Vis3D+

A TIGHTLY INTEGRATED GPU-ACCELERATED COMPUTATION AND RENDERING FRAMEWORK FOR INTERACTIVE 3D IMAGE VISUALIZATION

by

Irfa Nisar

Honors BSc, University of Toronto, 2004

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2015

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my dissertation may be made electronically available to the public.

Vis3D+

A TIGHTLY INTEGRATED GPU-ACCELERATED COMPUTATION AND RENDERING FRAMEWORK FOR INTERACTIVE 3D IMAGE VISUALIZATION

Master of Science 2015

Irfa Nisar

Computer Science

Ryerson University

Abstract

This thesis presents extensions to an interactive 3D image visualization framework. The existing software framework provides functionality for interactively visualizing 3D medical data. The extensions consist of software modules that execute directly on the graphics hardware, utilizing the massively parallel, general-purpose computing platform provided by modern graphics processing units (GPUs). These GPU-based software modules are designed to support the execution of volume image processing algorithms, implemented using recently available GPU programs known as "compute shaders", as well as to support interactive editing of the algorithms' output. The new modules are seamlessly integrated as new stages in a GPU-based rendering pipeline provided by the existing framework. In this thesis, an example volume image processing algorithm known as level set segmentation is implemented and demonstrated. In addition, a new editing module is demonstrated that enables user modification of this algorithm's output by extending a pre-existing volume "painting" interface.

Acknowledgements

I want to thank my supervisor, Professor Tim McInerney, for having the patience to listen to my ideas, theories and conjectures, some true, some false and everything in between. His astute feedback and specific advice on many issues has helped me shape this thesis from concept to implementation.

I profusely thank my hands-on father, who offered me extensive freedom and plenty of opportunities.

I also want to thank my mother for always providing a backdrop of support at home.

I want to thank my husband, Rashekh, whose encouragement has helped me on this journey and my daughter Inaya for patiently teaching me patience.

Contents

Declaration . . . iii
Abstract . . . v
Acknowledgements . . . vii
List of Tables . . . xi
List of Figures . . . xiii
List of Appendices . . . xv

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Thesis Outline . . . 3

2 Literature review **5**
2.1 GPU Programming and the Graphics Pipeline . . . 5
2.2 GPU-based Volume Rendering . . . 6
2.3 GPU-based Volume Image Processing . . . 9
2.3.1 GPU-based Volume Image Segmentation . . . 9
2.4 Interactive Volume Image Segmentation . . . 11

3 Methodology **13**
3.1 The Existing Framework . . . 13
3.2 The Extended Framework . . . 15
3.2.1 Painting and Editing on 3D Slice Planes . . . 16
3.3 Image Processing: Image Filtering . . . 19
3.4 Image Processing: Level Set Segmentation . . . 19
3.4.1 A Brief Look at Level Sets . . . 20
3.4.2 Initial Level Set Construction . . . 20
3.4.3 Overview of Level Set Implementation . . . 21
3.4.4 Compute Shader Implementation . . . 23
3.5 Editing a Labeled 3D Region . . . 25

4 Results and Validation **27**
4.1 Segmenting Synthetic Data Sets 27
4.2 Segmenting Real Data Sets 29
4.3 Editing 30
4.4 SpeedUp 30

5 Conclusions **33**

Bibliography **37**

List of Tables

4.1 Segmentation Time Comparison of Central Processing Unit (CPU) versus Graphics Processing Unit (GPU) (100 iterations) 31

List of Figures

2.1 Simplified diagram of a modern 3D graphics pipeline. 5
2.2 A 3D medical image depicted as a stack of 2D slices and as a 3D grid of voxels. Each voxel stores an intensity value. 6
2.3 Depiction of volume ray casting. (a) Example of a transfer function for mapping intensity values to a color and opacity. (b) Ray casting into the volume and sampling at regular intervals along the ray. 7
2.4 Example of Direct Volume Rendering (DVR) using a CT image of the hand. In (a) the user maps volume intensity values to a color and opacity. 8

3.1 Exposing bones in a CT hand data set by painting and blending a series of superellipsoid brush strokes to form a 3D region of interest. 13
3.2 High-level overview diagram of the existing framework. 14
3.3 Graphics Pipeline showing the stages of the existing framework. 15
3.4 High level overview of the extended framework. 16
3.5 Graphics Pipeline showing the stages of the extended framework. 17
3.6 Painting on a 3D slice plane. (a) Paint brush "tip" blob, rendered as an opaque surface, can slide along a slice plane that clips the volume rendered data. (b) Application of paint brush strokes on the slice plane. The 3D surface of the painted envelope is transparent, and a simple transfer function highlights the voxels inside the envelope and on the slice plane. 17
3.7 High Level Overview of editing a labelled 3D region via 3D slice paint painting. 18
3.8 Overview of basic image processing pipeline. 19
3.9 Depiction of level set evolution in 2D. 20
3.10 Illustration of initial level set. Values inside the Volume of Interest (VOI) are marked as $-\rho$ and outside the VOI are marked as $+\rho$, where ρ is a constant (28). 21
3.11 Overview of level set segmentation-processing pipeline. 22
3.12 Parallel computing in a compute shader. A computational problem on a 3D grid, exhibiting a parallel pattern, is broken down and computed in parallel on compute shader threads. 23
3.13 Conceptual Diagram of Adding and Removing Paint 26

