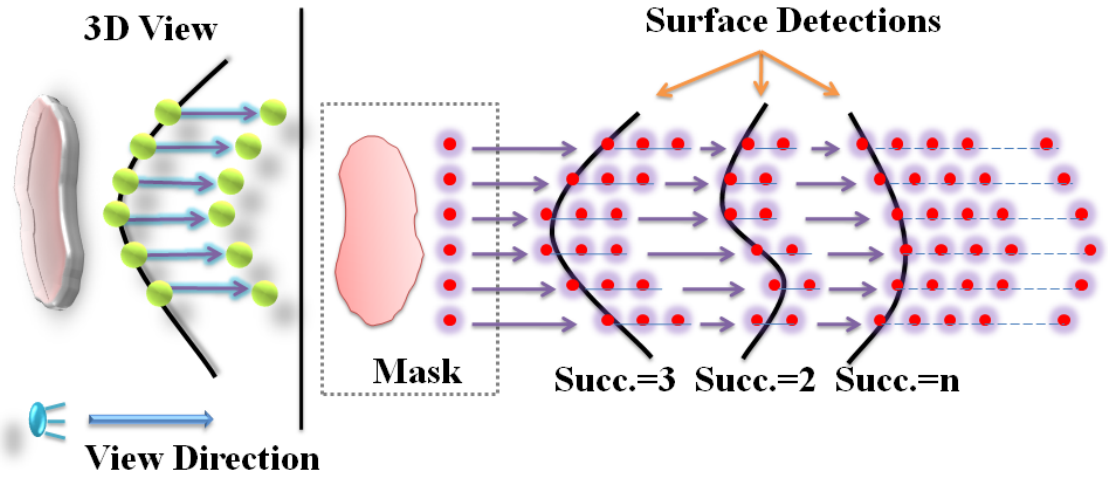

Figure 4.8: (Left) A 3D view of the view-dependent drilling tool showing the drilling positions and directions for each of the points on the mask. (Right) Point radiation applied with multiple surface detections. Mask radiation succession of 2, 3, and n are shown. The succession representing the input pressure amount determines the depth of drilling.

view-direction. Figure 4.8 (right) shows the mask elements implemented using the point radiation method and demonstrates three consecutive pressure levels received from the stylus with equivalent mask radiation succession of 3, 2, and n. With each pressure change, a new surface condition is determined. For each mask radiation operation, the position of point radiation is determined by:

$$\mathbf{p} = \mathbf{p_s} + \mathbf{v} * \Delta d, \tag{4.2}$$

where $\mathbf{p_s}$ is the position sampled from the surface position texture (created by the surface detection algorithm); $\mathbf{v}$ is the view-direction; and Δd is the voxel offset or displacement amount.

4.3.2 The Laser Tool

Our laser tool is analogous to the laser burning process frequently performed by dermatologists or during eye surgery. This tool is also useful in areas like laser marking or laser engraving. Similar to the view-dependent drilling tool, the user first draws a closed free-form stroke on the screen to define the shape of the laser (Figure 4.9, a). Next, a transparent region depicting the laser mask (Figure 4.9,

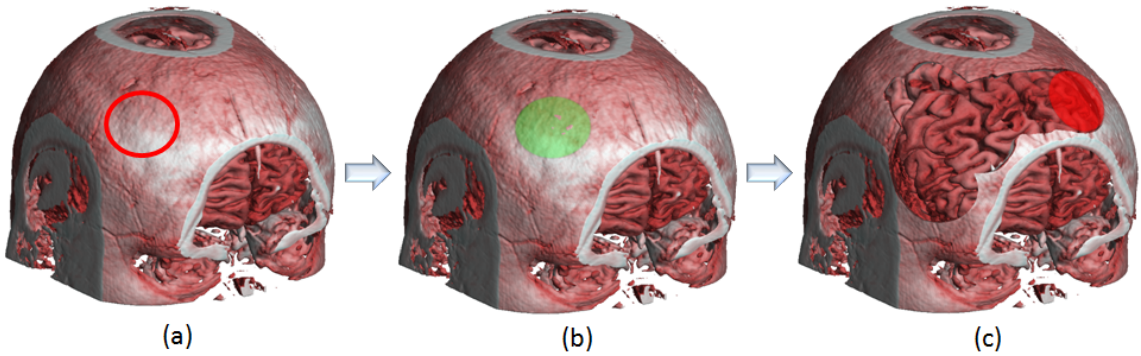

Figure 4.9: Our laser tool applied over the raw super-brain data: user sketches the shape of the laser tool on the screen (a), user applies varying pressures to remove portions of the skull (b), and more materials are removed to reveal the internal structure (c).

b) is presented. The user moves the pen around and applies pressure to perform mining operations (Figure 4.9, c). In contrast to view-dependent drilling, the laser mask is projected onto the displayed surface instead of being aligned with the view-plane. The orientation of the laser mask is determined by the surface normal. This approach allows the user to use a 2D input device and obtain a 3D manipulation effect. Similar to how dermatologists would hold the laser wand over the skin of a patient, our laser tool is automatically positioned perpendicular to the surface of the object. For the example shown in Figure 4.7, if the user wishes to mine a hole near

the top of the skull, it would be necessary to rotate the scene and then perform the view-aligned drilling operation. But with the laser tool, the user is able to drill the hole without manipulating the scene. It is only required to shift the pen to the top of the skull and the system dynamically computes a mining direction by detecting the surface normal.

The laser tool derives from the basic mask sweeping technique (Section 3.5.3). The mask elements have uniform starting positions and sweeping directions (Figure 4.10, left). As the pen moves, we update the mask position and orientation by sam-

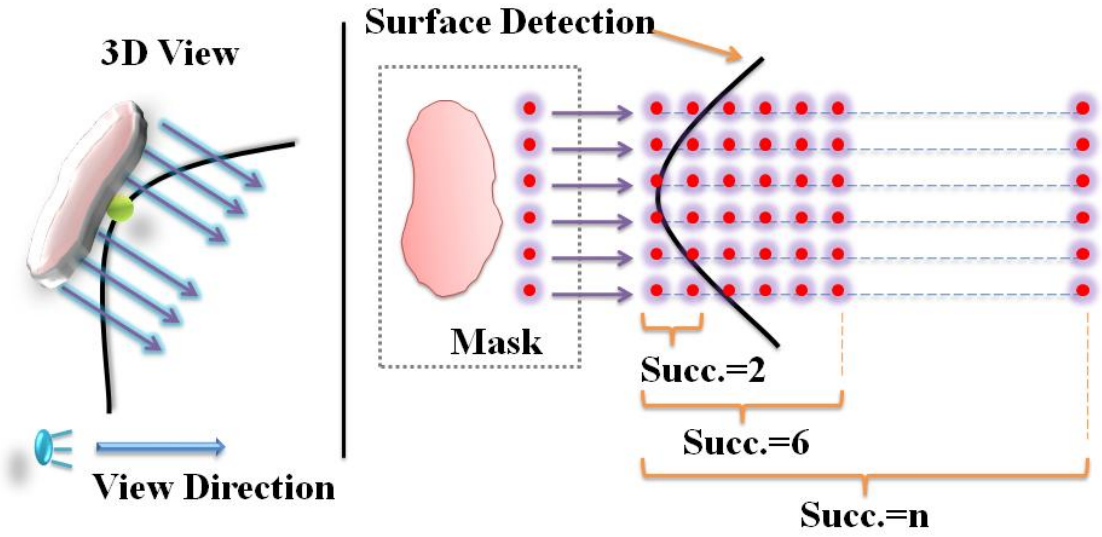

Figure 4.10: (Left) A 3D view of the laser tool showing the location of the mask and the mining directions for each of the points on the mask. (Right) Point radiation applied with a single surface detection. Mask radiation seccession of 2, 6, and n are shown.

pling the surface position and normal texture (Section 4.2.3) with the 2D coordinate received from the stylus. When pressure is applied on the pen tip, the laser operation is performed with the detected surface information. The depth of lasering is computed according to equation 4.1 with the amount of pen pressure applied. Our

experience has shown that setting the $depth_{max}$ value to the range of 18 to 22 works well for a volume of size 256^3 and prevents removing excessive surface layers. Figure 4.10 (right) depicts the laser mask burning operation implemented with the point radiation technique and shows the possible laser pressure levels with mask radiation succession of 2, 6, and n. To determine the mask position for mask radiation, we use the following formulation:

$$\mathbf{p_m} = \mathbf{p_s} - \mathbf{n_s} * \Delta d, \tag{4.3}$$

where $\mathbf{p_m}$ is the mask position for mask radiation; $\mathbf{p_s}$ is the position sampled from the surface position texture; and $\mathbf{n_s}$ is the normal vector sampled from the surface normal texture.

4.4 The Peeling Tool

The goal of the peeling tool is to simulate the skull opening operation as illustrated in Figure 4.1 (b). In the first stage, the user first draws a closed free-form stroke directly over the displayed volume to define a region for peeling (Figure 4.11, a). Next, the sketched region is displayed as a thin transparent layer overlaying the target area (Figure 4.11, b). To precisely control the peeling layers, the user drags down the pen on the tablet instead of applying pressures to specify the depth (or number of layers) for the peeling operation. The result is the paring of surfaces with edges cut in directions parallel to the top-most surface normals (Figure 4.11, c).

The peeling tool removes surface layers by combining the view-dependent mask positioning aspect of the drilling tool and the surface-normal-driven cutting characteristic of the laser tool. It derives from the mask-based sweeping scheme (Section

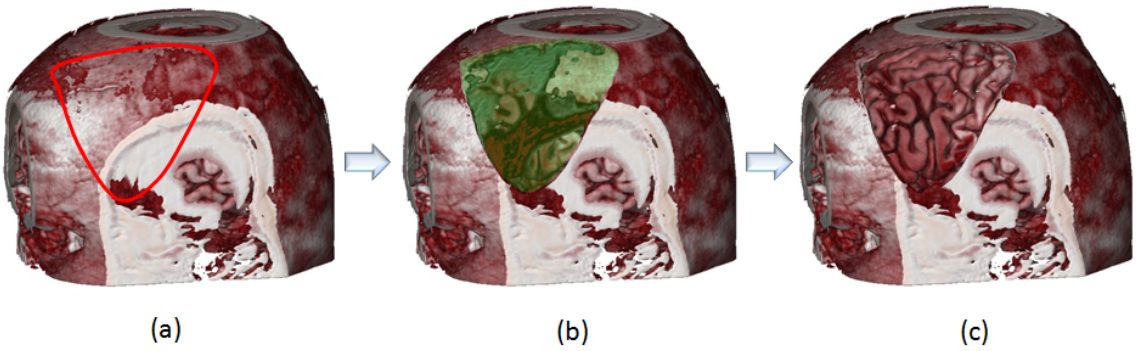

Figure 4.11: Our peeling tool applied over the raw super-brain data: user sketches the shape of the peeling region directly over the displayed volume (a), user drags down the pen to peel off layers on the skull (b), and the sketched region on the skull is removed to reveal the gray and white matter (c).

3.5.3) and can be seen as wrapping a mask and cutting towards the enclosed surface of the mask. Note that the condition of mask sweeping holds only if the target surface contains a smooth change of gradients. Each element of the mask has individual starting position and sweeping path (Figure 4.12, left). The peeling mask is fixed on the screen after the sketch. Then, the peeling operation is initiated as soon as the user drags the pen. For each pen shift, we define the instantaneous layers of peeling as:

$$layers = \Delta Y * s, \tag{4.4}$$

where ΔY is the vertical movement of the stylus (in terms of pixel count) and s is a scaling factor that converts pixel to layer unit. In our system, we obtain smooth peeling by converting 10 pixels to 1 layer (i.e. setting s to 0.1). To keep track of the current depth of peeling, we accumulate the number of layers that were peeled since the mask creation. Successive peeling operations continue from the recorded depth of peeling (i.e. the base layers). To compute the sculpture volume for peeling, we

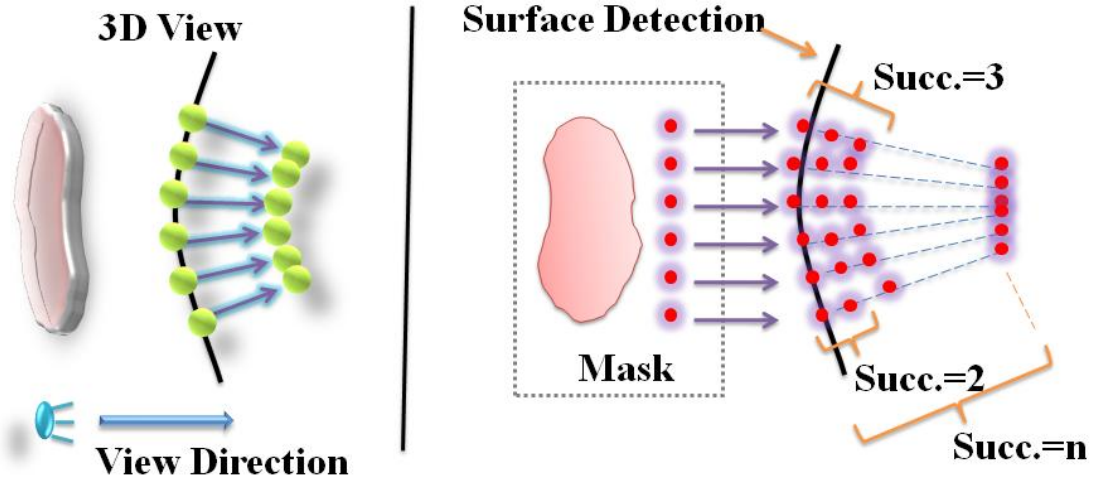

Figure 4.12: (Left) A 3D view of the peeling tool showing the peeling positions and directions for each of the points on the mask. (Right) Point radiation applied with a single surface detection. Mask radiation succession of 2, 3, and n are shown.

utilize the mask radiation technique. We relocate the starting positions of the mask elements to the detected surface points and assign corresponding inverse surface normals as individual sweeping paths. Figure 4.12 (right) depicts the peeling mask implemented with the point radiation technique and illustrates the possible layers of peeling with mask radiation succession of 2, 3, and n. For each mask radiation operation, the position of point radiation is determined by:

$$\mathbf{p} = \mathbf{p_s} - \mathbf{n_s} * l_b * \Delta d, \tag{4.5}$$

where $\mathbf{p_s}$ is the position sampled from the surface (recorded as texture); $\mathbf{n_s}$ is the vector sampled from the surface normal (i.e. recorded as texture during mask creation); and l_b is the base layers of peeling.

4.5 The Cut and Paste Tool

Figure 4.1 (d) illustrates the effect for simultaneously presenting the removed abdominal walls and the remaining parts of the anatomical manikin. Our cut and paste tool achieves the same effect by allowing the region *cut* by the drilling/lasering/peeling tool to be *pasted* back into a different position and orientation of the scene through an additional rendering criteria. Figure 4.13 depicts the rendering criteria for the *cut* and *paste* scene. As the radiation values range from 0 to 1.0, we select 0.5 as a natural candidate for distinguishing the cut and paste rendering. In a *cut* scene, voxels with associated radiation values (i.e. sampled from the sculpture volume) less than 0.5 are rendered. In a *paste* scene, the visibility criteria is reversed (i.e. those with radiation values greater than or equal to 0.5 are rendered instead).

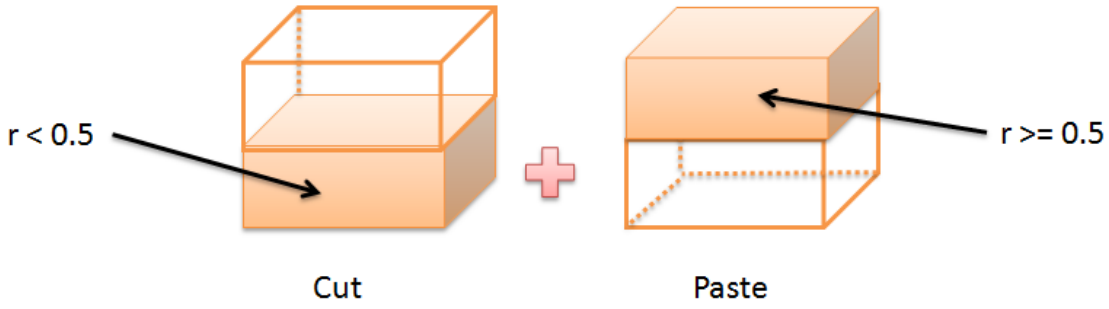

Figure 4.13: In a cut scene (left), voxels with radiation values <0.5 in the radiation volume are rendered. In a paste scene (left), voxels with radiation values $\geq$ 0.5 in the radiation volume are rendered.

To simultaneously display the *cut* and *paste* scene, we perform dual-rendering with our GPU-based ray-casting engine. In the first cast, we check for samples whose values are less than 0.5 in the sculpture volume; and in the second cast, we check for values that are greater than or equal to 0.5. To combine the scenes, we utilize

the depth-buffer to yield a correct blending over the two volumes (i.e. voxels that are closer to the viewer are rendered). Figure 4.14 provides an example illustrating our cut and paste tool applied on the super-brain data. The pasted volume can be rotated and translated to reveal the sculpted portion from the original volume. In addition, the cut and paste tool can also be enabled while the sculpting operations are performed. This provides a real-time feedback on the sculpted volume as well as the cut-away portion.

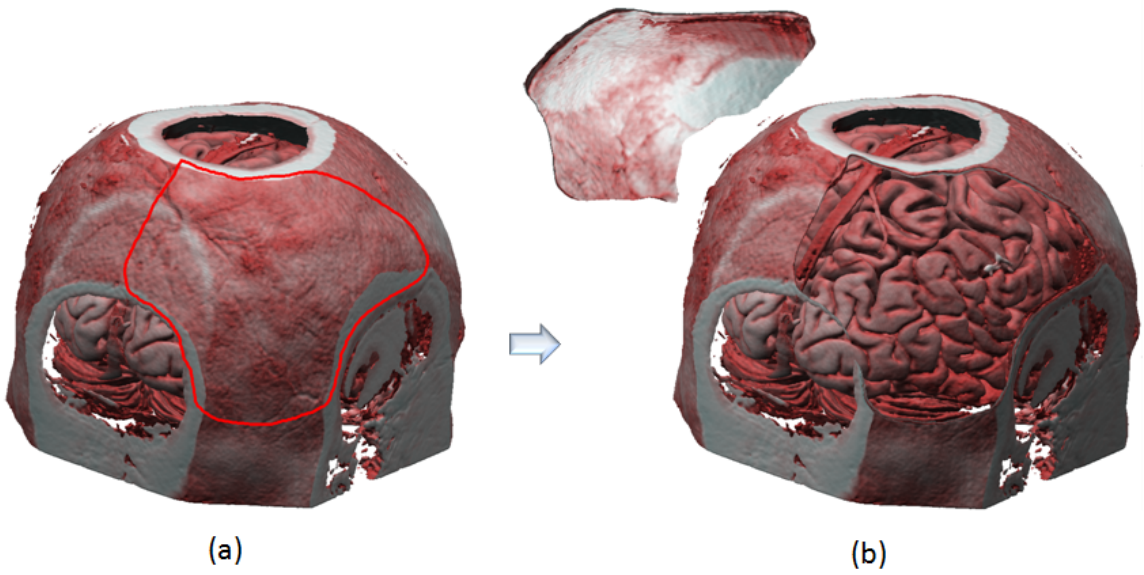

Figure 4.14: Our cut and paste tool applied on the super-brain data. (a) shows the original volume with the peeling sketch. (b) shows the cut region and the removed skull.

4.6 Chapter Summary

In this chapter, we explained the interface problem for directly interacting with a mask sweeping approach. While the concept of sweeping a mask along an arbitrary

trajectory is ideal for sculpting, there was a lack of interactive tools for creation and controlling of this arbitrary sweeping path. Therefore, we proposed a set of intuitive volume tools for mapping the 3D sweeping path problem into an interactive tool framework. We first discussed the framework components, including displaying the data with transfer functions to enhance visualization, constraining the volume tools to the surface of the displayed volume via ray-casting, and adapting a two-pass rendering approach for updating the tool sculpture volume as well as performing the actual rendering operation. We then provided details for each of the volume tools, including the view-dependent drilling tool, the laser tool, the peeling tool, and the cut and paste tool. We also demonstrated examples for each of the volume tools implemented with our GPU-based system.

Chapter 5

Sketch-Based Volume Segmentation

Clinicians and surgeons often use computer-based segmentation to identify and analyze anatomical structures of interest in medical image data sets. In general, volume segmentation is an essential and important step in medical image processing. For example, neuroradiologists often segment and examine the internal carotid artery (Figure 5.1) to determine its degree of stenosis in patients suffering from transient ischemic attacks (TIAs - "mini" strokes). The degree of carotid stenosis is a critical factor to determine if TIA patients should have surgery to open up this vital vessel. Other measurements (such as the shape, topology, and cubic volume) could also be obtained during the segmentation process.

Volume segmentation can be described as a process that analyzes input raw volumes, which may contain a mix of materials, to isolate one or more materials from the rest of the data. In this chapter, we first introduce the problems associated with the segmentation process and discuss the limitations in existing approaches (Section 5.1). In Section 5.2, we provide an overview of our approach with an example demonstrating our proposed interface. In Section 5.3, we discuss the region growing algorithm and the current implementation of this method. In Section 5.4, we introduce our GPU-based techniques for parallel seeded region segmentation and lay out the building blocks.

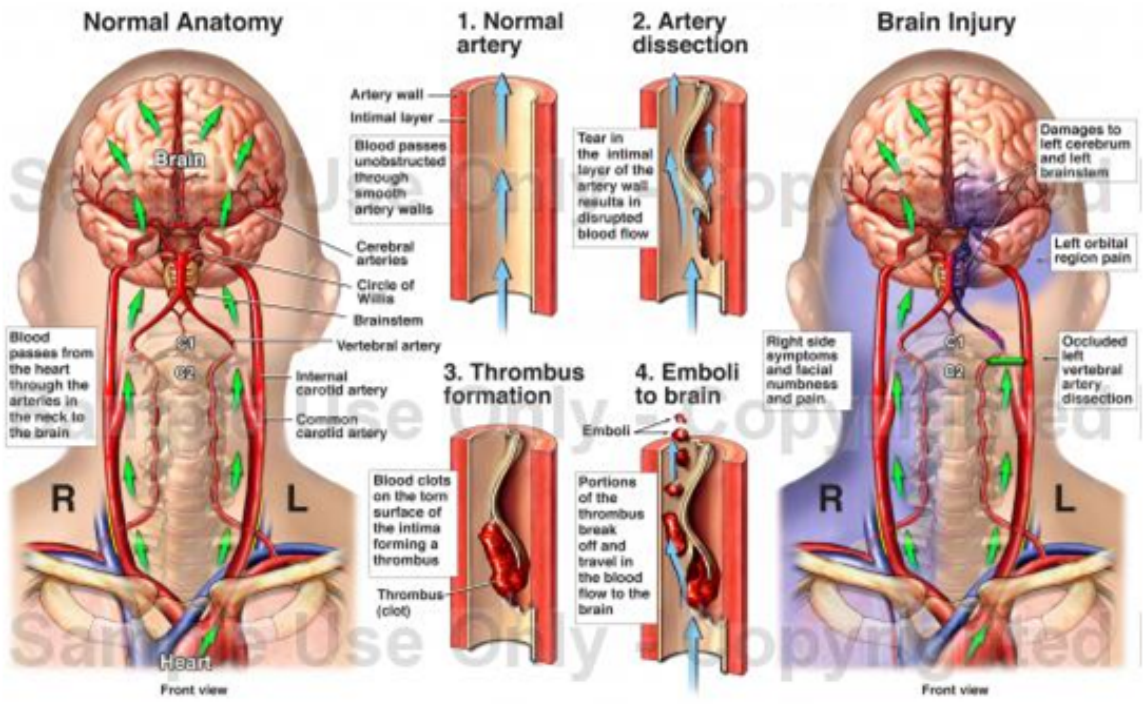

Figure 5.1: Medical illustration of vertebral artery dissection with development of emboli resulting in cerebral infarction. (Copyright ©2007 Nucleus Medical Art. All rights reserved. www.nucleusinc.com)

5.1 Introduction

Segmentation is often broken down into *edge based* or *region based* methods. Each of these in turn may be *manual* or *computer assisted* (including completely automatic). Along the edge-based category, a typical manual segmentation process requires a trained specialist to draw contours around the region of interest (ROI) on cross-sectional images (Figure 5.2). These contour lines are then linked and reconstructed into a 3D representation for further analysis (Figure 5.2, top). This procedure can become a challenging task if the target is, for example, blood vessels in the brain, which by nature involves complex shape with many components and unpredicted

turning directions. Automatic methods currently focus on low-level features such as edge detection and texture analysis. A number of contributions and efforts were made in the research direction for obtaining automatic segmentation results; however as Kirbas and Quek pointed out in [33], all such attempts for developing automatic or semi-automatic segmentation algorithms are limited to some global parameters that can fail with certain data. An example is the semi-automatic generation of transfer functions by Kindlemann and Durkin [32], they analyzed the relationship between different data quantities but were only able to control the segmentation with global histograms.

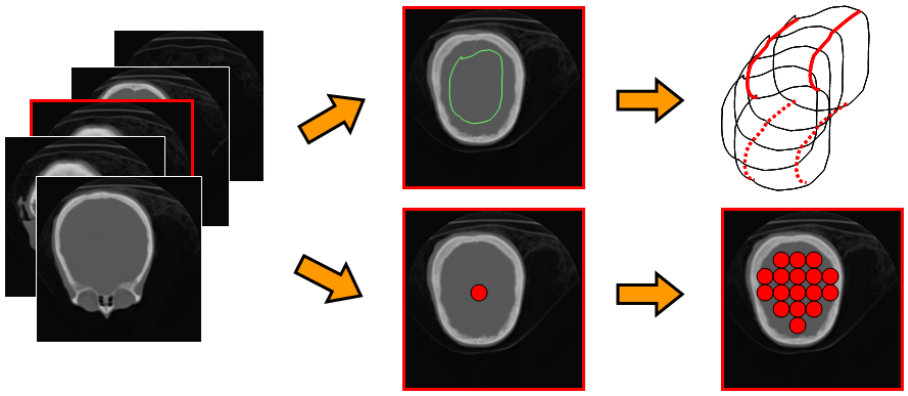

Figure 5.2: Conventional segmentation methods. Top row: edge-based method, which is relatively simple but requires a high contrast border between the target object and the background material. Bottom row: region-based method, which is more difficult in finding an appropriate seed point but works well with data that has pixels with a similar intensity/color in different parts of the image.

The region growing algorithm [46] is one of the well-known region-based segmentation methods that is simple to compute and applicable to a wide range of data types. Adams and Bischof [2] introduced seeded region growing for volume data. Their algorithm requires the planting of an initial seed point in the 3D volume.

However, specifying a 3D coordinate from a 2D device, such as the mouse, is a challenging task. In conventional methods such as proposed by Sherbondy et al. [50], the user navigates from a stack of 2D image slices, selects a desired slice, and places the seed point from the cross-sectional view of the data (Figure 5.2, bottom). As a result the seed point is propagated to the entire volume based on neighboring areas with close or same intensity.

The key limitation with the conventional seeded growing region process involves browsing through the large amount of cross-sectional images. A priori knowledge of the data is required in order to quickly identify the correct slice number and the appropriate seed location on the 2D grey-scaled image. This procedure demands a significant amount of time and does not lend to a natural interaction scheme with the 3D volume (i.e. direct manipulation of the 3D data).

In our previous work [13], we have introduced a sketch-based interface for seeded region growing volume segmentation (Sec. 2.1.4, Fig. 2.1). It provides a straight extrusion path for quick navigation to the region of interest and direct 3D seed planting; however, limitations include lack of parallel seeding, free sweeping path, and dynamic region growing control. In the next section, we describe our approach addressing the above limitations.

5.2 Our Approach

We propose a sketch-based interface for volumetric seeded region segmentation and provide high quality rendering using ray-casting. Figure 5.3 illustrates the key stages of our method applied over a raw MRI dataset (256x256x256). Initially, a volumetric

data set is loaded and intensity range histograms are selected (Figure 5.3, a). Our volume tools (Chapter 4, Sec. 4.3, 4.4, and 4.5) are selected and applied (Figure 5.3, b) to define a region of interest or remove occluding parts of the volume (Figure 5.3, (c)). Next, the user places several sketches directly over the displayed volume to define the seeds (Figure 5.3, d). The region starts to grow (Figure 5.3, e) and finally the computed segmentation is obtained (Figure 5.3, f).

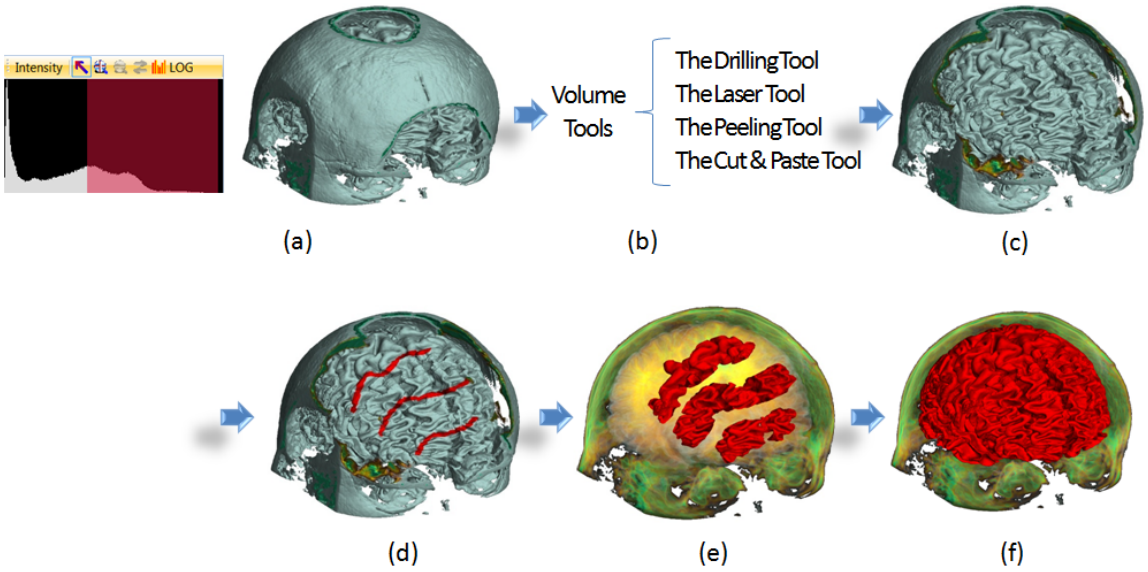

Figure 5.3: Our sketch-based volume segmentation method: user loads the volumetric data and selects an intensity range from the histogram (a); user applies different volume tools (Chapter 4) to sculpt the volume (b) with the results shown in (c); user places several sketches over the displayed volume to define the seeds (d); regions grow (e); and the cerebrum is segmented (f).

Our technique provides sketch-based seed planting directly within a 3D environment. Instead of browsing through hundreds of 2D image slices, the user only needs to provide simple strokes over the displayed volume for segmentation. Due to noise in the image acquisition process, materials with similar intensity values are sometimes disconnected, resulting in an incomplete segmentation of the entire object. We

address this issue by allowing multiple sketches to be specified on different regions. To obtain rapid feedback, we perform parallel region growing on groups of seeds at the same time. As a mix of objects that have close intensities often exist within the volume, false segmentation can occur when the growing threshold is carelessly chosen. In order to adapt to a variety of situations, we allow segmentation results to be dynamically modified through a series of undo, redo, and resume operations. In our approach, seeds are processed as points using the programmable hardware. We maintain multiple seeds by storing their state information in a separate 3D buffer. We utilize point radiation to create an anti-aliased seed map and render the region growing result with high-quality ray-casting.

5.3 Seeded Region Growing

In the seeded region growing algorithm, we start from the selected seed point (the parent seed) as the current voxel and move to adjacent voxels (the child seeds) with feature values (such as intensity or gradient magnitude) close to the values in the current voxel. We utilize the breath-first search approach as appears in the context of graph traversing techniques [16]. This approach helps to maintain a balanced and coherent growth because only the nearest voxels are examined and advanced in each step.

5.3.1 CPU-Based Approach

Standard CPU implementation of the region growing algorithm involves the following steps. At first, the parent seed is placed in a processing queue (Figure5.4, a). Seeds

are drawn from the queue in a FIFO order and expanded to the surrounding region that has not yet been segmented. During the expansion process, a valid move from the parent seed to the unsegmented neighboring location is determined by comparing the feature values of the respective voxels against a pre-defined threshold. If the difference of the feature values falls within the pre-defined range, the target voxel, or the child seed, is marked as segmented and appended to the processing queue for further growth consideration (Figure5.4, b). The set of voxels that were marked as segmented in the last stage now becomes the parent seeds in the next iteration. The growth continues until no more advancement is permitted or when the queue becomes empty (Figure5.4, c). At last, we terminate the region growing process and report the final segmentation result. In order to render the result with high quality ray-casting techniques, an additional storage matching the input data size is required to represent the map of the segmented portion. Figure 5.4 demonstrates an example of using a 2D grid as the segmentation map that indicates the segmented (in orange) and unsegmented (no color) region.

The CPU-based implementation has several limitations in the context of high quality ray-casting. (1) When real-time rendering is required, the segmentation result should be refreshed on the GPU in a timely fashion. However, when the update involves a large volume, the amount of data that has to be transferred over the system bus (from CPU to GPU) can become a bottleneck (Figure 5.5). Hence, the transfer is usually done at the end of the region growing algorithm; and as a consequence, the user has to wait for the entire growing process to finish until the next action could be initiated (i.e. re-seed, adjust thresholds, and etc.). This hinders the ability to interactively modify and update the segmentation process.

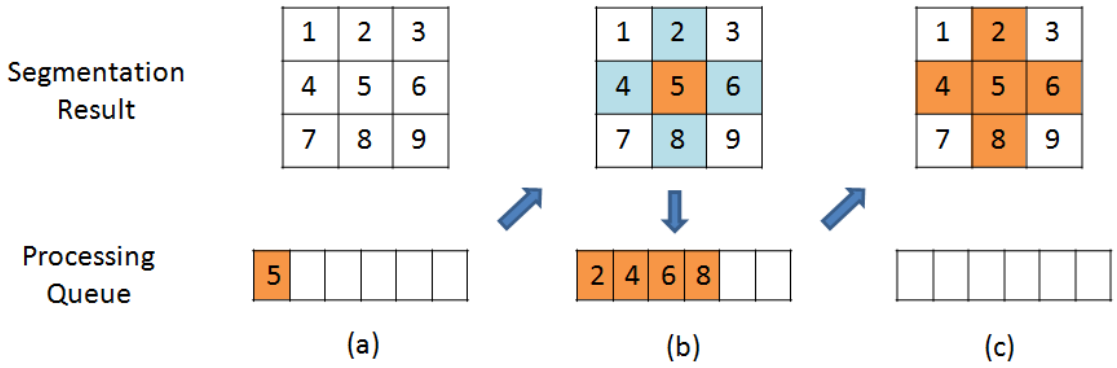

Figure 5.4: Example showing a 2D region growing process implemented on the CPU. The segmentation result (top row) is stored as a binary grid, where the colored pixels indicate the potential or segmented region. In the beginning (a), a seed located in cell 5 is placed in the processing queue (bottom row). The seed (in orange) is dequeued and marked in the segmentation grid (b); and then, the four neighboring pixels (in blue, indicating the potential growth) are validated and enqueued. Next, the items (in orange) in the processing queue (b) are processed sequentially by marking the corresponding cells in the segmentation result (c). The queue becomes empty in the last stage, and the segmentation process is then terminated.

(2) For sequential processing a large input volume on the CPU, the region growing computation can become a lengthy process; for example, a 512^3 volume requires at most $512 \times 512 \times 512 \times 6 = 805,306,368$ voxels to be compared with neighboring elements; hence, the user cannot interactively control the segmentation process due to a lack of real-time feedback.

5.4 GPU-Based Segmentation Framework

Recent advancement of programmable graphics hardware enables parallel execution of general algorithms on the GPU, which brings the potential of accelerating the region growing process in a factor of 128 (i.e. the number of stream processors on GeForce 8800 GTX). However, the mapping of segmentation operations on the GPU

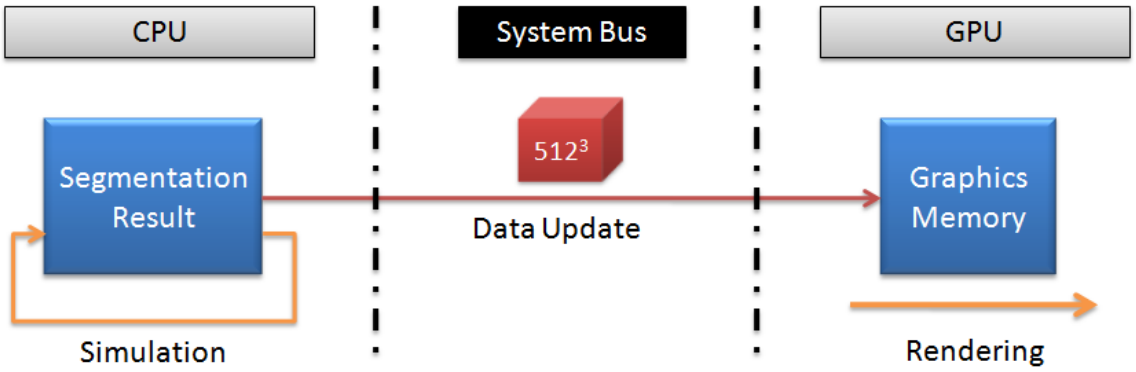

Figure 5.5: Data update scheme for the CPU-based region growing implementation. The region growing simulation is performed entirely on the CPU with the result cached in a system buffer. Once the simulation is completed, the segmentation result (i.e. a large volume) is transferred to the graphics memory for rendering on the GPU.

is a difficult task due to the parallel execution environment. In the GPU architecture, the parallel memory access prohibits algorithms that rely on the order of execution, such as the region growing method. In this algorithm, a single seed must finish with its exploration operations before another seed can proceed. This is because of the advancement in the first operation must not re-occur in the second; or equivalently, the same location (or voxel) cannot be visited twice.

In this work, we leverage geometry shader and multi-pass GPU processing to achieve interactive segmentation operations that include sketch-based seeding, parallel region growing, parallel region shrinking, and dynamic threshold modification. In addition, we take advantage of our volume tools to constrain the segmentation to be embraced by the sculpted sub-volume as a rough estimate of the desired region. This eliminates the growing process from being sensitive to uncertain boundary conditions between the variety of materials in the volume. Furthermore, we utilize the point radiation technique and provide high quality rendition in the context of

ray-casting.

5.4.1 System Architecture

Figure 5.6 provides an overview of our GPU-based architecture for seeded region segmentation. Our system contains mainly four functions: *Sketch Seeds*, *Grow Region*, *Shrink Region*, and *Recover Seeds*. Each function takes a set of input seeds to produce output seeds with a self-feedback loop. The *Sketch Seeds* function converts the user strokes into a set of master seeds which are used to initiate or add into the segmentation process (Section 5.4.2). The *Grow Region* function expands the set of parent seeds to produce child seeds by applying a parallel region growing algorithm (Section 5.4.3). The *Shrink Region* function reverses the region growing steps by starting from the child seeds and searching for the parent seeds (Section 5.4.4). When the growing constraints are modified, the *Recover Seeds* function allows previously suspended seeds due to blocking conditions (i.e. above thresholds) to be reinstated (Section 5.4.5).

To achieve real-time operations, functions are evaluated on the GPU with multi-pass processing techniques. Each function operation starts by feeding a source seed collection (1), exchanging seed progression information with the seed map (2), and writing the result into the destination seed collection (3). The seed map is used to control the segmentation process by mapping the seeding statuses (such as master, parent, child, suspended, or reinstated seeds) into respective voxels. We store seeds in vertex buffers and manage communications with the seed map using the geometry shader, with its unique capability of reading/writing 3D seed points into any location in the volume. The result of each function operation is recorded in a segmentation

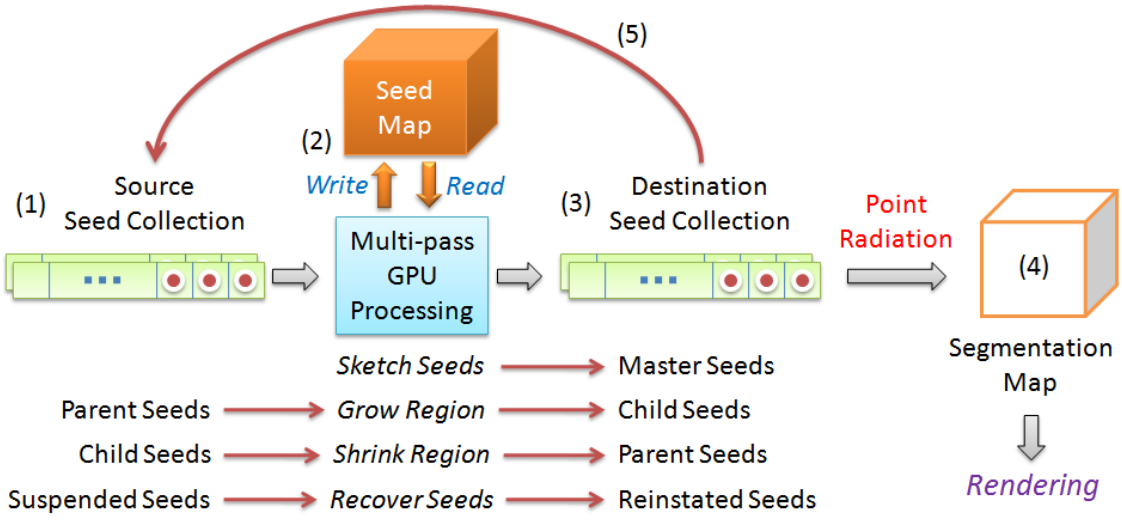

Figure 5.6: Architecture overview of our GPU-based segmentation system. Essential system functions include *Sketch Seeds*, *Grow Region*, *Shrink Region*, and *Recover Seed*. All functions are governed by a multi-pass GPU processing unit. The processing initiates from the source seed collection (1), enters the processing unit, exchanges seed progression information with the seed volume (2), and outputs the result in the destination seed collection (3). The result is point-radiated into the segmentation volume (4) for smooth rendering. The entire process is repeated (5) by swapping the source and destination seed collection.

map for rendering (4). When storing the segmentation map with a binary grid, the rendering produces aliasing artifacts due to under-sampling of the volume space (Figure 5.7, a). To obtain smooth and high quality rendering of the segmentation result, we utilize the point radiation technique (Chapter 3) to convert the destination seed collection into blended regions. When the result is rendered, we sample the segmentation map using 0.5 as a natural candidate to distinguish between segmented and un-segmented regions. Figure 5.7 shows the removal of visual artifacts by rendering the output seeds with point radiation. When a function completes with simulation and rendering, the entire process can be repeated by feeding the destination seeds of the current function as source seeds in the next functions (5).

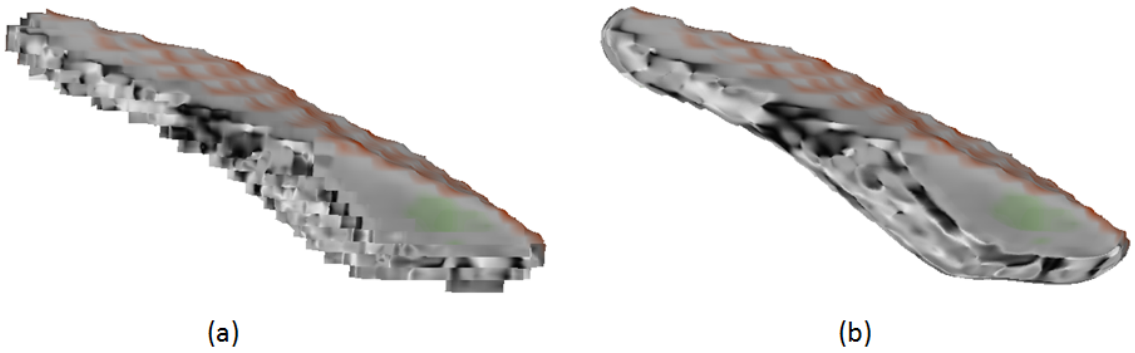

Figure 5.7: Partial segmentation of ventricles (a) using a binary segmentation map without point radiation and (b) with point radiation.