4.1 Segmentation algorithm run on a a 256 x 256 x 256 cloverleaf dataset. (a),(b) Initial painted level set surface. (c),(d) Final segmentation result. 28
4.2 Segmentation algorithm run on a 168 x 160 x 92 CT image of a human vertebra phantom. (a) Initial painted level set surface. (b),(c) Final segmentation result. 28
4.3 Segmenting the lateral ventricle in a 240 x 240 x 192 MRI brain image. (a) Initial 3D slice painted level set surface. (b) Final segmentation result. (c) Volume rendering of manually segmented ventricle. 29
4.4 Slice-by-Slice examination of the segmentation of the lateral ventricle in a 240 x 240 x 192 MRI brain image. In (a),(b),(c) user scans through the slices to visually verify the segmentation result. 29
4.5 Slice-by-Slice examination of the caudate nucleus segmentation. In (a) and (b) different slices are shown and the user scans through the slices for a visual check of the segmentation. 30
4.6 Segmentation of the caudate nucleus from a 3D MR brain image. (a) Initial 3D slice painted level set surface. (b) Final segmentation result. 30
4.7 Editing by erasing a portion of the cloverleaf segmentation on a 3D slice plane using a thin cylindrical paint brush tip. 31
4.8 Editing on a 3D slice by adding to the segmentation of the vertebra. 31
4.9 Erasing paint (i.e. the segmentation labels) on a 3D slice of a caudate nucleus segmentation. 32
4.10 A portion of the lateral ventricle segmentation is corrected using a small paint brush tip. 32

List of Appendices

1 Level Set Formulation **35**

Chapter 1

Introduction

Medical imaging currently plays a critical and expanding role in a host of clinical applications, from disease diagnosis and subsequent treatment planning, to surgical and radiotherapy planning, and even to intraoperative surgical navigation. For this reason many techniques and algorithms have been developed over the years to visualize, process and analyze 3D medical data, such as CT scans and MRI scans. Advances in medical imaging technology have resulted in the generation of massive volume images containing hundreds of high-resolution image slices. Therefore, efficiently visualizing and processing these data sets has become extremely computationally expensive.

Fortunately, graphics processing unit (GPU) technology continues to rapidly evolve. While a CPU consists of a few cores supporting complex instruction sets and optimized for serial processing, the GPU architecture consists of hundreds of simpler cores optimized for performing repetitive and independent data processing tasks in a massively parallel manner. Although GPUs were initially designed to efficiently render surfaces consisting of millions of polygons by taking advantage of the inherently parallel nature of polygon vertex and pixel operations, recent generations of graphics cards can now also be used as general-purpose parallel computing platforms and support programming for the GPU using high-level programming languages. The result of these advances in GPU programmability is the ability to not only perform real-time surface rendering but also real-time volume (image) rendering. Furthermore, the general purpose computing capability of GPUs, known as GPGPU, is well-suited for efficient volume image processing. Medical volume images are represented as a 3D grid of voxels (i.e. volume elements), the logical 3D extension of 2D pixels (i.e. picture elements). Many volume image processing algorithms are data-parallel and require repeating operations on individual voxels or on a small local neighborhood of voxels.

Graphics hardware is traditionally organized to render polygonal surfaces in stages, where the stages form a rendering "pipeline". For example, some stages are designed to transform vertex positions into various coordinate systems, while other stages break up (i.e. rasterize) polygons into fragments[1]. A fragment processing stage can, for example, blend fragments together based on their color. As men-

[1]Fragments are often considered as "potential pixels" since they may or may not appear as visible screen pixels, depending on operations performed on them.

tioned, GPUs are now programmable and most of the rendering stages can have their fixed functionality overridden by executing custom programs called "shaders". Many GPU-based volume visualization programs have been developed in recent years that provide real-time volume rendering by programming the fragment shader stage of the rendering pipeline to accommodate the 3D grid structure of a volume image. However, few of these programs are able to seamlessly and simply integrate general volume image processing algorithms, such as image segmentation, into the interactive volume rendering pipeline. Recently, a new programmable pipeline stage has been made available on graphics hardware called a "compute shader". Compute shaders can execute general purpose numerical calculations and can be flexibly inserted into the rendering pipeline. In this thesis we describe extensions to an existing software framework that tightly integrates interactive medical volume rendering with volume image processing capability by utilizing compute shaders. The volume visualization and volume processing integration is achieved by adding compute shader modules and other modules to the existing volume visualization framework (14). This previous framework, which in turn was based on a well-known open source visualization framework called ImageVis3D[2] (15), provided the ability to flexibly combine both volume rendering of 3D medical images with surface rendering of polygonal meshes. This feature is used to support the ability of the user to create surface envelopes, using a "painting" style interface, that defines a 3D region within the volume image.