5.4.2 Sketch-based Seeding

To allow quick multi-seeding directly in 3D, we use sketches to indicate seeds on the displayed volume. We map multiple input strokes onto surface points of the volume using ray-casting. The stroke is first discretized and stored in a binary 2D sketch texture (Figure 5.8). Each pixel on the texture is then projected into the volume space to collect surface points based on rendering parameters.

For each point on the surface, we find the closest voxel as the seed. However, as one or more of the detected surface points may be contained within the same voxel due to variance of volume resolution, it is necessary that we remove duplicated voxels to avoid multiplied seed points at the same voxel location. To ensure that we collect unique seed points, we first store the collection of detected voxels in a list (Figure 5.9). Then, for each of the non-repeating voxels in the list, we un-project the respective voxel into a second list, storing the master seeds that map one-to-one to unique voxels in the volume. We maintain lists of voxels using vertex buffers on the GPU. To find the non-repeating voxels, the list of detected voxels is first rendered

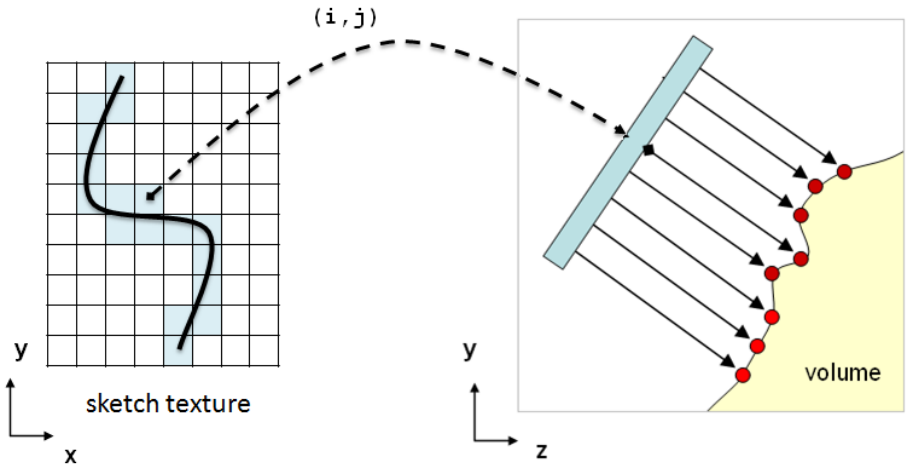

Figure 5.8: Sketching seeds directly over the displayed volume. The input stroke is digitized into a binary sketch texture. For each pixel with coordinate (i, j) on the texture, we project into the volume space and detect surface points.

to an intermediate volume (i.e. the seed map). This allows duplicated voxels to overlap within the same volume space. Then, to obtain the unique seeds, we unproject the rendered voxels by scan-converting the intermediate volume and allowing the geometry shader to write directly into a second vertex buffer, which is used for holding the collection of sketched and non-duplicating seeds.

5.4.3 Parallel Region Growing

To obtain interactive region growing segmentation, we develop algorithms based on the parallel execution environment provided by the GPU. Recent technology indicates that a speed multiplying factor of 128 on GeForce 8800 and 320 on ATI Radeon 3800, with the number rapidly increasing at a much higher rate than the doubling factor of CPU cores. The dramatic speed-up has benefits for providing interactive segmentation operations such as pause, rewind, and resume, allowing

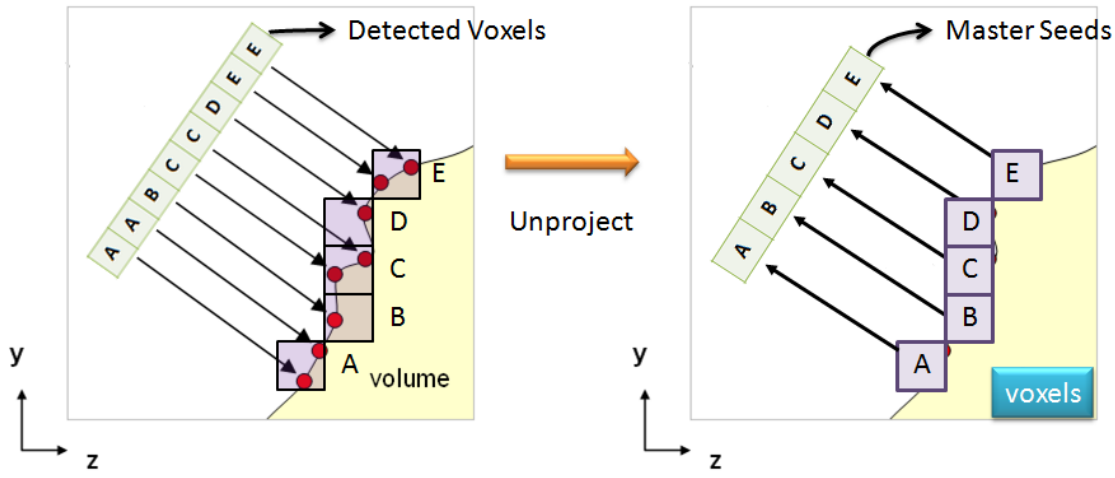

Figure 5.9: To remove duplicated voxels as seed points, a list (left) is first created to store the set of all detected voxels $\{A, A, B, C, C, D, E, E\}$. Each non-repeating voxel in the list is then un-projected into a second list (right) storing the unique set of master seeds.

better control of the segmented regions.

We enable parallel region growing and allow multiple seeds to be explored simultaneously. During the growing process, we start from a set of parent seeds and advance to a set of child seeds. Figure 5.10 shows that the parents seed set $\{P_1, P_2\}$ is concurrently exploring neighborhood conditions and expanding to its respective child seeds, C_1 and C_2. However, a conflict of advancement arises when two or more parent seeds are simultaneously attempting the same voxel location. In Figure 5.10, voxel C' next to both P_1 and P_2 can be a potential child seed grown from either direction. To avoid duplicated child seeds being produced at the same voxel location, we consider C' as the child of the latest parent seed. For this, we just need to allow over-writing the previous values at C'. For the case in Figure 5.10, if P_2 is processed after P_1, we set $C' = C_1$ first and then $C' = C_2$. Hence, C_2 replaces C_1 to become the final value in C'. To determine the most recent child seed produced on the GPU,

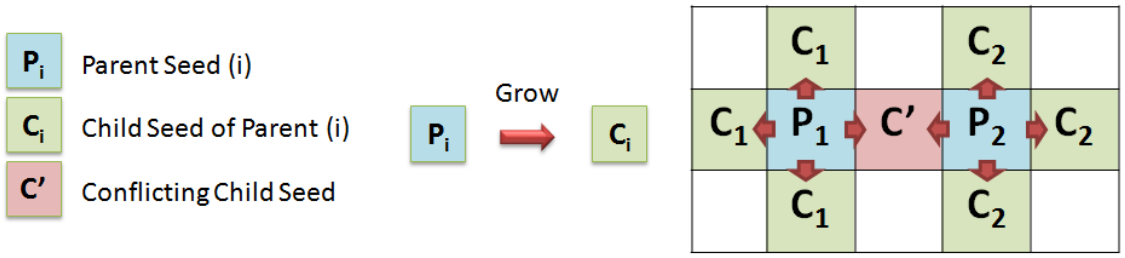

Figure 5.10: Parallel region growing from parent seed set $\{P_1, P_2\}$ to child seed set $\{C_1, C_1, C_1, C', C_2, C_2, C_2\}$. The voxel location C' shows a conflict of potential advancements from parent seed P_1 and P_2 simultaneously.

we first render the parent seed set into an intermediate volume (i.e. the seed map) with directional labels (e.g. L, R, B, A, F, B). And then, we overwrite the directional labels in the intermediate volume and read back the final label to determine a unique direction of where the parent of the conflicting child is originating from. At last, we utilize the geometry shader to collect the set of unique child seeds in a separate vertex buffer.

5.4.4 Parallel Region Shrinking

In a number of region growing situations, different materials such as tissues and blood vessels have close intensity values and cannot be easily controlled by means of thresholding. The user could initially sketch seeds on the blood vessels, but during the growing process, a group of child seeds may start to spread to unrelated materials. In additional to the forward region growing function, we introduce region shrinking that reverses the grown regions and rewinds the segmentation to a previous state. This provides the user an opportunity to undo partial region growing and cease the segmentation process. Afterwards, the user could resume the growing process by

re-sketching seeds on the remaining part of the original intent. Figures 6.6 and 6.7 demonstrate the use of region growing and shrinking on the data sets where there exist close intensities between the blood vessels and the surrounding tissues.

To shrink the grown region, we first erase the segmented voxels occupied by the last set of child seeds, and then we search their neighboring voxels to find potential parent seeds. Figure 5.11 shows that the child seed set $\{C_1, C_2, C_3, C_4\}$ concurrently seeks and advances to the parent seeds (i.e. P' and P_4). However, voxel P' indicates a

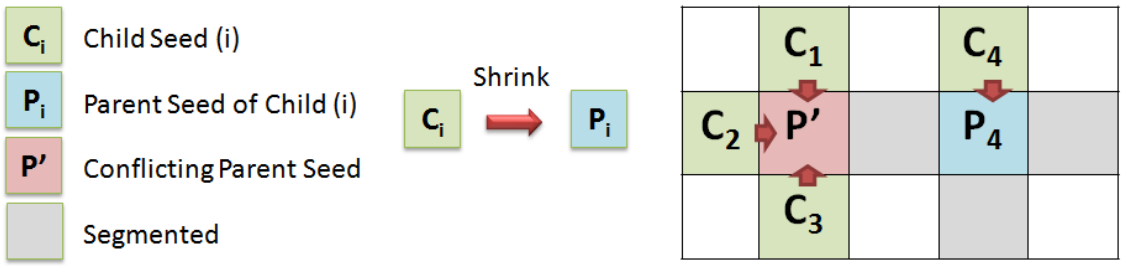

Figure 5.11: Parallel region shrinking from child seed set $\{C_1, C_2, C_3, C_4\}$ to parent seed set $\{P', P_4\}$. The voxel location P' shows a conflict of potential searching by child seed C_1, C_2, and C_3 simultaneously.

conflict of region shrinking where the child seeds C_1, C_2, and C_3 are simultaneously searching at the same location. We use a similar technique as in parallel region growing to resolve the conflict. We consider P' as the parent of the latest child seed by over-writing the previous values at P'. In the case of Figure 5.11, if C_3 is processed after C_1 and C_2, we have $P' = P_3$ as the final value. Similar to parallel region growing, we overwrite the directional labels of the involving child seeds to the seed map and read back the final label to be the direction for the source of child seed. We then utilize the geometry shader to collect the set of unique parent seeds in a separate vertex buffer. This process can be repeated until no more parent seeds

are found.

The reversing process is based on the order in which the child seeds are processed; therefore, it is not guaranteed that we always return to the original set of parent seeds. The reversing algorithm can be seen as a function of absorption of the expanded region, which approximates to the original setting. Another benefit of the reverse operation is that additional strokes can be drawn to indicate the area of removal. In this case, the sketched seeds become the source (i.e. the child seeds) of the reversing operation, allowing targeted region removal.

5.4.5 Dynamic Threshold Modification

In segmentation practices, the region growing process is often constrained by material properties such as intensity or gradient magnitude in addition to the connectivity information in a volume. The imposed constraints allow a better control of locating the region of interest. During the segmentation process, the thresholds (either defined locally or globally) of the extracted properties are frequently adjusted in order to obtain the desired result. When the segmentation thresholds are modified, existing algorithms require the entire growing process to be restarted from the initial seeds that the user specified. In our approach, we enable dynamic threshold modification during the region growing/shrinking process. This allows users to observe the change of material thresholds while the algorithm is running and eliminates the need for a lengthy restart.

In addition to searching for the child seeds in the parallel region growing process, we mark the voxels that could not be advanced (i.e. due to threshold constraints) as *suspended* (Figure 5.12, a). The suspension information is recorded using the seed

map. When thresholds are modified, we lookup the suspended seeds from the seed map and put them into a *reinstated* state (Figure 5.12, b), allowing the suspended seeds to re-participate in the normal region growing/shrinking iterations (Figure 5.12, c). Depending on the current segmentation direction, the reinstated seeds re-enter either the region growing or the region shrinking function.

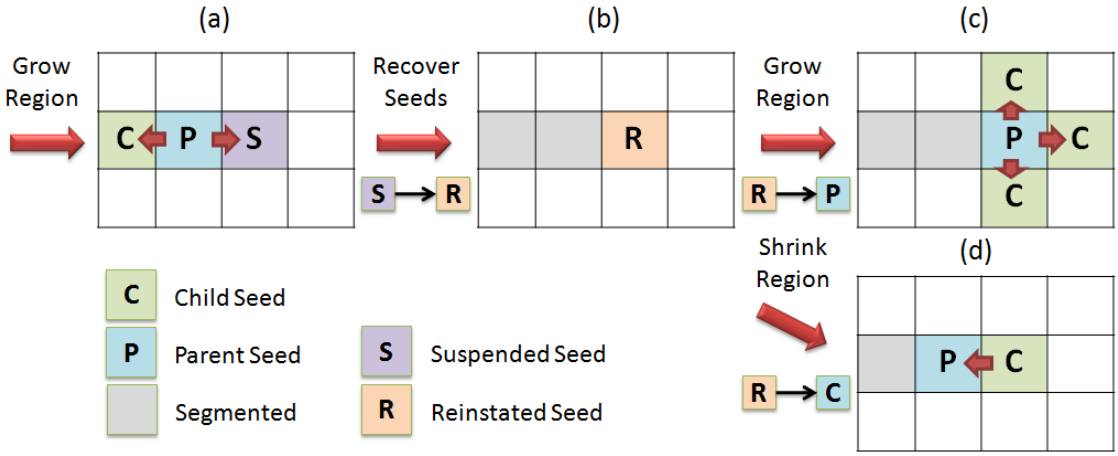

Figure 5.12: Steps for recovering seeds from threshold modification: (a) mark the voxels that are outside the thresholds as suspended in the seed map during the region growing process, (b) search the suspended seeds and update their states as reinstated, and (c) merge the reinstated seeds into existing region growing/shrinking process.

5.5 Chapter Summary

In this chapter, we introduced the common segmentation problem, a frequently encountered problem in medical diagnostic practices. Segmentation is often categorized into edge-based or region-based methods. Along the region-based category, the region growing algorithm is a famous method and has applications in a wide range of volumetric data. Conventional user interface for region growing requires browsing a

large number of 2D cross-sectional images for placing seed points. Our approach provided an evolutionary solution that incorporates a suite of sculpting tools and allows the user to sketch seeds directly in 3D. Our system also supports multiple sketches during the region growing process. We leveraged GPU-programming and achieved real-time processing of parallel seeded region growing. In addition, we provided a number of VCR-like features for fine tuning the segmentation result. The playback options encompassed forward growing, reverse growing, pause, stop, resume, and dynamic threshold control. In our system, we utilized a multi-pass GPU processing scheme to achieve parallel and efficient computation for the following operations: sketching seeds, region growing, region shrinking, and seed recovery.

Chapter 6

Results and Discussion

In this thesis, we combined our volume manipulation tools to enable powerful sculpting and segmentation operations. We created cut-away views that illustrate inner structures of volumetric data. We also segmented blood vessels, bones, teeth, brain, and colon via interactive segmentation tools. All the results were generated on an AMD Anthlon 64 X2 3800, with 4 GB of RAM and a GeForce 8800 GTX, 768 MB graphics card. We selected raw and pre-segmented volumetric data sets of the brain, skull, and abdomen in either CT or MRI modality. Measurements were recorded with a screen resolution of 1230x870. To achieve interactive volume manipulation, we required a factor of five input volumes to be stored on the GPU memory (e.g. data volume, gradient volume, sculpture volume, seed volume, and segmentation volume). Hence, experiments were limited by the available graphics memory. In all experiments, we set the maximum drilling depth to 10, maximum laser mining depth to 20, and the peeling pixel-to-layer conversion factor to 0.1. Table 6.1 provides an overview of our system performance over 10 experiments. We observed that the system performs uniformly in terms of loading time, rendering speed, and the tool interactive rate. All of our results were reported based on our own perception without user studies. For data sets with higher resolution, our system performed slower due to the larger amount of computation required for processing. However, this issue can be compensated with more advanced GPUs as our core technique (i.e. GPU-based point radiation) scales with the amount of parallel (or stream) processors

on board the graphics card. In addition, the speed of the stream processor is also a contributing factor in determining the overall performance of our system.

6.1 Superbrain Experiments

6.1.1 Exploring the Superbrain

Figure 6.1 demonstrates a series of sculpting operations applied on the raw superbrain data set (MRI, 256x256x256). The data was loaded in 0.410 seconds with intensity, gradient magnitude, and 2D histograms computed directly on the GPU. We applied a color transfer function via the intensity histogram and selected a wide intensity range from the histogram. The rendering of MIP, X-ray, and surface mode was provided through our GPU-based raycasting engine. The MIP rendering was measured with 60 Frames Per Second (fps), the X-ray mode with 48 fps, and the surface mode with 62 fps. We first selected the peeling tool to pare off a region on the face, with the peeling operation measured at 38 fps. Next, we utilized the laser tool to remove materials on the left side of the head, with the laser operation measured at 36 fps. Finally, we utilized the view-dependent drilling tool to cut away the upper-right portion of the head, revealing the cerebral regions. The drilling operation was measured at 30 fps.

For the peeling tool, the effect of peeling largely depended on the surface condition and number of layers being peeled. For the laser tool, the drilling direction could become slightly unpredictable when working under a rough surface condition where normal directions change abruptly. For the view-dependent drilling tool, the quality of drilling depended on the density of displayed volume (i.e. quantity of voxels

Table 6.1: System Performance

	Super-brain (6.1)				Skull	Angiogram	Abdomen (6.4)			Head
	Exploring	Ventricles	Opening	Matter	(6.2)	(6.3)	Aorta	Bones	Colon	(6.5)
Load (sec)	0.410	0.417	0.289	0.287	0.472	0.332	1.077	2.536	2.585	0.701
MIP (fps)	60	62	62	60	58	56	36	38	46	58
X-Ray (fps)	48	56	58	58	60	56	34	38	45	56
Surface (fps)	62	58	62	62	60	48	38	40	48	62
Drilling (fps)	30	N/A	N/A	N/A	38	N/A	N/A	24	38	12
Laser (fps)	36	N/A	N/A	N/A	38	N/A	N/A	N/A	N/A	N/A
Peeling (fps)	38	50	50	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Cut-Paste (fps)	42	38	44	60	34	30	22	32	36	38

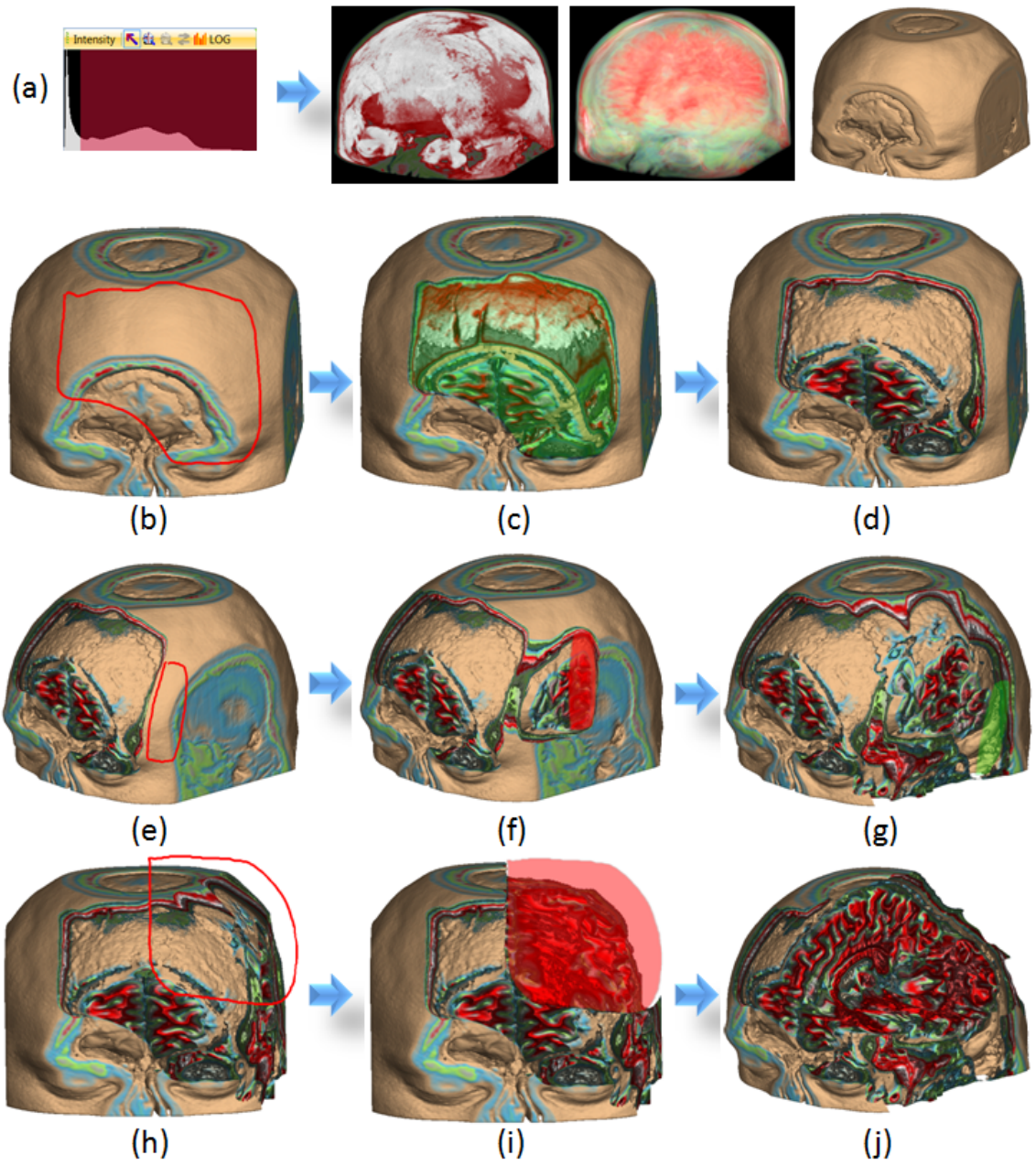

Figure 6.1: Sculpting the raw super-brain data set with the peeling, laser, and drilling tool. (a) Selecting a wide intensity range from the histogram, rendered with MIP, X-ray, and surface mode. (b) Selecting the peeling tool, with sketched-region (c) Removing surface layers with the peeling mask (red overlay). (d) Resulting peel, revealing the underlying materials. (e) Selecting the laser tool, with sketched tool shape. (f) Applying varying pen pressures to sculpt the skull. (g) Resulting cut, removing surface on the side. (h) Selecting the drilling tool, with sketched tool shape. (i) Showing the drilling mask (red overlay), applying pressure for drilling. (j) Resulting sculpture.

selected from the intensity histogram). If the volume was sparse, the drilling tool could create gaps that would be visible after a scene rotation; however with a dense volume, the drilling tool produced no artifacts. Note that the speed of volume tools depended largely on the size and resolution of operating masks.

6.1.2 Segmentation of the Ventricles

Figure 6.2 shows the segmentation of the left and right lateral ventricle from the same super-brain data set. The volume was rendered without using color transfer functions. At first, we selected a low intensity range from the histogram and examined the visibility of ventricles with X-ray rendering (56 fps). Next, we turned to surface rendering (58 fps) and applied the peeling tool to open a small corner on the head, with the peeling mask operating at 50 fps. Until we could clearly observe the hidden ventricles, we placed a stroke on the left lateral ventricle and began with unconstrained region growing. The region grew but extended to other areas. To undo partial region growing, we utilized the reverse growing function and stopped the region shrinking operation when the result contained only the left ventricle. To complete the segmentation, we placed another stroke on the right lateral ventricle. The region grew and stopped without spreading to undesired areas. In the final image, we rendered the segmented lateral ventricles in surface mode and the background in X-ray mode to enhance the contrast.

For the reverse region growing function, it was also possible to incorporate the seed sketching interface to begin region shrinking on an isolated region. For example, the user may accidentally sketched on an undesired region during segmentation and carelessly stopped the entire region growing process. This destroyed most of the

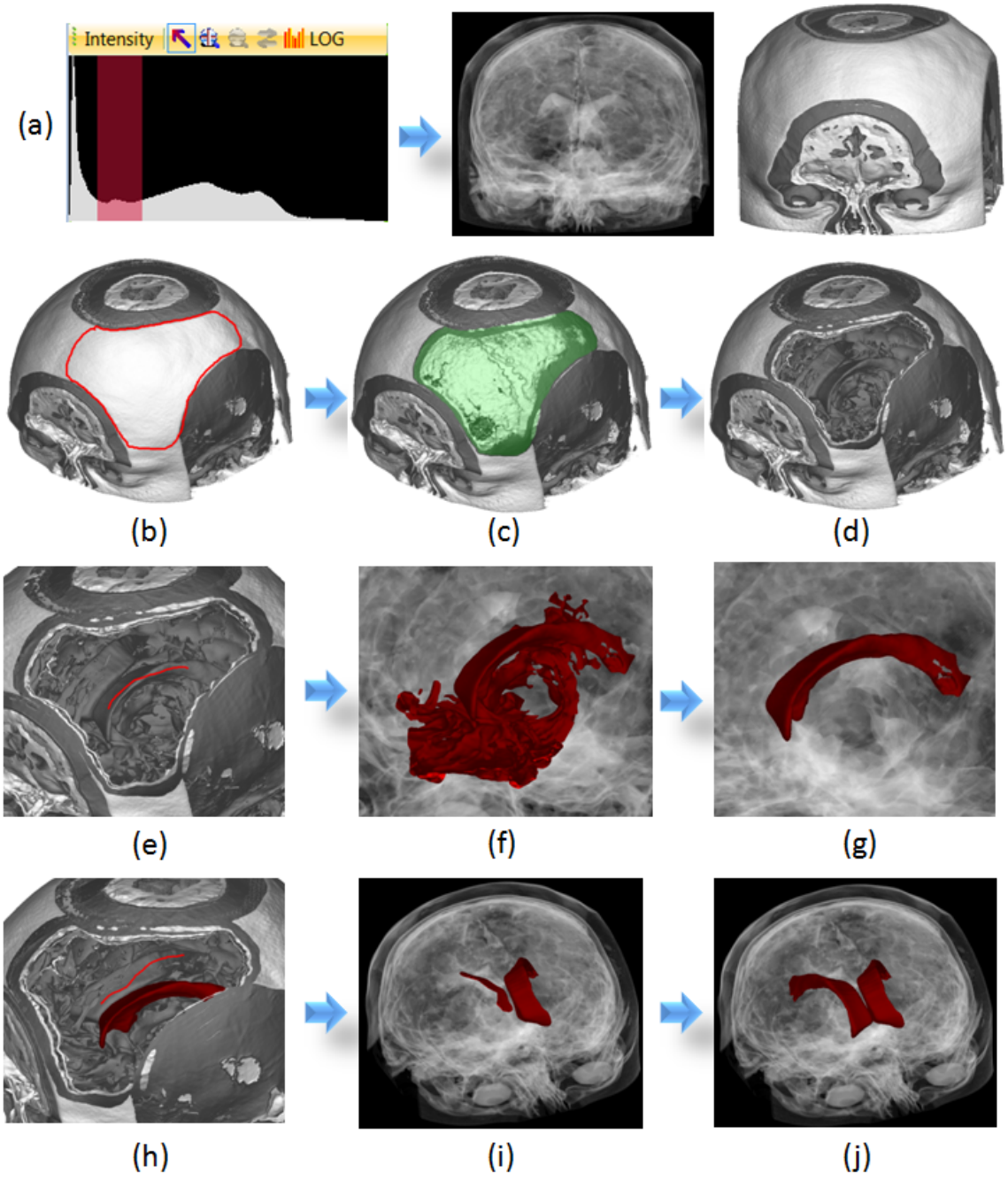

Figure 6.2: Segmentation of the left and right lateral ventricle. (a) Selecting a low intensity range from the histogram, X-ray and surface rendering. (b) Selecting the peeling tool, sketching on the corner of the head. (c) Showing the peeling mask (green overlay), with layers removed from the surface. (d) Resulting peel, creating an opening on the skull. (e) Selecting the segmentation tool, sketching on the left lateral ventricle. (f) Unconstrained region growing that resulted in a spread to other areas. (g) Reversed region growing, followed by forward region growing, constrained by gradient magnitude (lower threshold = -28, upper = 256). (h) Sketching on the right lateral ventricle. (i) Constrained region growing. (j) Region growing stopped, with successfully segmented ventricles.

active or suspended seeds for reverse growing. In this situation, we first turned on the reverse growing function and sketched on the extra region that required removal. As a result, the undesired region would be absorbed by the reverse seeding operation. In short, the reverse growing function could be used both during the region growing process or after a full restart (i.e. by sketching seeds for region shrinking). In this experiment, we did not notice any problems while performing the peeling and the segmentation operation.

6.1.3 Opening the Brain

Figure 6.3 illustrates opening of the brain from the pre-segmented super-brain data set (MRI, down-sampled to 152x154x181). The data was loaded in 0.289 seconds. We first selected the peeling tool and then placed a circular stroke on the top of the head to indicate the opening region. As we dragged down the pen cursor on the screen, layers of skull were removed. The opening illustration was achieved by using the cut and paste tool. The cut-away portion was pasted back to the scene with rotation and translation. The data rendering (surface mode) was measured at 62 fps, and the peeling operation was measured at 50 fps. In the cut and paste illustration, the dual-rendering was measured at 40 fps. In this experiment, we observed that the peeling effect is less desirable in the center of the skull. This was due to the data being cut off on the top portion, and hence failing to provide a smooth surface condition in the central region.

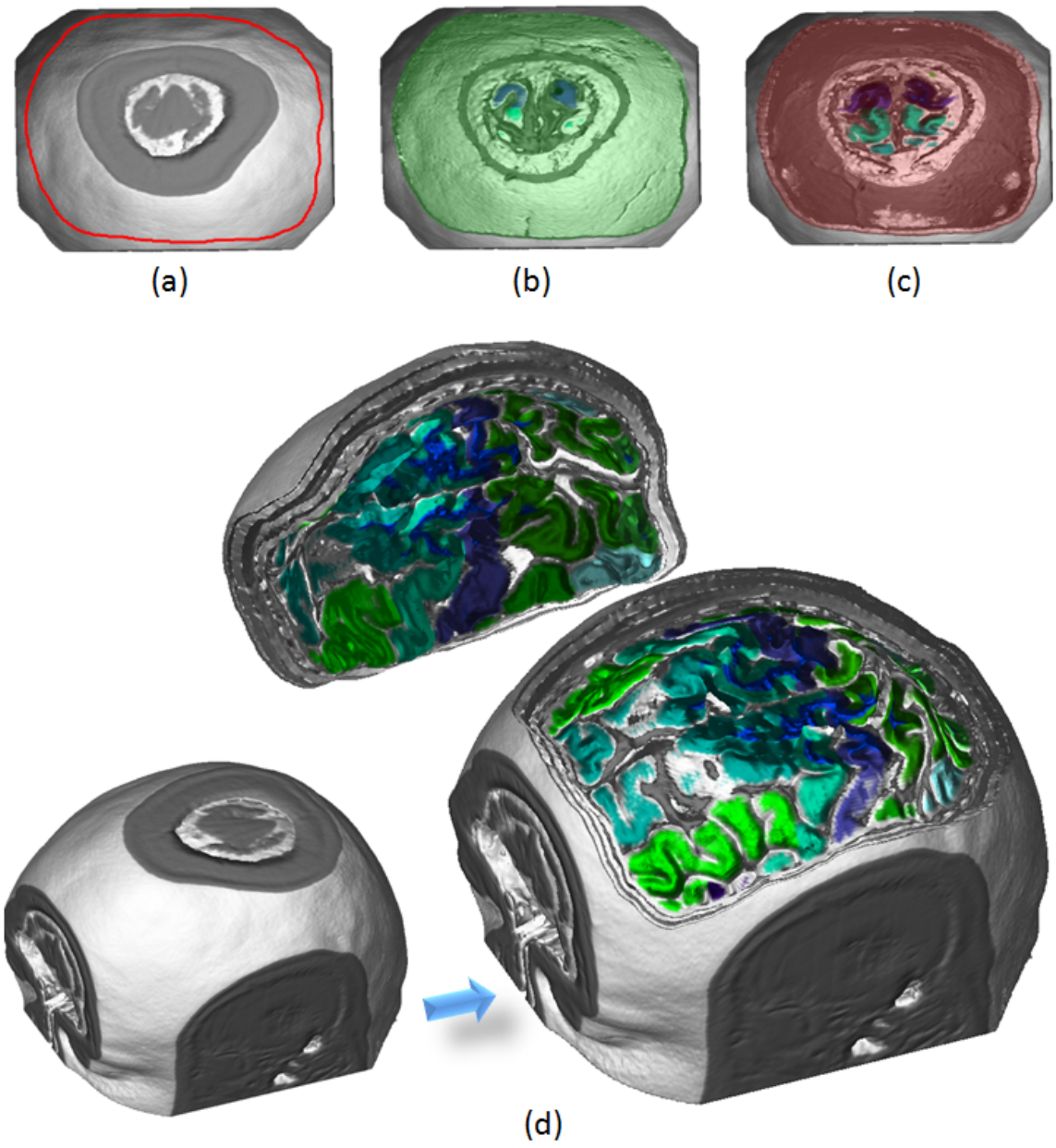

Figure 6.3: Opening the top of the skull on the pre-segmented super-brain data set. (a) Selecting the peeling tool, with sketched region from a transverse view. (b) Showing the peeling mask (green overlay), with layers removed from the surface. (c) Removing more layers from the surface, dragging down the pen on the screen. (d) Using the cut and paste tool to dual-display the cut-away portion of the skull.

6.1.4 Segmentation of the Gray and White Matter

Figure 6.4 depicts the segmentation of the gray and white matter with the same super-brain data set. In this experiment, we selected a high intensity range from the histogram. From the displayed volume, we placed a stroke directly over the visible cerebrum and started the region growing process. The resulting segmentation was zoomed, rotated, translated, and rendered by using the cut and paste tool (60 fps). In this experiment, the region grow was self-contained and no problems were observed during the segmentation process.

6.2 Skull Experiment

In Figure 6.5, we demonstrate the segmentation of a molar tooth from the skull data set (Rotational C-arm X-ray scan of phantom of a human skull, 256x256x256). After loading the data file (0.472 seconds), we selected a medium to high intensity range from the computed histogram. The data was displayed with X-ray (60 fps) and surface (60 fps) rendering, while applying the transfer function via the intensity histogram. At first, we rotated the skull to the left side and zoomed to the target tooth. Next, we drew a small circle on the screen to represent the laser tool tip. We applied varying pressures to remove the area that is occluding the tooth. Then, we turned to the other side and applied the same procedure. When the tooth became visible on both sides, we utilized the drilling tool and erased the surrounding materials. Finally, we placed a stroke that covers most of the tooth to allow a quick segmentation. As a result, the molar tooth was successfully segmented. To provide a better visualization on the segmentation result, we zoomed and isolated the molar

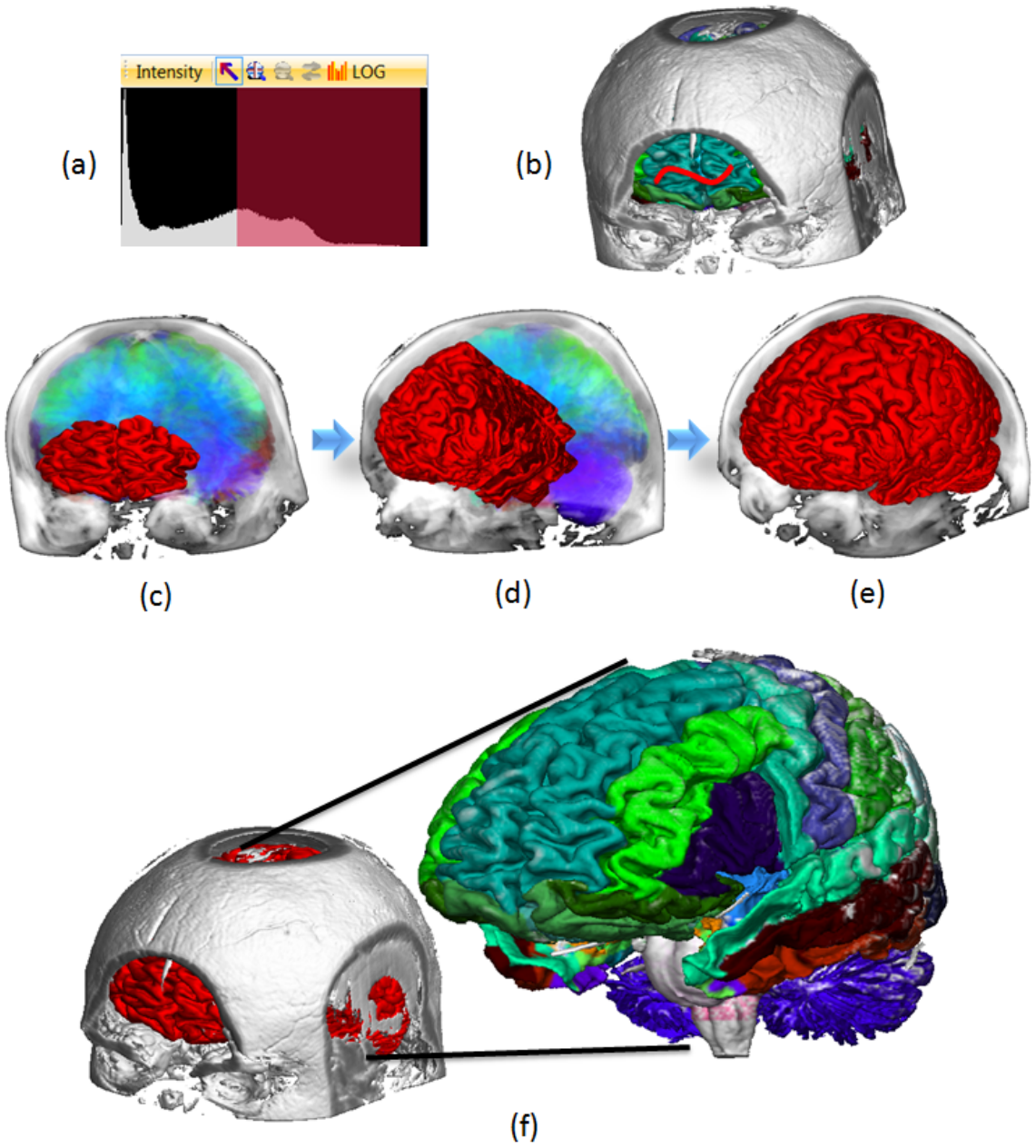

Figure 6.4: Segmenting the gray and white matter from the pre-segmented super-brain data set. (a) Selecting a mid to high intensity range from the histogram. (b) Directly sketching on the brain. (c) Region starts to grow. (d) Region growing, side view. (e) Resulting region grow, angled view. (f) The region in red indicates the grown region embedded within the data set; and the colored brain illustrates a pull-out view of the segmented cerebral regions (with color labels obtained from file).

tooth from the original data with the cut and paste tool (34 fps). In this experiment, we observed that the pressure of the laser tool should be carefully controlled in order to prevent cutting into the molar tooth.

6.3 Angiogram Experiment

Figure 6.6 illustrates the segmentation of carotid and cerebral arteries from the angiography data set (3T MRT Time-of-Flight of a human head, 256x320x128). The data was loaded in 0.332 seconds. Initially, we searched for the arteries in the high intensity range from the histogram and displayed the data in X-ray (56 fps) and MIP (56 fps) mode. When sufficient blood vessels became visible with MIP rendering, we switched to a transverse view and placed a zigzag stroke across the display, in attempting to cover most of the arteries for segmentation. Next, we began unconstrained region growing with the collection of seeds detected from multiple locations. As a problem with unconstrained region growing, one of the growing paths began to spread to unrelated tissues. In order to limit the segmentation to blood vessels, we reversed the growing action and stopped the process until excessive region was removed. In the last image, we utilized the cut and paste tool (30 fps) to separate the segmentation result from the original data. In addition, we zoomed in on the arteries to provide a clearer and more detailed visualization. In this experiment, we experienced a rapid segmentation process with no difficulties.

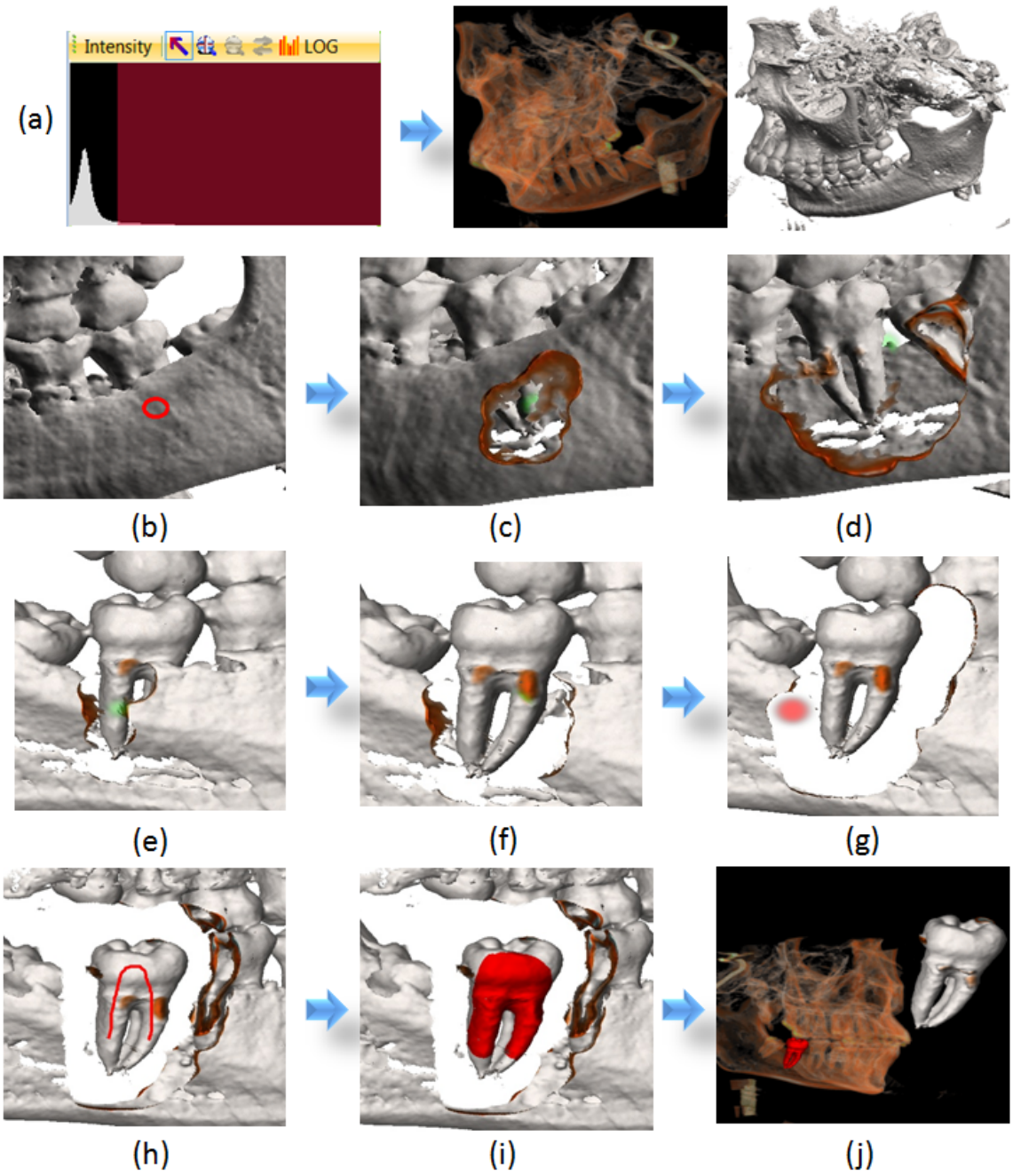

Figure 6.5: Segmentation of a molar tooth from the skull data set. (a) Selecting a high intensity range from the histogram, X-ray and surface rendering. (b) Selecting the laser tool, with sketched tool shape (circle). (c) Gradually removing the surface with varying pressures, showing the peeling mask (green overlay). (d) Resulting cut, revealing the entire tooth. (e, f) Removing occluding portions from the other side with the same laser tool. (g) Removing surrounding materials using the drilling tool, with a circular mask. (h) Selecting the segmentation tool, sketching on the target tooth. (i) Unconstrained region growing. (j) Resulting molar tooth, zoomed and dual-rendered by the cut and paste tool.