1.1 Contributions

The primary goal of this thesis was to create a complete software framework for interactive volume image visualization and volume image processing that harnesses the massively parallel computational and rendering power of modern graphics hardware. The volume visualization and volume processing capabilities should be seamlessly integrated into a single graphics pipeline to maximize the volume exploration and visualization work-flow. A secondary goal was to add an interactive editing capability to the framework that supports the editing of the volume image outputted by the processing algorithms. The following contributions were realized in an effort to meet these goals:

1. The extension of an existing GPU-based volume visualization framework(13),(14) with GPU-based modules for performing volume image processing algorithms utilizing state-of-the-art GPU programs known as "compute shaders". The existing visualization framework provided a flexible surface and volume rendering pipeline with a "front-end" "painting" based user interface for interactively defining 3D regions of interest. The compute shaders were designed such that volume image processing algorithms can be initialized, executed and (optionally) their output displayed entirely within this pipeline.

2. The implementation of an example volume image processing algorithm that performs volume image segmentation. Segmentation is a necessary component for visualizing noisy volume images and for performing volume image analysis. The segmentation algorithm is a variant of the well-known

[2]ImageVis3D is a volume rendering project developed by NIH/NCRR Center for Integrative Biomedical Imaging. It is written in C++, using the Boost library.

level-set algorithm(28). The compute shader-based implementation, written in a high-level GPU programming language, executes in a massively parallel manner and provides significant speedup compared to an equivalent CPU-based implementation using MATLAB[3](31). The GPU based implementation was designed such that the performance of the segmentation algorithm will naturally continue to improve as the number of GPU cores increases through graphics hardware evolution.

3. The implementation and integration of an interactive image editing capability for editing image regions defined by the segmentation algorithm. The editing capability extends the framework's existing intuitive front-end painting interface, allowing the user to use the same interface for both painting an initial 3D region and subsequently editing the refined region outputted by segmentation algorithm. The implementation of the editing capability involved modifying a GPU rendering shader in the existing framework such that a user-controllable 3D image slice is rendered together with a 3D rendering of the volume. This allows the user to edit 3D regions in a slice-by-slice manner, providing simple and precise editing control and the ability to paint and edit regions on noisy volume images.

1.2 Thesis Outline

Chapter 2 reviews alternative application program interfaces (APIs) for GPU programming and for GPGPU programming. A brief review of volume rendering algorithms is then presented, followed by a survey of GPU-based volume image processing algorithms, including segmentation algorithms. Finally a summary of techniques researchers have used to interactively define and edit regions of interest (ROIs) within a 3D image is presented. **Chapter 3** presents a detailed description of the extensions made to the existing visualization framework. The implementation and integration of the various compute shaders is described. In particular, the compute shader implementation of the specific variant of the level set segmentation algorithm used in this thesis is presented. Finally, the interactive 3D slice editing mechanism utilizing the existing painting interface is described. **Chapter 4** demonstrates the extended functionality of the visualization framework. Several synthetic, artificial, and real medical data sets are used as input to the level set segmentation algorithm. The results of the segmentations are visually validated and the algorithm speedup achieved is discussed. A demonstration of the editing capability is also presented. **Chapter 5** presents conclusions as well as suggestions for improvements and additions to the framework.

[3]MATLAB is a computer program development environment that is primarily geared towards the development of numerical programs.

Chapter 2

Literature review

In this chapter, we present a brief review of literature related to GPU programming in general, as well as its application to volume image processing and volume image rendering. The chapter concludes with a section on volume image editing.

2.1 GPU Programming and the Graphics Pipeline

Rendering is the process of creating an image, using computers, from a group of 2D or 3D models placed in a scene. In recent years, this process has been executed almost entirely on the graphics hardware, with more and more control of the process made available to the programmer. That is, the graphics hardware is structured such that the rendering process is executed in a staged pipeline fashion (Figure 2.1) and many of the stages are now programmable. Two of the best-known application programming interfaces (API's) that implement the pipeline are the Open Graphics Library(55) (OpenGL) and Microsoft Corporation's Direct3D(2). OpenGL is an open-standard API for rendering 3D vector graphics and interacting with the GPU and is used in this thesis.