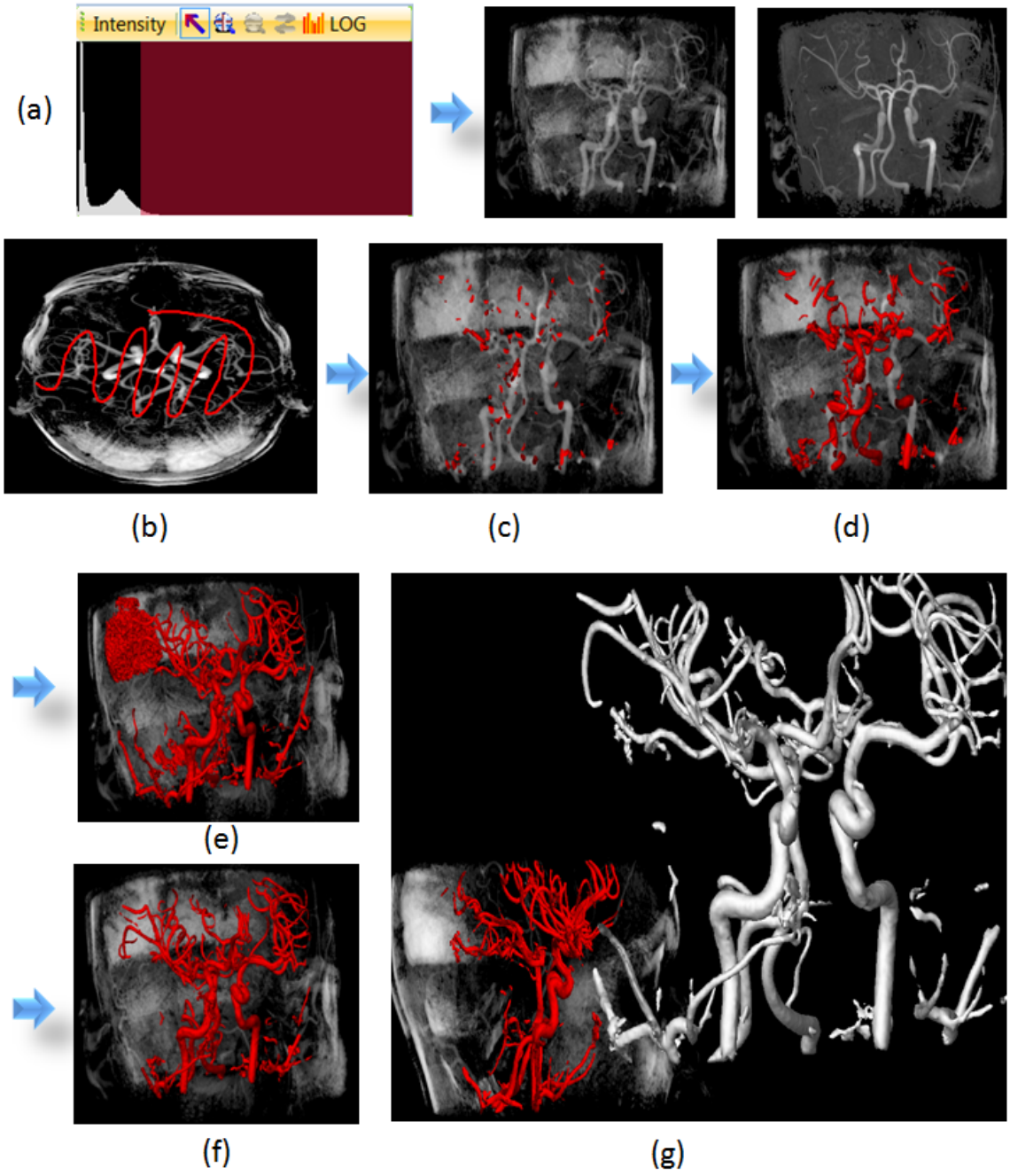

Figure 6.6: Segmentation of carotid and cerebral arteries from the angiogram data set. (a) Selecting a high intensity range from the histogram, with X-ray and MIP rendering. (b) Selecting the segmentation tool, sketching from a transverse view. (c, d) Unconstrained region growing, frontal view. (e) Region growing containing undesired tissues. (f) Reversing the region growing process. (g) Stopping the region growing process, with blood vessels dual-rendered and magnified using the cut and paste tool.

6.4 Abdomen CT Experiments

6.4.1 Segmentation of the Stented Abdominal Aorta

Figure 6.7 depicts the segmentation of the stented abdominal aorta from the stent data set (CT scan of the abdomen and pelvis, 512x512x174). The data loading time was recorded at 1.077 seconds. In the first stage, we chose a high intensity range from the histogram and rendered the data in the surface mode. Next, we placed a short stroke on the stent and began the region growing algorithm. The region grew in both directions along the abdominal aorta; however, one of the growing paths was spread to the pelvis (Figure 6.7, c). We stopped and reversed the growing process until unexpected growing was removed. To continue with segmenting the entire aorta, we placed sketches on different parts of the visible blood vessels. As a result, the growing regions merged and connected the entire abdominal aorta. During the growing process, we utilized the cut and paste tool (22 fps) to simultaneously display the segmentation result on the side. This provided an easier understanding of the current segmentation progress. In this experiment, although a few attempts led to region grows spreading to some undesired regions, but we were able to quickly stop and re-sketch in other areas to continue the process.

6.4.2 Segmentation of the Pelvis, Spine, and the Rib Bones

Figure 6.8 shows the segmentation of the pelvis, spine, and the rib bones from the supine data set (CT scan of abdomen in supine orientation, 512x512x426). The data was loaded in 2.536 seconds. We first selected a high intensity range from the histogram to reveal the bones. And then we rendered the data in X-ray (38

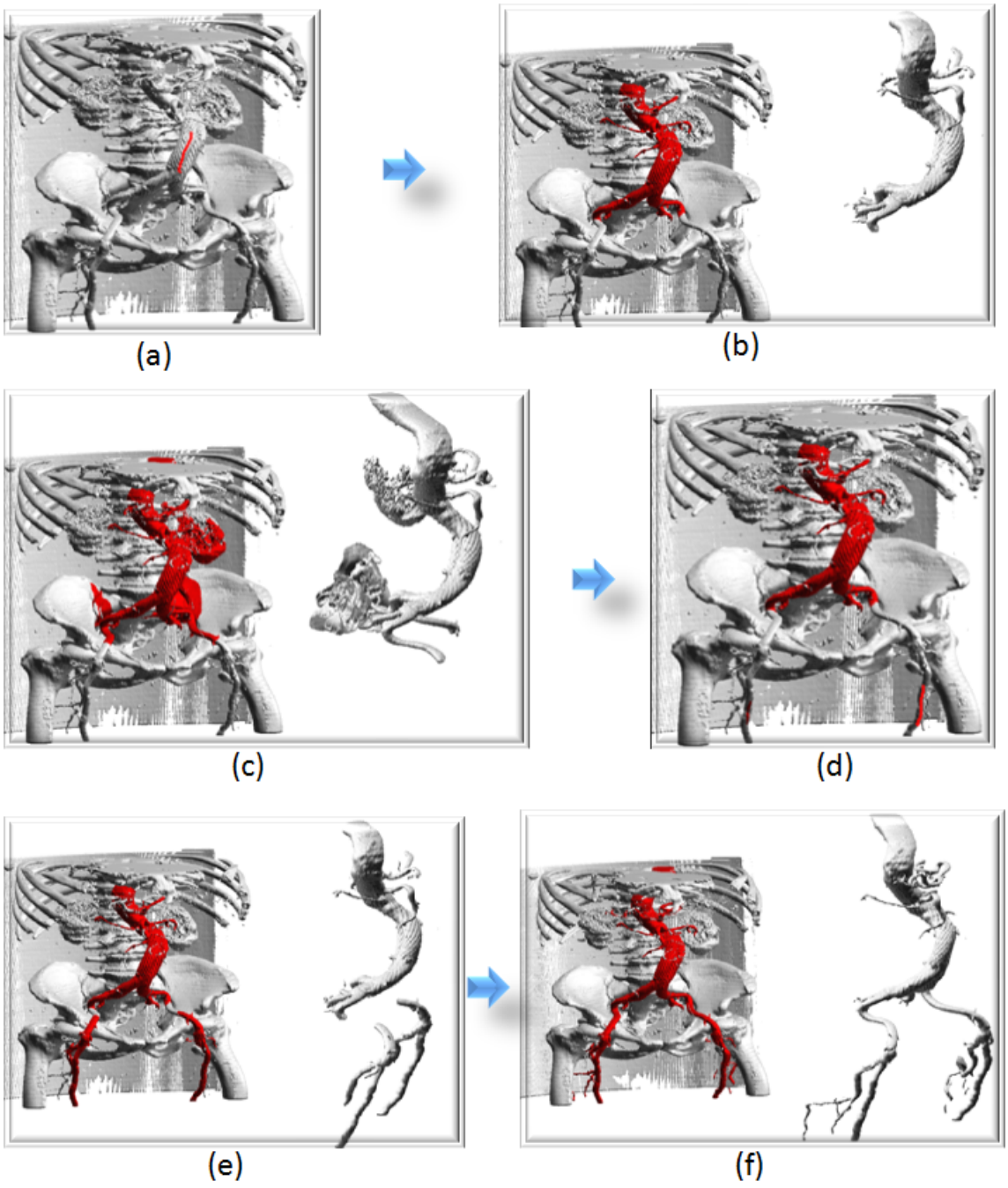

Figure 6.7: Segmentation of the abdominal aorta containing a stent. (a) Selecting the segmentation tool, directly sketching on the stent surface. (b) Unconstrained region growing, with a side view of the segmentation progress utilizing the cut and paste tool. (c) Undesired region growing into the pelvis. (d) Reversing region growing, terminating the growing process. (e) Sketching more seeds on other blood vessels. (f) Resulting region growing combining multiple sketched regions.

fps) and surface (40 fps) mode with a transfer function via the gradient magnitude histogram. To allow a fast region growing segmentation, we placed multiple strokes on the displayed surface, including one on each side of the pelvis and the rib bones. The regions grew without constraint and successfully contained the pelvis, spine, and rib bones. In the last two images, we cut away portions of the original data (i.e. with the view-dependent drilling tool sculpting in the transverse direction, at 24 fps) and pasted the segmentation result using the cut and paste tool (32 fps). Figure 6.8 (h) illustrates an embedded view of the segmentation, and Figure 6.8 (i) depicts an enlarged and divided version. In this experiment, the sketching operations worked well without major difficulties.

6.4.3 Segmentation of the Colon

In Figure 6.9, we demonstrate the segmentation of the colon from the supine data set. We were able to visualize a large portion of colon in an X-ray display (45 fps) with a low intensity range selected from the histogram. In the surface rendering (48 fps), we utilized the peeling tool to create an opening in the abdominal area. We sketched an elliptical shape to form the peeling mask and cut open the abdominal wall so the hidden colon became accessible. Then, we placed several strokes over the segments of the colon and began with a constrained region growing process, with a gradient magnitude range of 20. The regions grew without spreading to other areas, however, only half of the colon was obtained. From choosing a high intensity range in the histogram, we were able to observe the other half of the colon. Hence, we sketched more seeds on the remaining colon segments and utilized the same region growing technique. To reveal the entire colon, we selected a wide intensity range

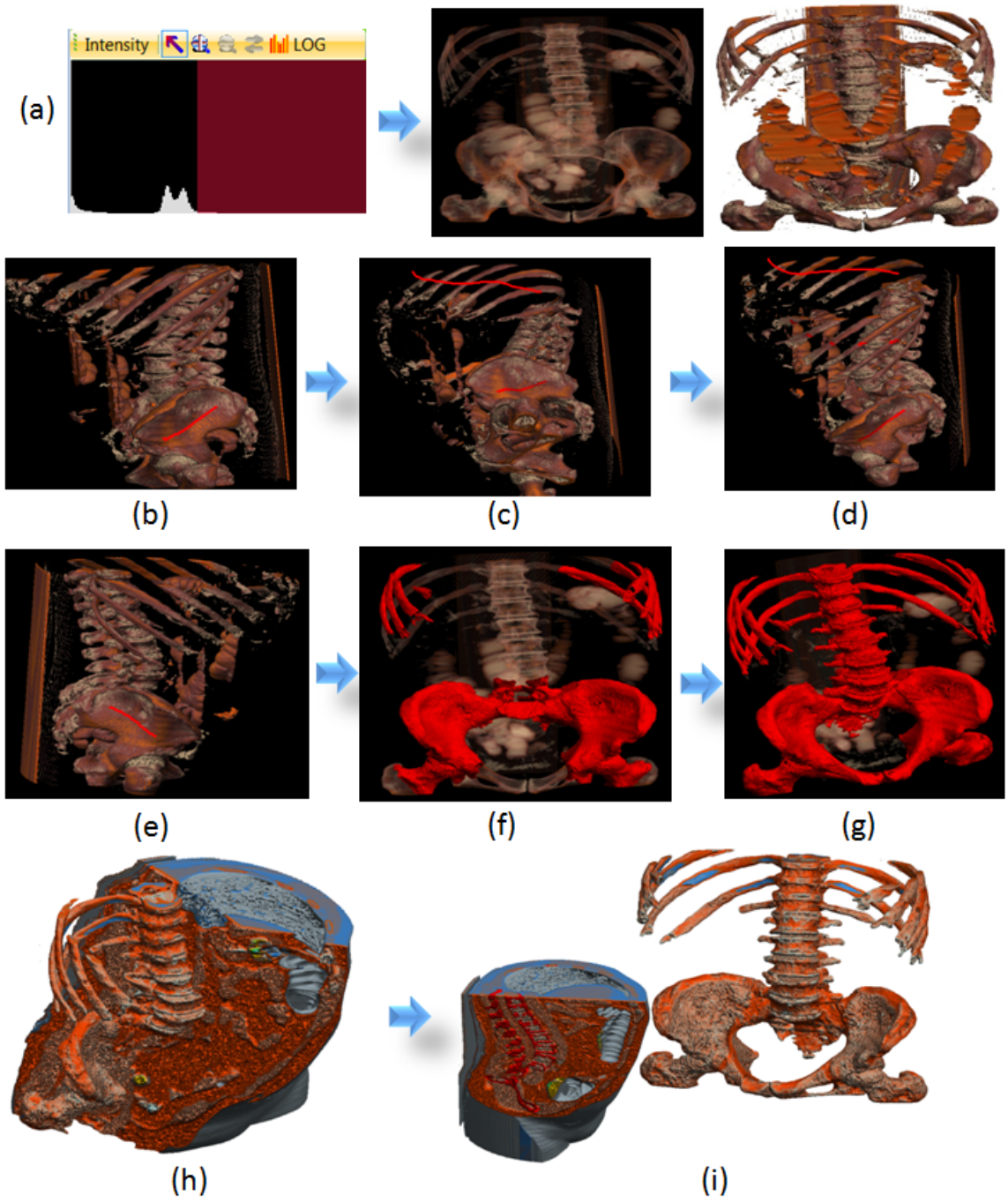

Figure 6.8: Segmentation of the pelvis, spine, and the rib bones from the supine data set. (a) Selecting a high intensity range from the histogram, with X-ray and surface rendering. (b) Selecting the segmentation tool, sketching on the left aspect of pelvis, sagittal view. (c) Rotating the view, sketching across the left sets of rib bones. (d) Rotating the view, sketching across the right sets of rib bones. (e) Rotating the view, placing another sketch on the right aspect of pelvis. (f) Starting the region growing process. (g) Unconstrained region growing. (h) Showing the segmentation result contained within the original data, with removed portion from a drilling operation. (i) Enlarging the segmentation result with the cut and paste tool.

from the histogram. In the final image, we highlighted the segmentation result with the cut and paste tool (36 fps) and incorporated the transfer function via the gradient magnitude histogram. In this experiment, we observed a potential hazard of sketching on the background material rather than the target organ (i.e. the colon). However, we were able to revert the process either by restarting the segmentation procedure or switching to the reverse growing function.

In previous segmentation situations (e.g. Figure 6.8 and Figure 6.9), the user may also wish to begin the region growing process after multiple sketches were laid. In this case, we disabled the automatic region growing start function and waited for all seeds to be detected. And then, we started the region growing operation and launched the parent seeds from multiple locations.

6.5 Head CT Experiment

In Figure 6.10, we illustrate an exploded view of human head from a CT data set (256x256x171). The data was loaded in 1.329 seconds. A medium to high intensity range was selected and rendered in the surface mode (62 fps). We exploited the view-dependent drilling tool and performed the drilling operation (12 fps) from a transverse view. We placed a stroke that surrounds the exterior of the head, in attempting to isolate the interior region. Then, we applied the mask to three fourth of the depth. At last, we utilized the cut and paste tool (38 fps) to provide the opening illustration, with a transfer function via the gradient magnitude histogram. In this experiment, we observed a slightly slower drilling operation due to a larger mask.

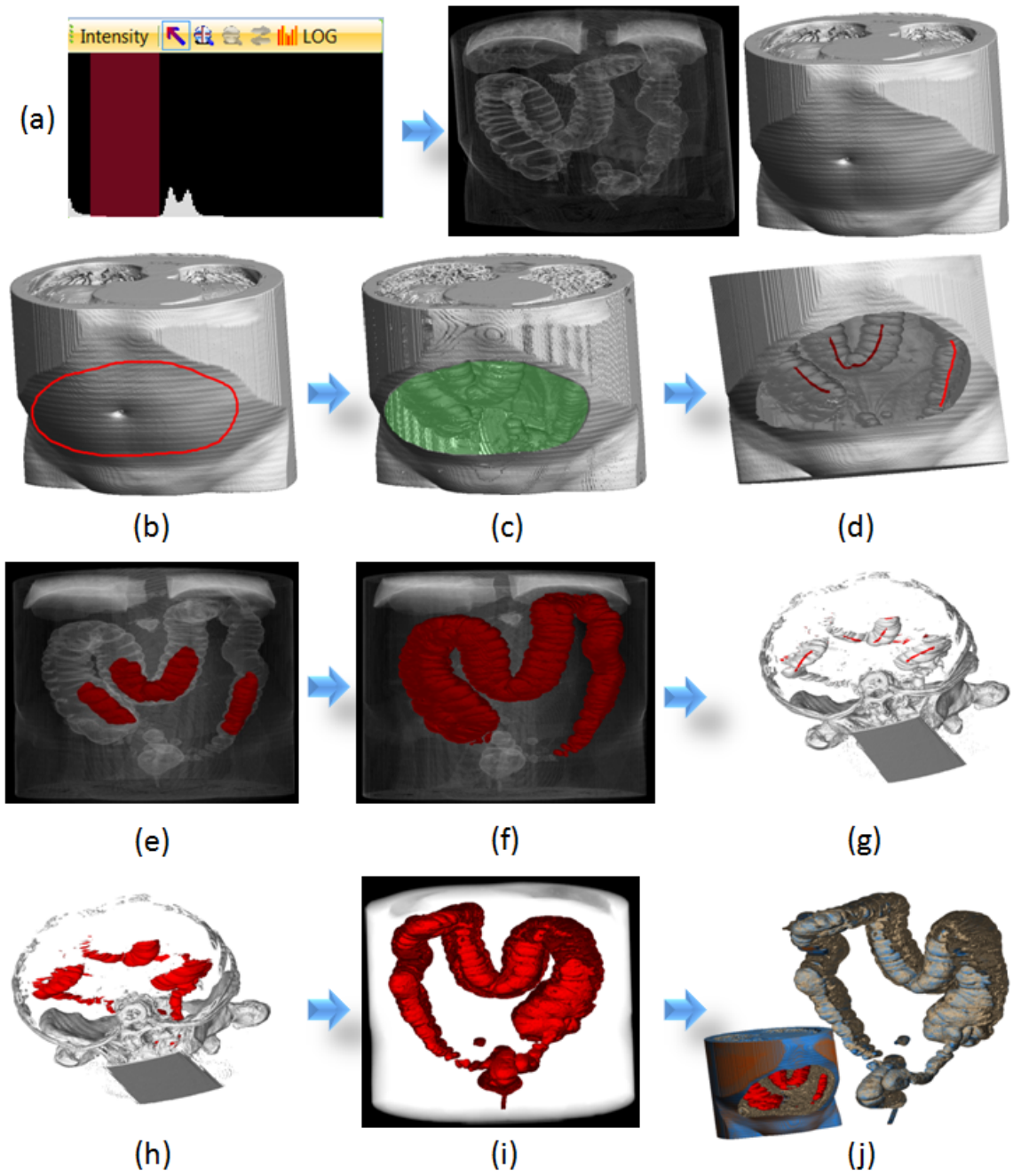

Figure 6.9: Segmentation of the colon from the supine data set. (a) Selecting a low intensity range from the histogram, X-ray and surface rendering. (b) Selecting the peeling tool, with sketched region in the abdominal area. (c) Removing the abdominal wall, showing the peeling mask (green overlay). (d) Selecting the segmentation tool, placing three strokes on the colon. (e) Constrained region growing, with gradient magnitude thresholds (lower = -23, upper = 25). (f) Resulting region growing. (g) Selecting a high intensity range from the histogram, placing multiple strokes on the visible parts of colon. (h) Unconstrained region growing. (i) Selecting a full intensity range from the histogram, revealing the entire colon. (j) Illustrating the segmentation result with a pull-out view using the cut and paste tool, with color enhanced via gradient magnitude transfer function.