Rendering stages in the OpenGL pipeline can be programmed using OpenGL's C-style programming

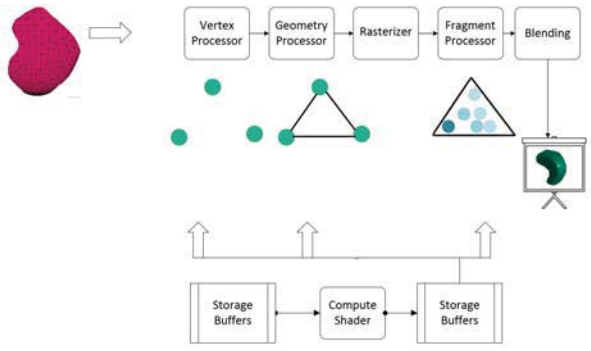


Figure 2.1: Simplified diagram of a modern 3D graphics pipeline.

language GLSL (OpenGL Shading Language). These programs are commonly referred to as "shaders". Common shaders include the vertex shader, geometry shader, and the fragment shader. Initially, the graphics architecture was designed such that separate custom processors were reserved for vertex shaders and fragment shaders. The hardware quickly evolved, however, resulting in a "unified" shader architecture that provided one large grid of general data-parallel floating-point processors. This hardware advance coincided with the emergence of general purpose computing (GPGPU) on the graphics card. GPGPU programs are not graphics programs but rather are used to solve general numerical algorithms or image processing algorithms in a massively parallel manner. To support the creation of GPGPU programs, two API's have emerged in recent years. CUDA(34) is a parallel computing platform and programming model created by NVIDIA Corp. It is supported on NVIDIA graphics cards and allows programmers to write GPGPU programs using a C-like language. OpenCL(56) (Open Computing Language) is an open standard for general-purpose parallel programming on GPUs, as well as other processors.

It is possible to mix CUDA programs and OpenGL programs. The results of the computation output by the CUDA program can be inserted into the rendering pipeline via a memory buffer. However, this process involves several steps including mapping and unmapping of a buffer into formats understood by CUDA and by OpenGL. In 2012, a new stage of the OpenGL graphics pipeline, called a"compute shader"(55), was released. A compute shader provided the general-purpose computation functionality of CUDA but was designed such that it can be, if desired, more tightly and seamlessly integrated as a new stage of the rendering pipeline. Compute shader programs are flexible. They do not require vertices as input, as in a vertex shader, or fragments as in the fragment shader; they can use any input and output type. Unlike the fixed vertex-geometry-fragment shader execution order, a compute shader can be inserted into the pipeline in a much more flexible manner (Figure 2.1). Furthermore, any resource that can be made available to a specific rendering shader is available to a compute shader. A compute shader's design and versatility makes it a more suitable mechanism by which to integrate computational tasks that may have some sort of visual output, such as volume image processing, into a visualization system. For these reasons, this thesis adopts the compute shader approach for the processing and visualization of 3D medical data.

2.2 GPU-based Volume Rendering

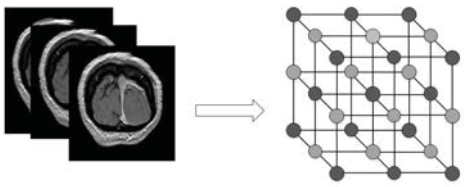
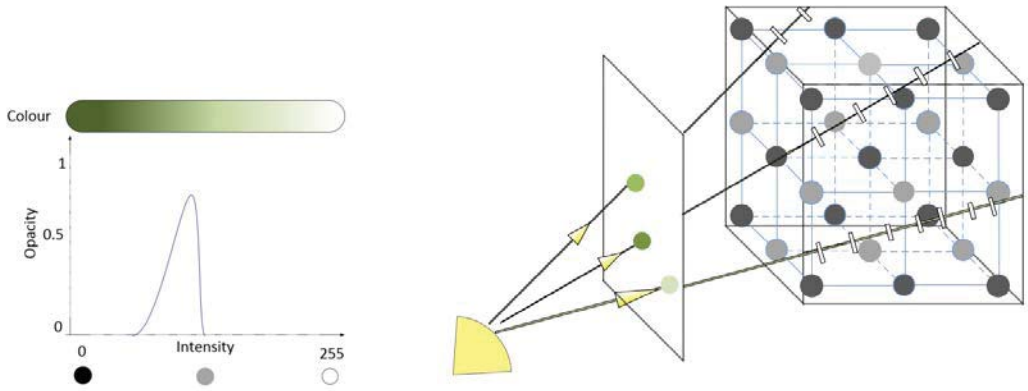