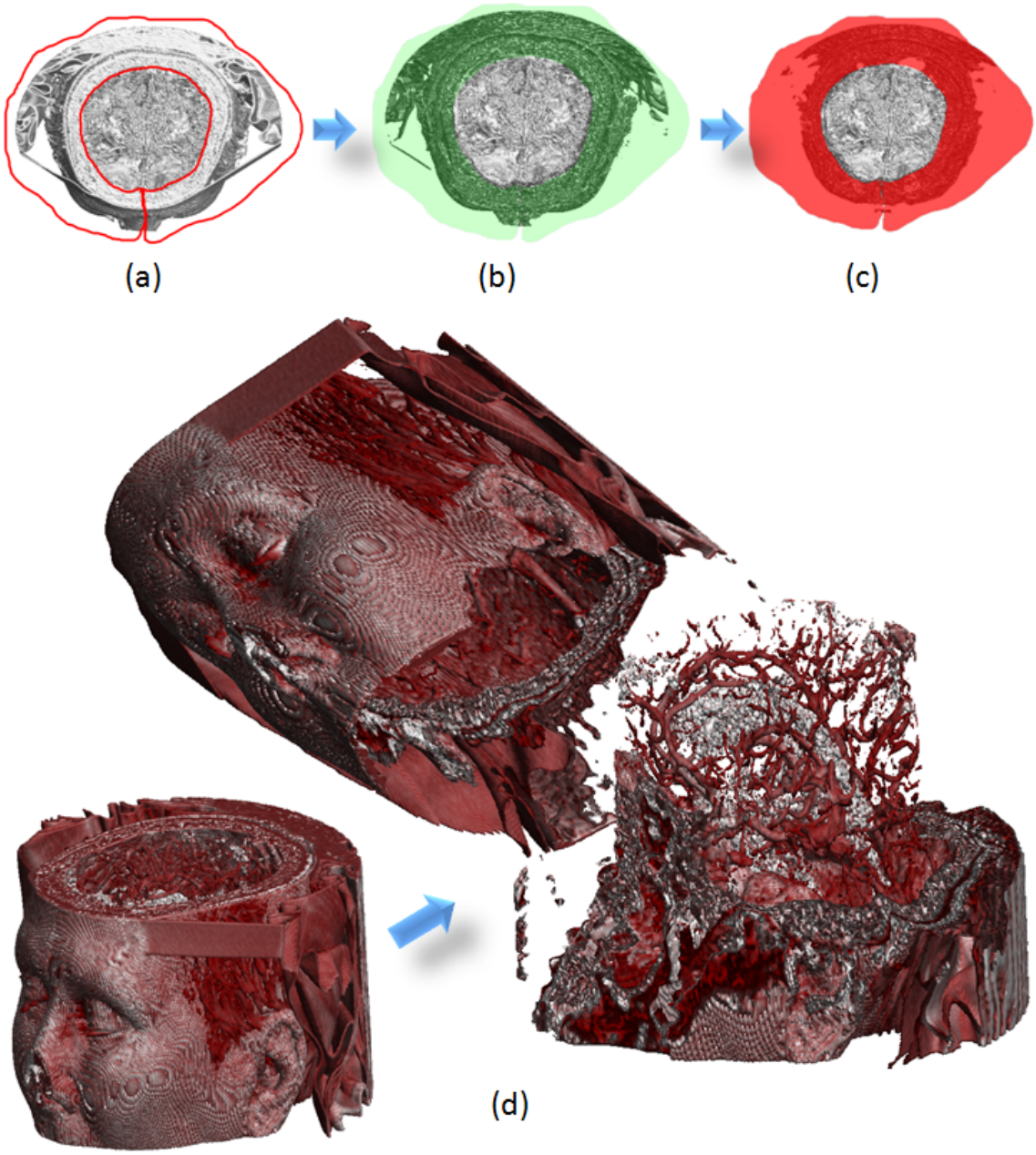

Figure 6.10: Exploded view of a CT data set. (a) Selecting a wide intensity range from the histogram, selecting the drilling tool, sketching a tube shape enclosing the outer region of the data set, transverse view. (b, c) Drilling in the transverse direction, showing the drilling mask (green - no pressure applied, red - pressure applied). (d) Illustrating the original data set as well as the opening view using the cut and paste tool.

Chapter 7

Conclusion and Future Work

In this thesis, we presented a framework of tools for volume manipulation. Instead of the traditional way of browsing from hundreds of cross-sectional slices or adapting 3D devices, we proposed novel interaction techniques for volume sculpting, cutting, and segmentation. We also achieved real-time volume exploration and navigation with a minimum of 128-times performance improvement over conventional methods. In order to handle the large amount of information in medical volume data, we exploited programmable hardware for fast data loading, histogram computations, ray-casting, point-radiation, volume tool creation, and dynamic 3D segmentation.

Our point radiation technique (Chapter 3) was inspired by 3D splatting, one of the pioneering works in volume sculpting. In comparison to the original approach, we provided a vast improvement and achieved real-time interaction on large volumetric data sets. As discussed in Chapter 3, we extended the 2D version of point sprite and introduced the 3D point sprite from exploiting the latest graphics feature—the geometry shader. Being inspired by the concept of swept volume, we further extended the point radiation technique to build mask radiation, which allowed a mask to be swept along an arbitrary trajectory based on a forward splatting approach. When compared to the traditional way of handling swept volumes (i.e. projection-based), we were able to provide global anti-aliasing effects without any post-filtering steps.

We proposed a collection of sketch-based volume tools for volume sculpting and for obtaining the volume of interest in Chapter 4. The tools included the view-

dependent drilling tool, the laser tool, the peeling tool, and the cut and paste tool. The view-dependent drilling tool allowed the user to sketch a drilling mask and use it to cut through the volume with the current view. The laser tool attached to the visible surface and performed a normal-dependent drilling operation based on a user-defined mask. The peeling tool used the user-sketched mask and allowed independent points to cut and radiate along the respective surface normals. Finally, the cut and paste tool provided a dual-view of the original data as well as the cut portion.

We proposed interactive tools (sketch-based and playback buttons) for volumetric seeded region growing in Chapter 5. With the GPU-based surface-detection strategy, we were able to collect multiple seed points directly on the displayed surface. We also allowed multiple sketches to be laid during the region growing process. In addition, we introduced a reverse growing technique that enables a false segmentation area to be reverted. During the region growing process, we saved what we called the *suspended seeds* that could be revoked after an update of the thresholding criteria. We also provided a set of VCR-like playback functions and achieved interactive rates of processing with the geometry shader.

And finally, we provided a number of experiments and measured the rates of interaction in Chapter 6. In the experiments, we tested with large medical data sets including the super-brain, skull, angiogram, abdomen CT, and the head CT. In the results, we demonstrated the usefulness of our volume tools in combination with a number of interactive segmentation techniques.

Future improvements include extending our system to create more sketch-based and interactive volume tools, such as painting tool, filling tool, and surface incision tool. The painting tool would allow marking on the visible surface with different col-

ors, similar to how surgeons would mark on the skin to indicate the opening section. The filling tool would allow adding materials to the existing volume of interest, analogous to a dentist filling the hole on a tooth. The surface incision tool would enable a surgical-like cutting, and the result could be combined with interactive volume deformation to simulate the opening effect. It would also be useful to modify the point radiation technique to include other shapes of 3D footprints; examples include a possible extension of the elliptical weighted averaging (EWA) technique [62] to 3D. This could create a non-spherical radiation in space and possibly generate a variety of other cutting or sculpting primitives. Another possibility is to create a cube-like cutting primitive based on point-radiation to generate sharp edges for the cutting tools. The bottom line is that one could potentially utilize the point-radiation strategy to create arbitrary sculpting primitive shapes directly on the GPU. Our point radiation technique could also be adapted in other sources of applications, such as physically-based simulation and real-time volume deformation. The movement of materials could be achieved by subtracting and adding 3D footprints in the volume space.

The criteria that we used to judge the quality of the results were solely based on our observations on the speed and flexibility of volume data cutting, exploration, and seed planting/growing control. It is important to conduct more formal evaluations and user/clinical studies to provide quality sketch-based volume manipulation tools for professionals in medical science.

Bibliography

[1] K. Abdel-malek, D. Blackmore, and K. Joy. Swept volumes: Foundations, perspectives, and applications. *International Journal of Shape Modeling*, October 08 2000.

[2] R. Adams and L. Bischof. Seeded region growing. *IEEE Trans. on PAMI*, 16(6):641 – 647, June 1994.

[3] R. S. Avila and L. M. Sobierajskim. A haptic interaction method for volume visualization. In *IEEE Visualization*, pages 197–204, 1996.

[4] J. Bloomenthal and B. Wyvill. Interactive techniques for implicit modeling. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 109–116, New York, NY, USA, 1990. ACM.

[5] J. Bloomenthal and B. Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[6] D. Blythe. The direct3d 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.

[7] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Proc. of the Eurographics Symposium on Point-Based Graphics '05*, 2005.

[8] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern GPUs. In *Pacific Conference on Computer Graphics and Applications*, page

335. IEEE Computer Society, 2003.

[9] M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. In M. Gross, H. Pfister, M. Alexa, and S. Rusinkiewicz, editors, *Symposium on Point-Based Graphics*, pages 25–32, Zürich, Switzerland, 2004. Eurographics Association.

[10] S. Bruckner and M. E. Gröller. Volumeshop: An interactive system for direct volume illustration. In *Proc. of IEEE Visualization*, pages 671–678, 2005.

[11] I. Buck, K. Fatahalian, and P. Hanrahan. GPUbench: evaluating gpu performance for numerical and scientifc application. In *Proceedings of the ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–20, Los Angeles, California, 2004.

[12] H. Chen and H. Sun. Real-time haptic sculpting in virtual volume space. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 81–88, New York, NY, USA, 2002. ACM Press.

[13] H. L. J. Chen, F. F. Samavati, M. C. Sousa, and J. R. Mitchell. Sketch-based volumetric seeded region growing. In *Proc. of Eurographics Workshop on Sketch-Based Interfaces and Modeling 2006*, pages 123 – 129, 2006.

[14] L. Coconu and H.-C. Hege. Hardware-accelerated point-based rendering of complex scenes. In Simon Gibson and Paul Debevec, editors, *Proceedings of the 13th Eurographics Workshop on Rendering (RENDERING TECHNIQUES-02)*, pages 43–52, Aire-la-Ville, Switzerland, June 26–28 2002. Eurographics Association.

[15] R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[17] NVIDIA Corporation. Nvidia opengl extension specifications for the geforce 8 series architecture (g8x). November 2006.

[18] NVIDIA Corporation. Technical brief: Microsoft directx 10: The next-generation graphics api. *TB02820001v01*, November 2006.

[19] NVIDIA Corporation. Technical brief: Nvidia geforce 8800 gpu architecture overview. *TB02787001v01*, November 2006.

[20] C. D. Correa, D. Silver, and M. Chen. Feature aligned volume manipulation for illustration and visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1069–1076, 2006.

[21] E. Ferley, M. P. Cani, and J. D. Gascuel. Practical volumetric sculpting. *The Visual Computer*, 16(7):469–480, 2000.

[22] T. A. Galyean and J. F. Hughes. Sculpting: an interactive volumetric modeling technique. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 267–274, New York, NY, USA, 1991. ACM Press.

[23] R. Gonzalez and R. Woods. *Digital Image Processing*. Addison-Wesley Publishing Company, 1992.

[24] K. Gray. *The Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press, 2003.

[25] G. Guennebaud and M. Paulin. Efficient screen space approach for hardware accelerated surfel rendering. In Thomas Ertl, editor, *VMV*, pages 485–493. Aka GmbH, 2003.

[26] A. Hanson and H. Ma. Parallel transport approach to curve framing. *Tech. Rep. 425, Indiana University Computer Science Department*, 1995.

[27] M. Harris and D. Luebke. *SUPERCOMPUTING 2006 Tutorial on GPGPU*. General-Purpose Computation Using Graphics Hardware, 2006.

[28] R. Huff, C. A. Dietrich, L. P. Nedel, C. M. D. S. Freitas, J. L. D. Comba, and S. D. Olabarriaga. Erasing, digging and clipping in volumetric datasets with one or two hands. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 271–278, New York, NY, USA, 2006. ACM Press.

[29] G. Johansson and H. Carr. Accelerating marching cubes with graphics hardware. In Hakan Erdogmus, Eleni Stroulia, and Darlene A. Stewart, editors, *CASCON*, page 378. IBM, 2006.

[30] J. T. Kajiya and B. P. V. Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, 1984.

[31] A. Kaufman and K. Mueller. *Overview of Volume Rendering*. The Visualization Handbook, 2005.

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

[33] C. Kirbas and F. K. H. Quek. Vessel extraction techniques and algorithms: a survey. In *Proc. of Bioinformatics and Bioengineering '03*, pages 238 – 245, 2003.

[34] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

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

[36] I. H. Manssour, L. G. Fernandes, C. M. D. S. Freitas, G. Serra, and T. Nunes. High performance approach for inner structures visualisation in medical data. *IJCAT*, 22(1):23–33, 2005.

[37] M. McGuffin, L. Tancau, and R. Balakrishnan. Using deformations for browsing volumetric data. In *Proc. of IEEE Visualization*, pages 401–408, 2003.

[38] D. P. Mitchell and A. N. Netravali. Reconstruction filters in computer graphics. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 221–228, August 1988.

[39] N. Neophytou and K. Mueller. GPU accelerated image aligned splatting. In Arie E. Kaufman, Klaus Mueller, Eduard Gröller, Dieter W. Fellner, Torsten Möller, and Stephen N. Spencer, editors, *Volume Graphics*, pages 197–205. Eurographics Association, 2005.

[40] N. Neophytou, K. Mueller, K. T. McDonnell, W. Hong, X. Guan, H. Qin, and A. Kaufman. GPU-accelerated volume splatting with elliptical RBFs. In B. S. Santos, T. Ertl, and K. Joy, editors, *EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization*, pages 13–20, Lisbon, Portugal, 2006. Eurographics Association.

[41] S. Owada, F. Nielsen, and T. Igarashi. Volume catcher. In *Proc. of the Symposium on Interactive 3D graphics and games '05*, pages 111 – 116, 2005.

[42] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, and Kru. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*.

[43] H. Pfister, M. Zwicker, J. v. Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *Proc. of SIGGRAPH '00*, pages 335 – 342, 2000.

[44] D. L. Pham, C. Xu, and J. L. Prince. A survey of current methods in medical image segmentation. *In Technical Report JHU/ECE 99-01, The Johns Hopkins University*, 1999.

[45] M. Rochowski. The frenet frame of an immersion. *Journal of Differential Geometry*, 10:181–200, 1975.

[46] A. Rosenfeld and A. Kak. Digital picture processing. *New York Academic Press*, 2:138 – 145, 1982.

[47] R. J. Rost. *OpenGL(R) Shading Language*. Addison-Wesley Professional, 2006.

[48] H. Scharsach. Advanced gpu raycasting. In *Central European Seminar on Computer Graphics 2005*, pages 69–76, 2005.

[49] R. Schmidt and B. Wyvill. Generalized sweep templates for implicit modeling. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 187–196, New York, NY, USA, 2005. ACM.

[50] A. Sherbondy, M. Houston, and S. Napel. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *Proc. of IEEE Visualization '03*, pages 171 – 176, 2003.

[51] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, K. Sung, W. Thompson, and P. Willemsen. *Fundamentals of Computer Graphics, Second Ed.* A K Peters, Ltd., 2005.

[52] C. Sigg and M. Hadwiger. *Fast Third-Order Texture Filtering in GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005.

[53] F.-Y. Tzeng, E. B. Lum, and K.-L. Ma. A novel interface for higher-dimensional classification of volume data. In *Proc. of IEEE Visualization '03*, pages 505 – 512, 2003.

[54] I. Viola, A. Kanitsar, and M. E. Gröller. Importance-driven volume rendering. In *Proceedings of IEEE Visualization'04*, pages 139–145, 2004.

[55] S. W. Wang and A. E. Kaufman. Volume sculpting. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 151–ff., New York, NY, USA, 1995. ACM Press.

[56] A. Watt. *3D Computer Graphics*. Addison-Wesley Publishing Ltd, 1989.

[57] D. Weiskopf, K. Engel, and T. Ertl. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.

[58] L. Westover. Footprint evaluation for volume rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 367–376, August 1990.

[59] A. S. Winter and M. Chen. Image-swept volumes. *Computer Graphics Forum*, 21(3):441–441, 2002.

[60] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide Third Edition*. Addison-Wesley Publishing Ltd, 1999.

[61] B. Wyvill, A. Guy, and E. Galin. Extending the csg tree - warping, blending and boolean operations in an implicit surface modeling system. *Computer Graphics Forum*, 18(2):149–158, 1999.

[62] M. Zwicker, H. Pfister, J. v. Baar, and M. Gross. EWA splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.

Appendix A

Volume Studio

In this section, we present the interface of our system—volume studio. Figure A.1 shows a quick start-up screen after the application is loaded. This wizard allows the user to quickly select from a set of pre-configured volumetric data sets. The colored image on the icon depicts a pre-segmented data set, and raw volumes otherwise. The user can also click on the folder icon to load other volumetric data.

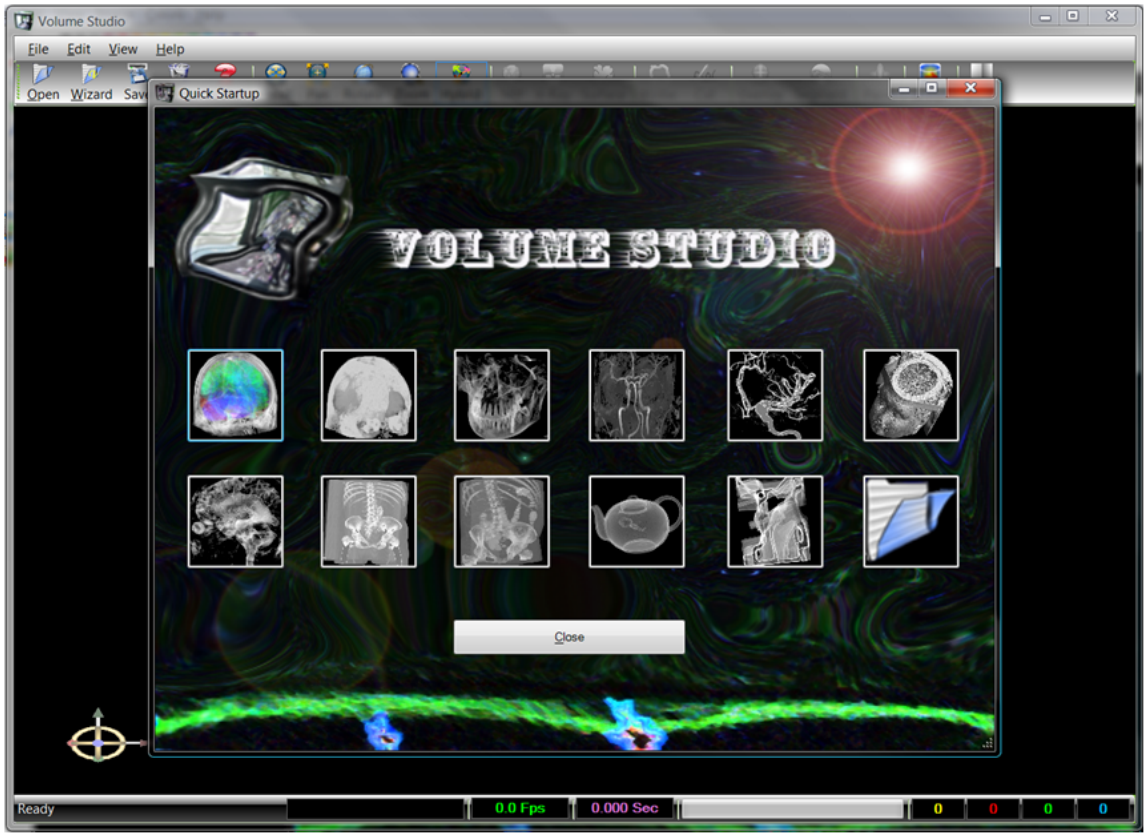

Figure A.1: The quick start-up screen.

Figure A.2 shows the main system interface after the data is loaded. The top-left border shows the name of the data file loaded. The main toolbar (near the top) organizes the major functions of the system. The first group from the left (i.e. Open, Wizard, Save, Close, and About) handles the file operations. The second group (i.e. Reset, Pan, Rotate, Zoom, and Hybrid) manages the scene transformations. The hybrid mode combines rotation, panning, and zooming operations. The third group (i.e. MIP, X-Ray, and Surface) switches between the different rendering modes in raycasting. The fourth group (i.e. Probe and Cut) turns on the volume sculpting engine. Probing is the reverse rendering of cutting. The fifth group (i.e. Overlay and Segment) controls the rendering mode during the segmentation process. Overlay shows the segmentation result in surface mode and the background in X-ray mode. The Segment option renders both the segmentation result and the background in surface mode. For the remaining buttons, the Paste option allows the sculpting or segmentation result to be pasted back to the scene. The Sculpt option turns on the experimental mode for building swept volumes.

On the left of the screen, we can find the histogram panel. The first chart (from the top) shows the intensity histogram for displaying the data. The following charts are transfer function histograms, which are organized as the intensity histogram, the gradient magnitude histogram, and the 2D histogram respectively. The last chart (at the bottom) shows a color table for the transfer functions.