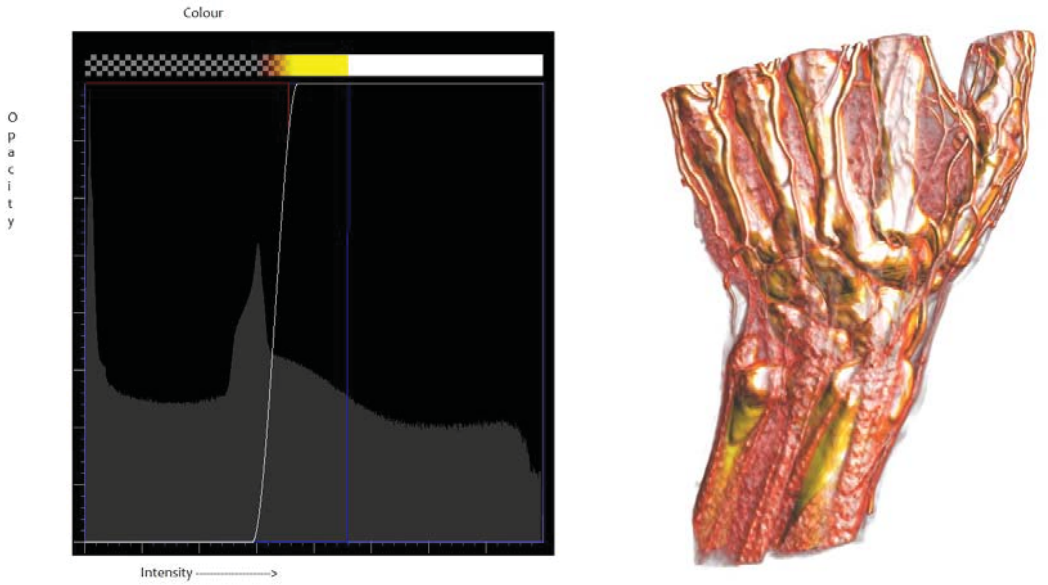
Figure 2.2: A 3D medical image depicted as a stack of 2D slices and as a 3D grid of voxels. Each voxel stores an intensity value.



(a) (b)

Figure 2.3: Depiction of volume ray casting. (a) Example of a transfer function for mapping intensity values to a color and opacity. (b) Ray casting into the volume and sampling at regular intervals along the ray.

As mentioned in the introduction, medical volume images are represented as a 3D grid of cubical elements called voxels (Figure 2.2). Each voxel stores an intensity value representing a sample of a continuous three dimensional signal generated by a medical imaging device such as a CT or MRI scanner. Volume rendering is a set of techniques for generating a 2D image from a volume image. The two main categories of these techniques are iso-surface rendering and direct volume rendering (DVR). Iso-surface rendering extracts points in the volume image with intensity values equal to a user-defined threshold value and uses them to form a mesh of triangles, which can then be rendered using the standard surface rendering pipeline. The marching cubes algorithm(29) is the most widely known and widely utilized Iso-surface rendering algorithm due to its simplicity and highly parallel nature. Unlike Iso-surface rendering, which considers only points in the volume that are equal to a threshold value, DVR (10),(27) considers the entire volume as a material that interacts with and emits light according to a volume rendering integral representing a physically realistic model of light absorption, emission and scattering (see (17) for a detailed treatment of DVR). For this reason, this approach generates high-quality images but is computationally intensive and the volume rendering integral is discretely approximated. One of the most well known DVR techniques that computes a discrete approximation to the integral is volume ray casting (27). In this image-based technique, a ray is cast from the eye through each screen pixel into the volume image (Figure 2.3). The ray is sampled at regular (or adaptive) intervals as it passes through the volume. For efficiency, the ray-volume entry and exit intersection points are calculated and used as the start and end points of the sample point generation. At each sample point, a volume-image intensity value is determined using interpolation - typically tri-linear interpolation. A user-controllable transfer function is used to map (or classify) the intensity value to a specific color and opacity value. The transfer function allows the user to make target anatomical structures or regions more visible or highlighted, and conversely to make background structures less visible (Figure 2.4). The next step is to compute the gradient of the intensity field at the sample point. The gradient is the normal vector of a



(a) (b)

Figure 2.4: Example of Direct Volume Rendering (DVR) using a CT image of the hand. In (a) the user maps volume intensity values to a color and opacity.

corresponding Iso-surface of the volume image and is a measure of the orientation of the Iso-surface at the sample point. The mapped color value is altered (i.e. "shaded") according to a simplified scattering (i.e. reflectance) model of light emanating from outside the volume. The normal vector, color value, eye position, and external light source position(s) are used in the calculation. The shaded color and opacity values at the sample point are then added into the current accumulated color and opacity of the ray sample points via a compositing operation. The entire process is repeated until the ray exits the volume. The composition of the sample point colors and opacities along the ray represents the discrete approximation of the emission and absorption terms in the volume rendering integral.

As may be inferred from the above description, the volume ray-casting algorithm lends itself well to a GPU-based implementation The generation and processing of ray sample points is the same for all rays and rays are independent of each other. Furthermore, the image-based nature of the algorithm suggests the use of a fragment shader. The entire volume can be stored into a 3D texture buffer[1]. A quadrilateral "covering" the entire screen window can be rendered by a volume visualization program's main rendering routine. The quadrilateral is rasterized into fragments where each fragment represents a screen pixel. Each fragment shader invocation handles one fragment and each fragment is associated with a single ray. The fragment shader uses the 3D texture, as well as other input parameters, and executes the ray-casting algorithm. The collective output of the fragment shader threads forms the final 2D output image. Volume ray casting, implemented as a fragment shader, can execute in real-time for reasonably sized data sets (14).