On the right of the screen, we see the various tools for volume manipulation. The first group (from the top, i.e., Rectangle, Ellipse, Circle, Poly-lines, Free-form, and Clear) defines the various options of sketch shapes. The second group (i.e. Sweep, Laser, Drill, Peel, Seed, and Clear) lays out the manipulation tools. In the last

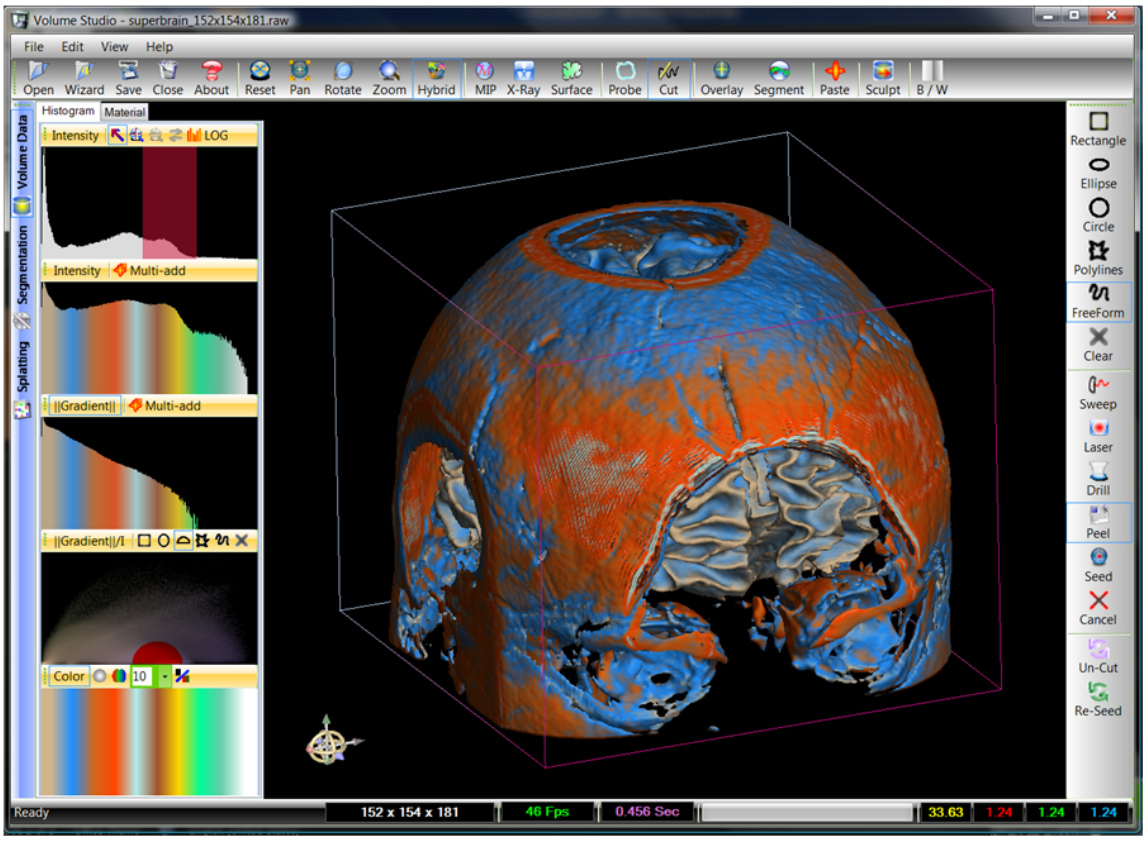

Figure A.2: The system interface with the histogram panel.

group, Un-Cut is an option to undo the entire sculpting operation, and Re-Seed is an option to remove all segmentations.

On the bottom of the screen, we find the various status, which include the notification area, size of volumetric data, frame rate recorder, time elapsed, progress bar, and the coordinate system information.

Figure A.3 shows the segmentation panel that is located on the left of the screen. The top portion organizes the region growing constraints, including the intensity and gradient magnitude threshold controls. The bottom portion puts the playback functions as VCR-like buttons, including the pause, play, forward, backward, and

the stop option. The user can also adjust the speed of region growing via the slider provided. The Auto Start button helps decide whether the segmentation will automatically begin after a stroke has been placed.

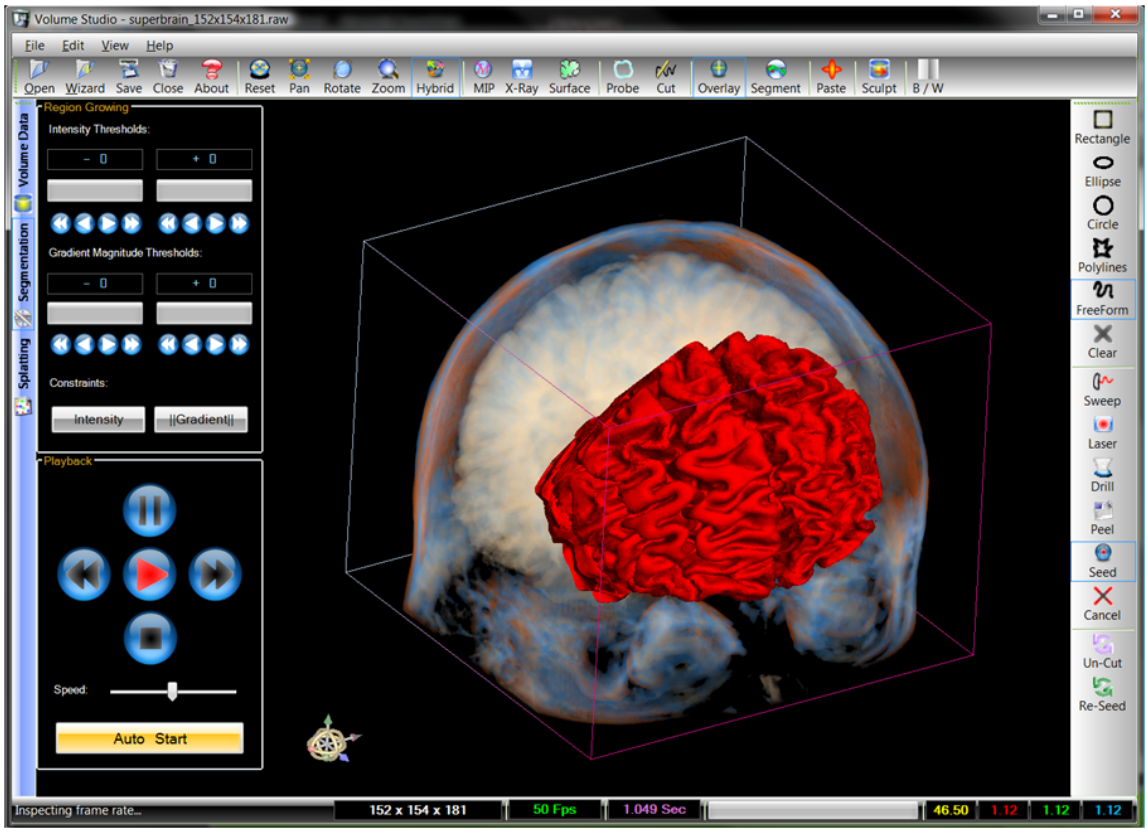

Figure A.3: The system interface with the segmentation panel.

Figure A.4 illustrates the varying types of sketching tools applied on the Utah Teapot, including the Rectangle, Ellipse, Poly-lines, and the Free-form tool. Each of the sketching tool shapes could be selected while performing the laser, drilling, peeling, and the seeding operation. Figure A.5 depicts the use of the free-form tool while performing the seeding operation.

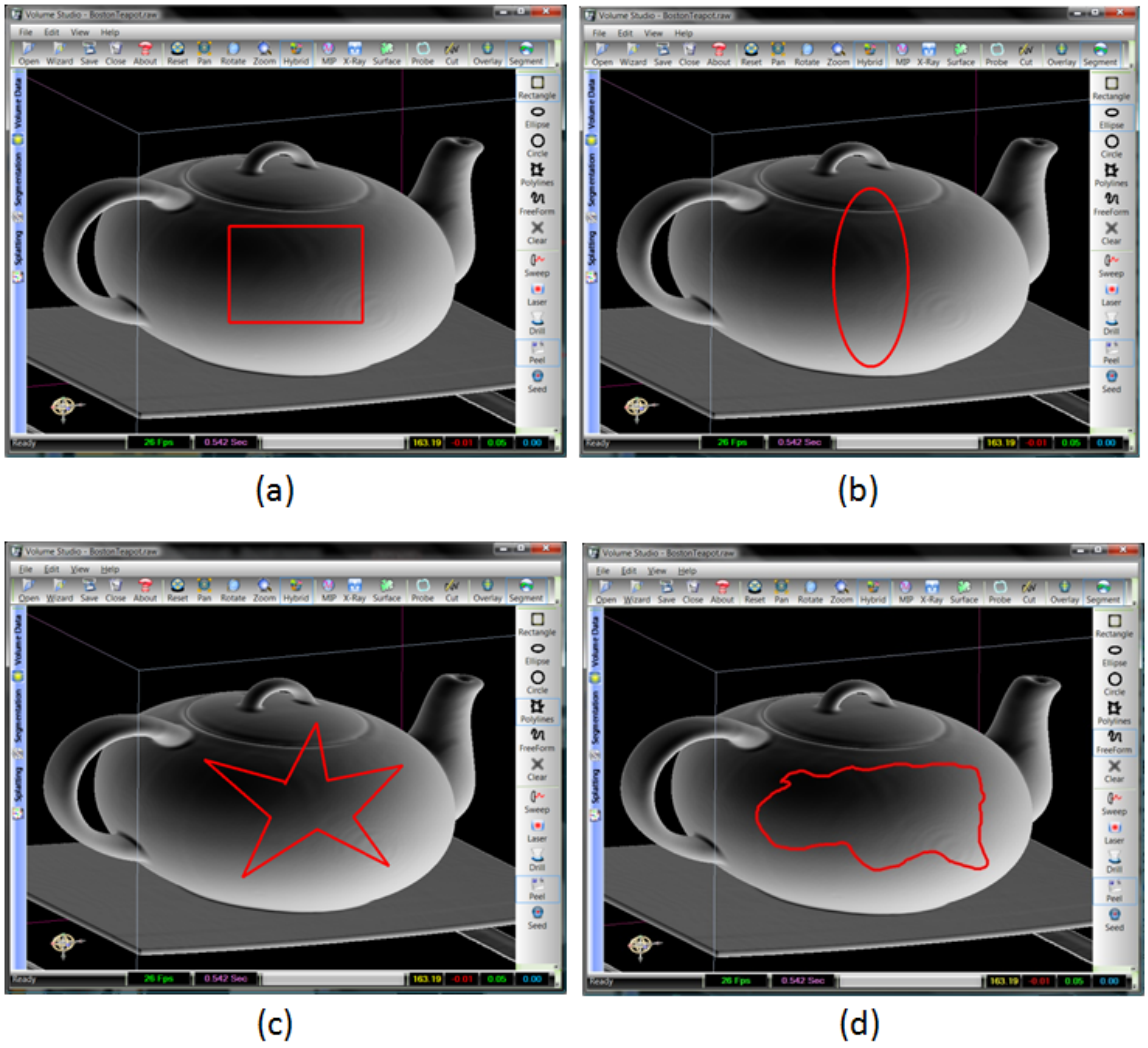

Figure A.4: The different types of sketch tool shapes: (a) the Rectangle tool, (b) the Ellipse tool, (c) the Poly-lines tool, and (d) the Free-form tool.

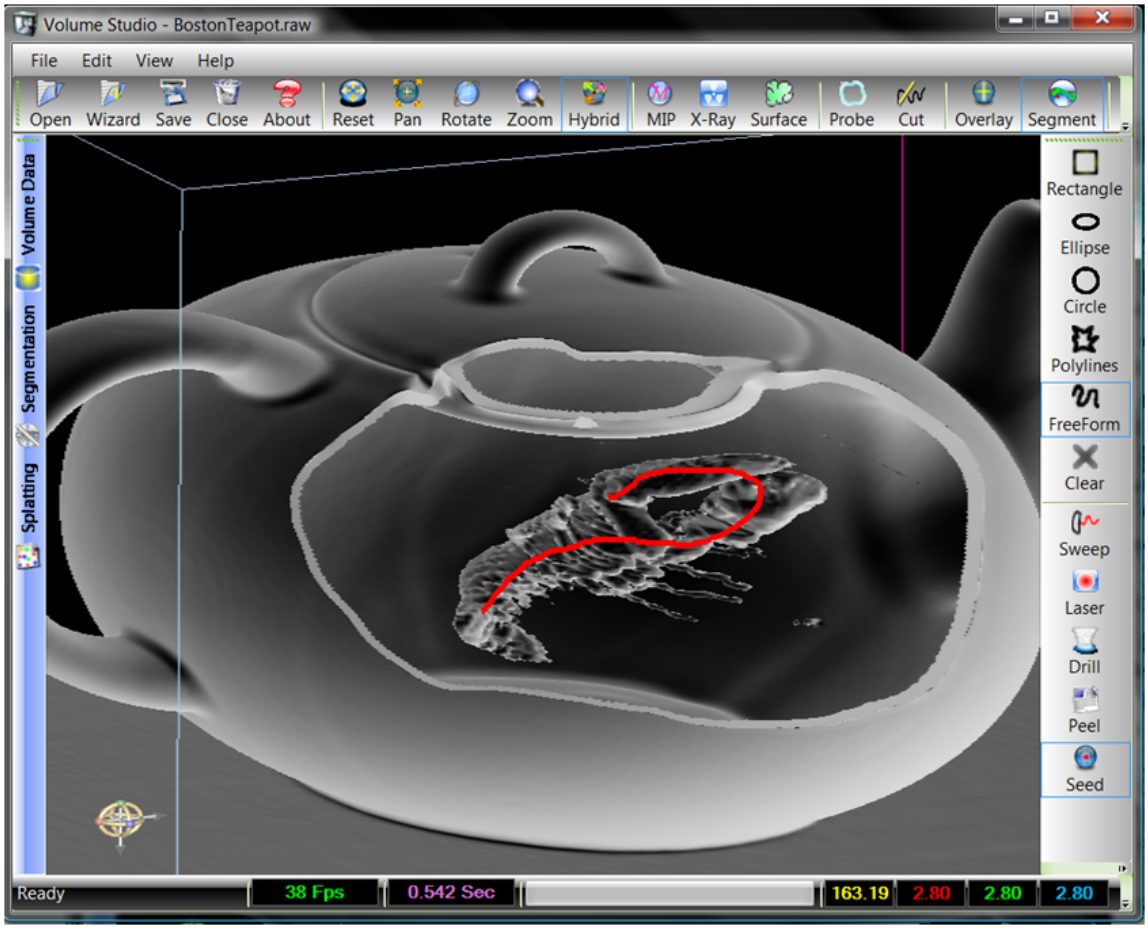

Figure A.5: Sketching seeds on the lobster surface by selecting the Free-form sketch tool shape.

Appendix B

GPU-Based Volume Graphics

The size of volumetric data is large, especially in the medical imaging field. For example, the data produced by MRI or CT ranges anywhere from 128x128x128 to 512x512x512. In the past research in volumetric data, CPU—a serial processing unit—had always been the major development platform. However, to process the volumetric data using a sequential programming paradigm is often a cumbersome task. Manipulating and rendering the data may take several minutes in order to obtain an intermediate result. For example, the famous marching cubes algorithm [35] takes an input volume and generates an iso-surface representation for use in the standard mesh-based rendering pipeline. The algorithm iteratively steps through each voxel and examines the iso-values of the eight neighboring grid points. The running time of the algorithm is hence proportional to the size of the input volume. Even with modern CPU horsepower, interactivity for the marching cubes algorithm is still difficult to achieve and the process is usually taken off-line.

Commodity graphics hardware has become ubiquitous and provides a powerful computational platform harnessing the massively parallel horsepower. The chips on the graphics board are designed for scalar/vector math computation in blazing speed. Special hardware is dedicated for pipelined processing with thousands of threads executing in concert. In recent years, programmable graphics processing units (GPUs) have been introduced and are frequently adapted by developers to perform general-purpose computation (GPGPU [27]). Volume graphics also fits well

in the new programming paradigm and has a great potential for taking enormous advantages of the latest graphics hardware. For example, a volumetric data set can be partitioned into independent voxels and processed in parallel. Each voxel may contain various properties to be processed, and the different tasks can be delegated to the programmable units on the graphics hardware.

In the following sections, we first provide an introduction to the graphics hardware in Section B.1, and then we review the classical rendering pipeline in Section B.2. Section B.3 discusses the details of a modern GPU developed by the NVIDIA Corporation. Section B.4 outlines the new features in the latest graphics APIs including the DirectX 10 system and the corresponding OpenGL extensions.

B.1 Introduction

B.1.1 Computational Power

Commodity graphics hardware has become a cost-effective computational device that is both powerful and easily accessible from most retail stores nowadays. In a recent survey of general-purpose computation on graphics hardware [42], Owens et al. provided a comparison between GPUs and CPUs on the memory bandwidth and computational horsepower. They pointed out that the flagship NVIDIA GeForce 7900 GTX ($378 as of October 2006) boasts 51.2 GB/sec memory bandwidth; and a similarly priced ATI Radeon X1900 XTX can sustain a measured 240 GFLOPS, both measured with GPUBench [11]. GPUs also use advanced processor technology (e.g. the ATI X1900 contains 384 million transistors and is built on a 90-nanometer fabrication process) [42]. While comparing to a dual-core 3.7 GHz Intel Pentium

Extreme Edition 965, only 8.5 GB/sec and 25.6 GFLOPS theoretical peak was observed. Statistics shows that the gap between CPUs and GPUs is rapidly expanding especially in the last couple of years.

More recently, the next generation graphics chips–NVIDIA GeForce 8800 GTX–have arrived with computational power measured 330 GFLOPS (observed), memory bandwidth of 55.2 GB/sec (observed), and a price at $599. However, the latest 3.0 GHz Intel Core2 Duo (Woodcrest Xeon 5160) with a higher price at $874 only boasts computational power measured 48 GFLOPS (peak) and memory bandwidth of 21 GB/sec (peak) [27]. Statistical data shows that CPUs only have a 1.4x annual growth rate while GPUs are growing at 1.7x (pixels) to 2.3x (vertices) annually [27]. It is clear that GPUs outpaced the computational power on CPUs. The fact that GPUs are growing at a higher rate than CPUs can be attributed to mainly two reasons. Firstly, when the advances in fabrication technology for both platforms stay at a similar rate, there exists a fundamental architecture distinction. CPUs are deeply pipelined for sequential processing and optimized for branch predicting. On the other side, GPUs are dedicated to graphics processing and parallel execution. Hence, it becomes easy for GPUs to have additional transistors for computation and can achieve higher arithmetic intensity. In addition, massive parallelism on the GPUs further hides the memory latency issue. The second reason falls more on the economic side. The multi-billion dollar gaming industry drives the demands for a better and faster gaming platform. A recent cooperation between Microsoft's DirectX 10 technology and NVIDIA's GeForce 8 Series hardware demonstrates an excellent example for pushing the limits of graphics hardware capability.

B.1.2 Flexibility and Programmability

Modern graphics hardware has become more flexible in terms of its programmability, usability, and extensibility. Video cards in the early days only supported fixed-function pipelines where direct programmability was not available. Later on, the programmable vertex and fragment shaders were added to the conventional pipeline. More recently, the programmable geometry shader has emerged to allow even more flexibility for mapping algorithms onto the graphics hardware. The GPU data output capability evolves from the original 8-bit per color channel to a full IEEE 32-bit floating-point precision per color channel. Current graphics cards also support vectorized and scalar operations, integer and floating-point operations, bit-wise operations, dynamic branching, and full texture access in all programmable stages of the pipeline. High level languages have emerged to ease the programming task and mostly appear in a C-style format. Examples are the Microsoft High Level Shading Language (HLSL) [24] and the OpenGL Shading Language (GLSL) [47]. The continuous increase in hardware speed and rapidly expanding parallelism has made the GPU programming platform an ideal solution for real-time volume graphics.

B.2 The Classic Rendering Pipeline

Before we take a close look at the latest GPU architecture, we review the classic graphics pipeline that was used over the past years. Figure B.1 shows an overview of DirectX 9 equivalent pipeline containing both programmable vertex and fragment shaders.

Both DirectX Shader Model 3.0 and OpenGL Shading Language version 1.10

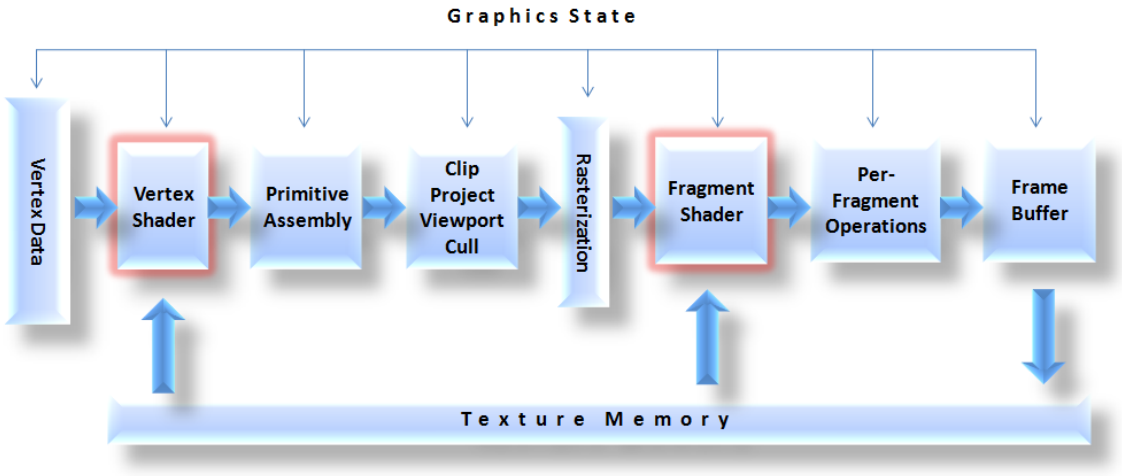

Figure B.1: Overview of the classic graphics pipeline showing the major stages with programmable vertex and fragment shaders.

support the ability to load a set of rather limited program instructions at driver load time. Instructions could be written to control either per-vertex operations (vertex shader) or per-fragment operations (fragment shader). The programs are stored in the video memory and could be switched from one to another between pipeline rendering cycles in order to achieve the various effects.

B.2.1 Vertex Data

From the beginning of the pipeline, the application program prepares vertex data that is stored either in the graphics device accessible memory (i.e. the video memory) or the host memory (i.e. the system RAM). The vertex data is used as initial input to the rendering pipeline. At this time, the application program also sets up graphics states to control custom processing behaviors for each of the rendering stages (e.g. culling should be clockwise, viewport is set to 800 x 600, the render target is replaced by off-line textures, and etc.).

B.2.2 Vertex Shader

The first stage of the pipeline is handled by the vertex shader. The vertex shader collects vertex data and performs per-vertex operations. If no vertex shader is available, fixed-function vertex processing is applied instead; that is, vertex positions are transformed by the model-view and projection matrices, normals are transformed by the inverse transpose of the 3 by 3 matrix derived from the upper left portion of the model-view matrix, texture coordinates are transformed by the texture matrices, vertex colors are modified based on lighting and material properties, texture coordinates are generated, and point sizes are adjusted [47]. During this stage, the object coordinate system has gone from world-space coordinates and camera-space coordinates to screen-space coordinates.

B.2.3 Primitive Assembly

After individual vertices have been processed, the next stage is primitive assembly where vertices are assembled into primitives. Depending on the primitive type, different number of vertices is required to compose a complete primitive. For example, a point requires one vertex, a line needs two vertices, a triangle requires three, and a general polygon can have more than three vertices. The primitive assembly stage is necessary at this moment because the processing in the following stages require a collection of vertices and the primitive types to be known (e.g. clipping can be varied among points, lines and polygons) [47].

B.2.4 Primitive Processing

The third stage comprises of several sub-routines including clipping, perspective projection, viewport transformation, and culling. Clipping applies any user-defined clipping planes and compares each primitive against the view volume (i.e. defined by the model-view-projection matrix). Primitives pass the clipping test if they fall inside the view volume, but otherwise rejected. Primitives are clipped if they are partially within the view volume. The next operation is perspective division. If perspective projection is used, the x, y, and z components of each vertex is divided by the w component. This operation is followed by the viewport transformation. Viewport operation transforms the object coordinate system from screen-space coordinates to window-space coordinates. Finally, the culling operation can be enabled to discard front-facing polygons, back-facing polygons, or both [47].

B.2.5 Rasterization

The rasterization stage decomposes geometric primitives, passed through the early pipeline stages, into smaller units (fragments) corresponding to pixels in the destination frame buffer. A basic fragment consists of a windows coordinate, depth value, and other attributes such as color and texture coordinates. Different shading models control the rasterization result. Smooth shading interpolates color values between vertices; whereas flat shading uses the last vertex color value for the entire primitive. The rasterizer can also be configured to allow anti-aliasing operations to be applied for points, lines, and polygons [47]. Furthermore, the point sprite mechanism allows a textured image to be attached to a point primitive and automatically generates a texture coordinate for each fragment produced.

B.2.6 Fragment Shader

After a series of fragments emitted by the rasterization process, a fragment shader performs first-level per-fragment operations. A fragment shader has the ability to discard the incoming fragment and eliminate further processing. If the fragment shader is absent, then the fixed-function fragment shading is applied instead. The most important operation happened in this stage is texturing. The texturing process takes incoming texture coordinates, performs texture look-up (including filtering, mip-mapping, and texture address translation), and modifies the fragment color value [47].

B.2.7 Per-Fragment Operations

After the fragment shader, successfully processed fragments go through the remaining per-fragment operations and they include the scissor test, alpha test, stencil test and depth test. The scissor test clips fragments against a rectangular region. The alpha test blends the incoming color values with the destination color values. The stencil test compares the stencil buffer with a reference value. Finally the depth test decides whether to draw the fragment or not.

B.2.8 Frame Buffer

In the final stage of the pipeline, pixel values are recorded in the frame buffer (the video memory) and rendered to the screen. Optionally, the pixels could be rendered off-line to the texture memory, and these textures could be referenced by the vertex and fragment shaders in the next rendering cycle.

B.3 NVIDIA GeForce 8800 GPU

After reviewing the classic rendering pipeline, we now examine the next generation graphics hardware, the NVIDIA GeForce 8800 GPU. NVIDIA's GeForce 8800 is the industry's first GPU equipped with a unified architecture. A total re-resign on the 8800 reveals a pioneering work in the hardware advancement and marks a watershed in the history of GPU revolution. GeForce 8800 is also the first GPU compatible with the new DirectX 10 technology [6]. In addition, the GeForce 8800 GPU can be configured with the NVIDIA SLI technology to double the performance by allowing multiple GPUs (currently up to 2x) to function simultaneously.

B.3.1 Unified Design

The GeForce 8800 GPUs employ a parallel and unified shader design. The flagship hardware (8800 GTX) is equipped with 128 stream processors each clocked at 1.35 GHz and can be dynamically allocated to vertex, geometry, or fragment operations for optimized load balancing. Compared to the last generation GPU (7900 GTX), 8800 GTX has an alleged twice the performance and up to 11 times scaling measured in some shader operations [19]. NVIDIA GeForce 8 series GPUs also implement the GigaThread technology to allow thousands of independent threads to execute in parallel for maximum GPU utilization [19].

The main motivation for having a unified design over the conventional setup is based on the fact that different applications have different shader processing requirements at any given point in time [19]. In general, we see a significant amount of processing requirement for pixels over vertices. Although the traditional GPU de-

sign tries to implement more pixel units than vertex units in order to satisfy the workload pattern, but the same strategy would not be valid in different scenarios. Some applications spend more time computing heavy geometry, and hence the vertex shader becomes overloaded while leaving the pixel shader idle the majority of the time. Others may focus on rendering and therefore use more pixel shader than vertex shader. Both scenarios explain poor utilization of the GPU hardware.

The classic GPU pipeline processes data in a linear fashion. For example, the pixel shader on the GeForce 7 series GPUs has over 200 sequential and physical pipeline stages. The unified shader architecture of the GeForce 8 series GPUs revamps the serial design and establishes a loop-oriented approach. The discrete design as in the classic pipeline has dedicated hardware for each shader type. The processing starts from the top with the data going through each shader in a sequential order. The unified design condenses all shader types into a generic shader core receiving data from the input buffer (ibuffer) and putting results in the output buffer (obuffer). The same shader core is re-used for each pipeline stage to complete the entire rendering cycle. The result of unifying shaders enables maximum GPU utilization for different types of application requirements. Now, the previous two scenarios become fully optimized with both achieving the highest workload performance. The first type of applications is now able to allocate more unified shaders for its heavy vertex workload and the second type can better concentrate on pixel-related tasks.

B.3.2 Stream Processing Architecture

The GeForce 8800 architecture replaces the existing pipeline model and is built around the stream processors. The first component is the host interface that contains

buffers for storing commands, vertex data, and textures. The input assembler collects vertex data from buffers and converts them to 32-bit floating-point format. The input assembler also generates index IDs for vertices, primitives, and instances (new in the DirectX 10 model) [19]. Next, the vertices are issued and handled by the next available stream processor (in green). Vertex, geometry, and pixel threads are scheduled by the thread processor. Until sufficient number of vertex threads are completed (based on the primitive type), a geometry thread is then used to collect vertex data and output primitive information. After the geometry thread has finished, the setup/rasterization/ZCull operations are invoked and followed by the pixel thread issues.

The GeForce 8800 GTX GPU contains 128 stream processors. Every stream processor can be assigned to perform different type of shader operations. The stream processors are further organized into clusters. A cluster contains a collection of stream processors (SP) with associated numbers of texture filtering (TF), texture addressing (TA), and cache units to ensure a balanced design. Each GeForce 8800 GPU stream processor is generalized, decoupled, can dual-issue a MAD and a MUL, and supports IEEE 754 floating-point precision [19]. In addition, the GeForce 8800 GPU adapts a scalar processor architecture where as existing GPUs have used vector processing units (since graphics operations are mostly based on 4-components). As more scalar operations occur in increasing number of applications, the scalar processor architecture has been proved to be more efficient than a mixed of scalar and vector processor architecture.

B.3.3 Performance

Each stream processor on the GeForce 8800 GTX operates at around 520 giga-flops of raw shader power. Instruction issue is 100% efficient with scalar shader units. Mixed scalar and vector shader program performs better than vector-based GPUs. Texture filtering units are decoupled from the stream processors and can obtain full speed bilinear anisotropic filtering. Texture units run at the core clock rate (i.e. 575 MHz for GeForce 8800 GTX) and has a fill rate of 18.4 billion texels/second [19]. As volume graphics often requires frequent texture access, the improvement in the GeForce 8800 GTX is noteworthy.

The GeForce 8800 GTX has six raster operation partitions. Each partition can process 4 pixels (or 16 sub-pixels samples). These amount to a total of 24 pixel/clock output capability with color and Z processing [19]. The raster operations also support frame buffer blending of 16-bit/32-bit floating-point render targets. Eight render targets are now available and each render target can have different color formats [19]. For the memory sub-system, a single GeForce 8800 GTX GPU contains six memory partitions and each partition provides a 64-bit interface. Therefore, the GeForce 8800 GTX GPU supports a total of 384-bit memory interface (48 byte-wide and running at 900 MHz) with a total memory bandwidth equal to 86.4 GB/sec [19].

B.4 Graphics API

Graphics APIs like DirectX and OpenGL have been appreciated by developers for providing a middle layer between the application and the graphics hardware for many years. In this model, applications issue a number of commands and the APIs

translate the commands to a machine understandable format. One problem with using this model is the high CPU overhead. Every time a command is dispatched to the API layer, the CPU needs to process it in order for the GPU to proceed. For example, state changes occurs frequently and often becomes a bottleneck for building rich and complex effects [18].

Microsoft has released the DirectX 10 system [6] that completely revamps the existing graphics APIs. DirectX 10 builds from the ground up and comprises a highly optimized architecture. The new features include the powerful geometry shaders, stream out capability, texture arrays, and the ability to render to 3D texture layers dynamically. Although the DirectX technology has evolved over the years and added new features in every release, however, the DirectX 10 platform is an entire re-architecture. After three years of research and development in collaboration with NVIDIA, the GeForce 8800 series architecture has emerged to deliver the world's first DirectX 10-compliant GPUs [18].

In the new DirectX 10 technology, a new runtime has been introduced to reduce the cost of draw calls and stage changes [18]. For example, the resource validation in DirectX 10 only executes once upon resource creation rather than being executed every time a resource is used. The following sections describe the new shader model, geometry shader, and a set of new features introduced by DirectX 10. Mappings from the NVIDIA OpenGL extensions for the GeForce 8 series will also be provided.

B.4.1 Shader Model 4.0

Graphics APIs offer high-level languages for writing shading programs that are explicitly loaded into the GPU. The shading programs are used to provide custom

behaviors such as special rendering effects, physics simulation, and general-purpose computation. Both DirectX and OpenGL offer a virtual machine-like environment for handling custom shading requirements. In this programming model, the GPU is virtualized, and device-independent shader code is compiled to specific GPU machine code at runtime by the graphics driver's built-in Just-In-Time (JIT) compiler.

DirectX 10 introduces Shader Model 4.0 and leverages a unified shading architecture. NVIDIA also provides a set of OpenGL extensions for the new architecture [17]. In the new architecture, a unified instruction set and common resources are well-recognized and shared by all three types of shaders (i.e. the vertex, geometry, and fragment shader); where in the previous shading models, only a limited set of instructions and resources were available. Shader Model 4.0 uses a unified instruction set with the same number of registers (temporary and constant) and inputs across the programmable pipeline [19]. There is also a vast increase in resources for shader programs. In addition, the supported size for texture 3D has been raised to 2048x2048x2048. The large 3D texture capacity is going to satisfy the ever-increasing size of volumetric data sets.

B.4.2 Geometry Shader and Stream Out

In prior shading models, vertex and fragment shaders were available but only had the ability to modify/forward the pipeline data (i.e. no data was allowed to be generated). The input data had to travel through the entire pipeline and the result was recorded in the frame buffer memory or off-screen textures.

When the seeded region growing algorithm was used to segment a volume, each growing step required generating new neighborhood seeds provided that the crite-

ria were met. Checking for neighborhood criteria on the GPU could be done by performing 3D texture lookups but producing more seeds were not possible. In the implementation by Chen et al. [13], the seeded region growing process was solely handled by the CPU and the result was then sent to the GPU for animation. Nevertheless, this prevents a real-time feedback for every seed placement.

Figure B.2 depicts the new DirectX 10 pipeline. In this new pipeline model, two innovative features are introduced: the geometry shader and the stream output. And it is also for the first time, the geometry shader enables data to be generated on the graphics processor. The geometry shader sits after the vertex shader and before the rasterizer while replacing the primitive assembly stage of the fixed-function pipeline (shown in Figure B.1). The geometry shader receives vertex data processed by the vertex shader and emits up to 1024 vertices as output [18]. Besides the ability to generate more data on-the-fly, the geometry shader is also allowed to delete input vertices (i.e. similar to how a fragment shader discards an incoming fragment). Thus, the data amplification and minimization tasks can solely be performed on the GPU without CPU intervention. This enables a variety of applications including to compute the seeded region growing algorithm directly on the GPU.

The geometry shader supports three types of input and output primitives including points, lines, and triangles. For lines and triangles primitive types, adjacency information is also available. However, only one of these primitive types can be selected as input and output type for every instance of the geometry program. In OpenGL, these features are supported via the GL_EXT_geometry_shader4 and GL_EXT_gpu_shader4 extensions [17].

DirectX 10 has enabled layered rendering through the geometry shader. This

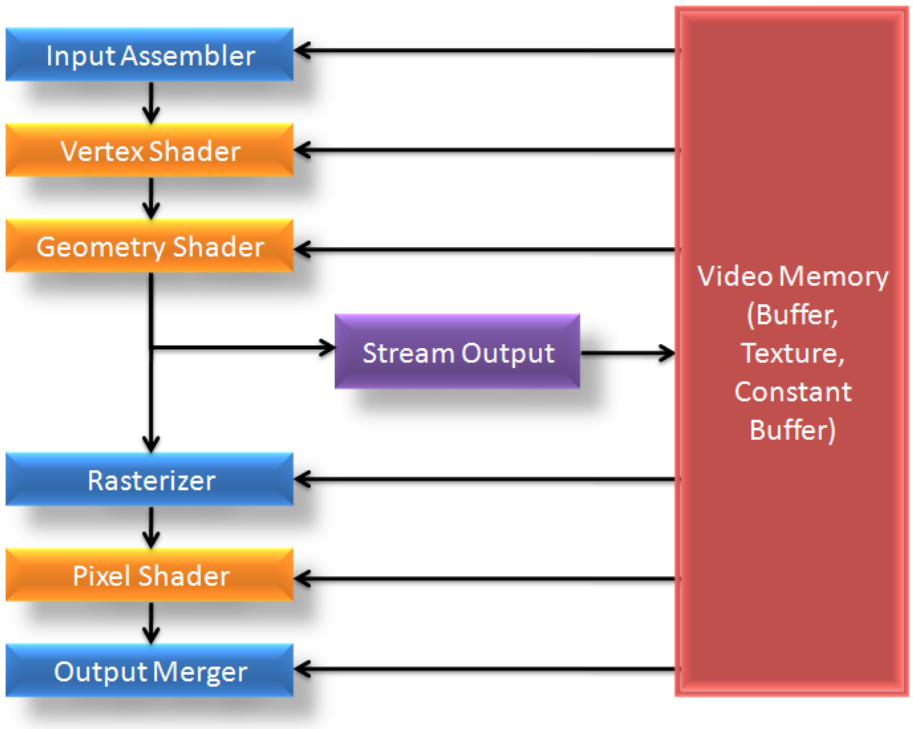

Figure B.2: The new DirectX 10 pipeline adds the geometry shader and stream output to allow data to be manipulated at incredible speed. [19]

feature is by far the most important highlight for volume computation using the GPU. In the previous graphics APIs, a 3D texture could be attached as a render target but only a single layer could be rendered to for every rendering pass. Now with layered rendering, each instance of the geometry program has the ability to dynamically compute and assign a texture layer for rendering. Together with the geometry shader's ability to change the vertex position, any voxel location can now be specified within the 3D texture.

B.4.3 HLSL 10 and GLSL 1.20

Besides the geometry shader as a revolutionary element in Shader Model 4.0, DirectX 10 also introduces a number of core features in the high-level shading language (HLSL) including improved instancing, state objects, constant buffers, views, and integer/bitwise instructions. Most of these features are also supported by OpenGL shading language (GLSL) version 1.20 (previously 1.10).

Object instancing enables a single API draw call to perform a number of similar operations on the same set of vertex data. In HLSL 10, shader programs are allowed to read in index values of texture arrays, render targets, and indices for different shader programs. In GLSL 1.20, index for vertices, primitives, and instances are available as read-only built-in variables via the GL_EXT_gpu_shader4 extension [17]. Instancing is particularly useful when working with iterative computations. For example, a WxH-sized quad can be rendered D number of times to compute a certain property for an input volumetric data set. In this case, the instance ID represents the volume slice index. OpenGL supports instancing via the GL_EXT_draw_instanced extension.

Constant buffers is yet another important feature in the new shading languages. Constants are defined in all shader programs as global parameters and are maintained by the application. For example, light attributes, ray-casting step size, and various filter sizes are all defined by constants. In HLSL 10, constant buffers allow up to 4096 constants be stored in a buffer [18]. In GLSL 1.20, a buffer object can be bound to program constants via the GL_EXT_bindable_uniform extension. Using constants backed by buffer objects enables quick data specification and a variety of effects.

For example, values written by the geometry shader can be read back by the vertex shader using bindable constants (or uniform variables).

DirectX 10 removes the notion of typed resources and introduced different *views* of the same resource. With multiple *views*, the same resource can be re-used for different purposes [18]. For example, the texture written by the fragment shader can be interpreted as a vertex buffer that is subsequently read by a vertex shader. OpenGL supports this feature via the GL_EXT_texture_buffer_object extension. In addition to one-, two-, and three-dimensional and cube map textures, a new type of texture—buffer texture—has been introduced. A buffer texture is similar to a one-dimensional texture except that the texel array is backed by a buffer object. In volume graphics, histograms could be computed and stored in a buffer texture for CPU or vertex shader read-backs.

Integer and bitwise instructions are often required to make general algorithms a lot easier to map to the GPU. The new shading languages have finally enabled these features. In OpenGL, the GL_EXT_gpu_shader4 extension outlines the possible instruction sets. In addition to supporting integer computations, GLSL 1.20 also supports integer texture format via the GL_EXT_texture_integer extension.

With the new shading languages embracing the set of advanced features, a whole new era of parallel computation using the GPU has arrived to enable an infinite amount of possibilities. Especially in volume graphics, where parallelism drives for real-time results, the new set of flexible and manageable graphics APIs would undoubtedly allow more advanced algorithms to be mapped and inspire more complicated volumetric operations to be accomplished in a truly interactive mode.

B.5 Code Examples

In this section, we show some shader code examples utilizing the latest features in GLSL version 1.2. The following sub-sections provide the vertex shader, geometry shader, and the fragment shader for the Laser tool (Section 4.3.2) discussed in this thesis.

B.5.1 Vertex Shader

```
bindable uniform vec3 MaskCenter;
bindable uniform vec3 MaskNormal;
uniform float StepDistance;

varying vec2 MaskCoord;
varying vec3 MaskPos;

void main(void)
{
    gl_Position = gl_Vertex;
    MaskCoord = gl_MultiTexCoord0.st;
    MaskPos = MaskCenter - MaskNormal * float(gl_InstanceID) * StepDistance;
}
```

B.5.2 Geometry Shader

```
const int SplatSize = 6;
const int SplatSizeHalf = SplatSize / 2;
const float SplatSizeF = float(SplatSize);
const vec3 YAxis = vec3(0, 1, 0);
const vec3 XAxis = vec3(1, 0, 0);
const vec3 ZeroVec = vec3(0, 0, 0);

uniform sampler1D Kernel1Texture;
uniform sampler2DRect MaskTexture;
bindable uniform vec3 MaskX;
bindable uniform vec3 MaskY;
uniform vec3 VoxelSize;
```

```
uniform vec3 VoxelExtent;
uniform vec3 VolumeSize;

// Make sure that the inbound varyings are in the form of arrays,
// otherwise linking error will be generated!
varying in vec2 MaskCoord[];
varying in vec3 MaskPos[];
flat varying out float ZWeight;
flat varying out float Mask;

void main(void)
{
    float mask = texture2DRect(MaskTexture, MaskCoord[0]).r;
    if (mask == 0.0)
        return;

    // Particle position ranges from -0.5 to 0.5
    vec3 particlePos = MaskPos[0] +
        MaskX * gl_PositionIn[0].x + MaskY * gl_PositionIn[0].y;
    float particleZNormalized = particlePos.z + 0.5;

    // Output position should be within the unit square (-1.0 to 1.0)
    vec4 screenPos = vec4(particlePos.xy * 2.0, 0.0, 1.0);

    // Layer0 = layer2(next nearest layer) - (SplatSize / 2)
    int layer0 = int(round((particleZNormalized - VoxelExtent.z) /
        VoxelSize.z)) - SplatSizeHalf;
    int layerK = layer0 + SplatSize;
    float samplingPos;
    float radiationCenterInLayerCoord =
        particleZNormalized * VolumeSize.z - 0.5;

    for (int z = layer0; z < layerK; z++)
    {
        if (z >= 0 && z < VolumeSize.z)
        {
            samplingPos = 0.5 +
                (float(z) - radiationCenterInLayerCoord) / SplatSizeF;
            ZWeight = texture1D(Kernel1Texture, samplingPos).r;
            gl_Position = screenPos;
```

```
            Mask = mask;
            gl_Layer = z;
            EmitVertex();
        }
    }
}
```

B.5.3 Fragment Shader

```
uniform sampler2D Kernel2Texture;
uniform float KernelCutoff;
flat varying float ZWeight;
flat varying float Mask;

void main(void)
{
    vec4 XYWeight = texture2D(Kernel2Texture, gl_TexCoord[0].st);
    float weight = XYWeight.r * ZWeight;

    if (weight < KernelCutoff)
        discard;

    gl_FragColor = vec4(Mask * 0.1, 0, 0, weight);
}
```