[1] A texture, in the context of OpenGL, is a memory buffer that is optimized for storing images and that typically resides in GPU memory.

Along with a GPU-based implementation, researchers have proposed other performance improvements of the volume ray-casting algorithm (24; 44; 48; 38). For example, Kruger and Westermann (24) integrated acceleration techniques to reduce per-fragment operations in the form of early ray termination and empty space skipping. Hadwiger et al.(18) employs a two-level hierarchical representation of the volume grid to support object-order and image-order empty space skipping. Penner (38) utilizes multi-pass coherent frustum casting to achieve significant performance improvements.

2.3 GPU-based Volume Image Processing

Volumetric data sets continue to grow in size due to advances in scanning technology. It is increasingly labor-intensive for radiologists and technicians to perform a slice-by-slice examination of these massive data sets. It is also difficult to set up complicated transfer functions to clearly visualize structures of interest embedded within noisy volume images. As a result, highly automated and efficient image processing algorithms that can filter and label a 3D image are becomingly increasingly important. Image processing is a natural fit for data parallel processing since the algorithms are often inherently parallel, which often translates into a simple GPU implementation. Image pixels can be mapped directly to GPU threads and many image-processing algorithms access only a small local neighborhood of pixels. Examples of processing algorithms that have been implemented on the GPU include various filtering and denoising algorithms, such as Gaussian smoothing, median filtering and edge detection algorithms, interpolation algorithms, histogram estimation algorithms, distance transforms, and finally registration and segmentation algorithms.

For a recent and thorough survey of medical volume image processing on the GPU, the reader is referred to Eklund et al.(12). Only a few representative research works are referenced here. Examples of filtering operations implemented on the GPU are median filtering (59; 7; 47; 39), a convolution approach to fast cubic interpolation(40, Chapter 20. Fast Third-Order Texture Filtering) and convolution using Gabor filters(60), and an implementation of Canny edge detection(30). Shams and Kennedy(51) and Shams et al.(52) present algorithms for histogram estimation using CUDA. Schwarzkopf et al.(50) accelerate nonlinear anisotropic diffusion-based 3D image denoising using CUDA. Ruijters et al.(45) present a fast implementation for non-rigid registration between pre- and intra-operative CT volumes.

The majority of these GPU implementations utilize CUDA for the programming environment. In order to support fast and uninterrupted exploration and visualization of a large and noisy 3D image, it is highly desirable to more tightly integrate volume image processing into the view generation work-flow. This type of integration is especially important for volume image segmentation, which we present in the next section.

2.3.1 GPU-based Volume Image Segmentation

Segmentation partitions a 3D image into segments (i.e. sets of voxels) by labeling voxels belonging to the same anatomical structure or tissue class. The segmentation simplifies the representation of an image and allows for the measurement of surface and volumetric properties of an object. In addition, for very

noisy volumes, segmentation must first be performed before the anatomical structures can be effectively volume rendered and their visual appearance controlled via transfer functions. Segmentation is still an active area of research with a variety of techniques published over the years, including region growing, deformable models, graph cuts and watershed algorithms. This section presents some representative examples of GPU-based implementations for each of these categories. The reader is once again referred to Eklund et al.(12) for a more complete review.

Perhaps the simplest of the segmentation approaches is region growing. Beginning with a region (or regions) defined by user-selected seed voxels, the algorithm iteratively scans neighbor voxels, expanding regions if neighbor voxels meet some inclusion criterion, which is often based on image properties such as voxel intensity homogeneity. Region growing can be GPU accelerated by processing neighbor voxels in parallel, although care must be taken to avoid processing the same neighbor voxel at the same time by different GPU threads (37; 54; 5; 6). A very popular segmentation technique is the Level Set algorithm(35). The algorithm defines a contour (a surface in 3D) as the zero level set of a higher dimensional implicit function. The function is evolved according to equations depending on image characteristics and the zero level set itself. The function is iteratively evaluated on a 3D grid and is inherently parallel. Early GPU implementations are presented by Rumpf and Strzodka(46), Lefohn et al. (26) and Cates et al.(4), while more recent optimized implementations are those of Roberts et al.(43) and Sharma et al.(53). The active contour (or active surface) approach to image segmentation represents the contour/surface explicitly as a set of connected nodes, rather than implicitly as in the level set approach. Therefore, rather than parallelization based on 3D grid points, active contours can be parallelized based on the nodes themselves. The positions of each node, as well as the image forces acting on each node, are iteratively updated in parallel. Examples of GPU accelerated active contour algorithms are He and Kuester(20) and Schmid et al.(49). More recent GPU-based segmentation algorithms are based on graph cuts (Vineet and Narayanan(58)) and random walks (Collins et al.(9), Grady et al.(16), Top et al.(57)).

While no single technique or algorithm has yet emerged that that can solve all segmentation problems, fast and accurate GPU-based implementations can have a significant impact on interactive volume image exploration, especially for noisy volumes. For those segmentation techniques that are interactive to some degree, the user can help the algorithm to provide a satisfactory result. For example, one of the main problems with level set methods is the difficulty in fixing an incorrect solution. Therefore it is highly desirable to integrate a simple, intuitive interaction model into the volume rendering and volume segmentation pipeline so that the user can quickly and simply initialize (and optionally constrain) the algorithm - mitigating the need for subsequent editing. However, if editing of the segmentation output is needed, the interaction model should support this operation, ideally using interactions similar to initialization. In the next section, we review interaction mechanisms for initializing and editing semi-automatic segmentation algorithms.

2.4 Interactive Volume Image Segmentation

Although image segmentation may be performed automatically, poorly defined boundaries, image noise and sampling artifacts due to limited image resolution, often cause the algorithms to generate erroneous results. Semi-automatic segmentation techniques, on the other hand, are designed to allow a medical expert to initialize, steer, and edit the algorithm. Initialization of interactive segmentation techniques are commonly in the form of contour delineation (19; 1), the planting of "seed" regions (4) or "painting" of 2D strokes(16; 58; 41; 61) or regions (6), or the "sketching" of regions (5; 42). The input operations are often performed on 2D image slices where the image slices are rendered in separate windows and with the slices in a standard orientation (i.e. axial, sagittal, coronal). An alternative is to draw directly on the view plane (36; 42). The marking and editing of contours and regions in a 2D window requires this input to be visually integrated back into the 3D volume-rendered view of the data. However, many of these techniques are constructed using a visualization package that commonly support interaction on 3D slice planes that can be arbitrarily oriented and displayed together with the volume rendered data.

The output of most segmentation algorithms is a set of labeled voxels representing the target structure. The exception is active contour algorithms that output a contour (or surface in 3D). However, there are fast, GPU-based voxelization algorithms that can label the voxels inside the contour (11). Segmentation errors typically consist of either the mislabelling of regions outside of the target structure as voxels belonging to it (i.e. the segmentation has "leaked" into neighboring structures), or not labeling voxels that are, in fact, part of the target structure. Editing of the segmentation output therefore typically requires either the deformation of a contour, or the interactive painting/erasing of voxels. A good example of editing via contour deformation is the work of Ijiri and Yokota(22). In this work, a boundary surface surrounding the labeled voxels is generated via Marching Cubes. Image slice planes, or image slice surfaces, are used to examine cross-sections of this boundary surface with respect to the region around the target structure. That is, boundary-surface contours are created as the intersection of the image slice plane and boundary surface. The user can repair mistakes by deforming these contours. The deformed contours are then used as constraints to smoothly deform the boundary surface. Heckel et al.(21) employ sketch-based editing on image slices where contours representing the boundary of labeled voxel regions can be cut away or added to by drawing a new boundary segment. Kang et al.(23) presented three interactive editing tools, applied on 2D image slices: a hole filling, a point-bridging tool and a surface-dragging.

Another common form of editing is to use a "paint brush" metaphor and manually add paint to the labeled region or erase existing paint. This operation is primarily performed on a 2D slice, on a slice-by-slice basis. We adopt this simple strategy in this thesis. This editing functionality is detailed in the next chapter.

Chapter 3

Methodology

In this chapter, we describe the various GPU-based modules that we added to extend the existing interactive volume visualization framework. "Seamless" integration of these modules was one of the primary design factors. The new modules consisted of the following:

1. Compute shaders providing basic volume image filtering in the form of Gaussian smoothing and edge detection.
2. A compute shader that implements a variant of the level set segmentation algorithm(28).
3. Compute shaders and modifications to an existing fragment shader to extend the existing framework's 3D ROI painting mechanism.

3.1 The Existing Framework

(a) (b) (c)

Figure 3.1: Exposing bones in a CT hand data set by painting and blending a series of superellipsoid brush strokes to form a 3D region of interest.

The existing framework was written to support user interaction with medical 3D data through a painting metaphor. The user can paint a 3D envelope directly on iso-surfaces of the volume rendered data in the 3D rendering window, creating a 3D region of interest (ROI) (Figure 3.1). The appearance of the

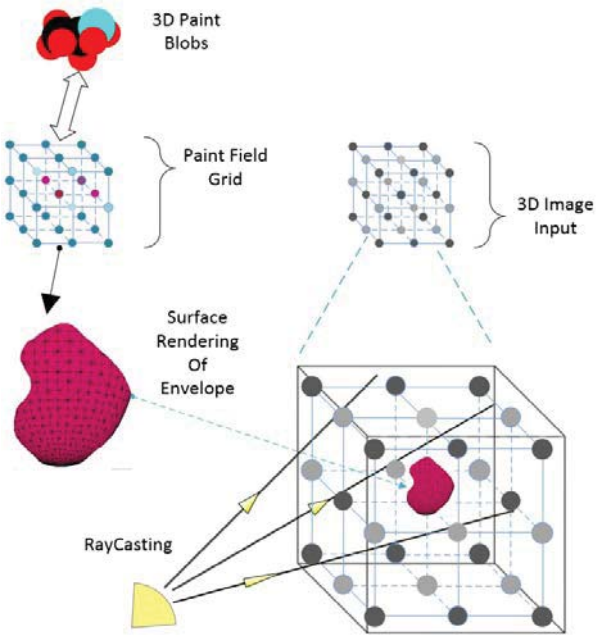


Figure 3.2: High-level overview diagram of the existing framework.

volume rendered data inside the envelope is controlled using a separate transfer function. ImageVis3D(8) supports 1D and 2D transfer functions. Faynshteyn(13) also implemented a visualization algorithm called Maximum Intensity Difference Accumulation (MIDA)(3). MIDA was proposed by Bruckner et al.(3) to quickly visualize volumetric data without the use of complex transfer functions. MIDA changes the accumulating properties of the conventional Direct Volume Rendering (DVR) approach. During ray casting, the opacity along the ray can quickly accumulate and local maximums depicting interesting regions are hidden behind opaque layers. In MIDA, local maximums are not occluded and the opacity-accumulation technique is modified to capture interesting regions even if they exist further along the ray. In the existing framework, to use MIDA, the user simply specifies a minimum and maximum intensity range using GUI sliders. Only features within this range are rendered, exposing internal structures (Figure 3.1c). Furthermore, a MIDA base color can set by the user, if desired, and mapped to the voxels within the defined intensity range. In this way, the user can highlight structures inside the envelope.

The 3D paint in the painting interface is realized by blending a set of shape primitives, where each primitive is defined using a superellipsoid implicit function(32). With superellipsoids the user is able to create "blobs" of paint or "brush strokes" of paint of different shapes, sizes and thicknesses such as spheres, cylinders and rectangular blocks. Furthermore, the user can use a mouse or other input device to slide a "brush-tip" paint-blob along iso-surfaces of the data in the volume render window. In this way, the brush tip acts as a lens supporting real time exploration of the medical data. Figures 3.2 and 3.3 illustrate the existing framework, as implemented in (14) and enhanced in (32). When the user deposits

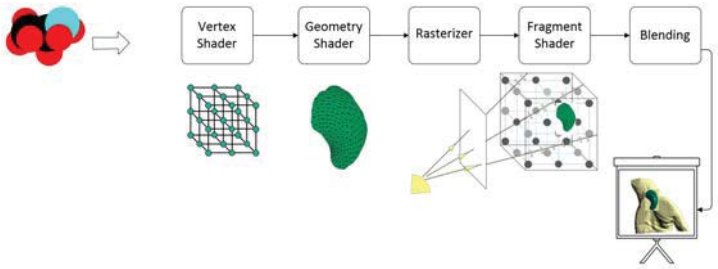


Figure 3.3: Graphics Pipeline showing the stages of the existing framework.

a blob or brush stroke of paint, an implicit superellipsoid shape primitive is defined and added to an array. The shape primitives in the array are blended in a vertex shader to collectively define an implicit scalar field function, referred to as "thick paint"(14). The vertex shader samples the scalar field function and stores the samples at grid points on a 3D rectangular grid, referred to as the *Paint Field Grid* in Figure 3.2 . Using this grid as input, the Marching Cubes algorithm (29) is run in a geometry shader to generate the 3D polygonal surface envelope. The surface envelope rendering and volume rendering via ray casting are then performed together in a fragment shader. As mentioned above, separate transfer functions are available to control the appearance of the volume rendered data inside and outside the envelope. The ability to mix surface and volume rendering is illustrated in Figure 3.2 by the pink surface envelope within the intensity volume grid.

3.2 The Extended Framework

Volume image processing capabilities have been added to the existing framework to extend it. Seamlessly integrating these capabilities into the interactive volume exploration and visualization work flow is, in part, achieved by reusing and extending the 3D painting interface. In the extended framework, 3D painting can be used for defining 3D regions of interest as an optional input to any volume image-processing algorithm. Figure 3.4 shows the extended framework, with the left side of the dotted line showing the extensions added in this thesis. Modules implementing computationally expensive volume processing algorithms are built using compute shaders. Basic compute shader programs are designed to accept a volume image grid and algorithm global parameters as input. The shaders iteratively execute the algorithm and generate an output grid. This output grid can then be optionally used for volume rendering, if desired. The upper middle and upper left part of Figure 3.4 shows an optional input grid generated from the result of painting a 3D ROI. More complex compute shader programs, such as the level set segmentation compute shader, can make use of this input. In the case of the level set segmentation algorithm for example, the algorithm refines the 3D region and labels this region in an output grid (Figure 3.4 lower left). The output grid can then be optionally input to a geometry shader where the marching cubes algorithm will generate a boundary surface representation of the labeled 3D region. This envelope surface is treated in exactly the same manner as a painted envelope in the

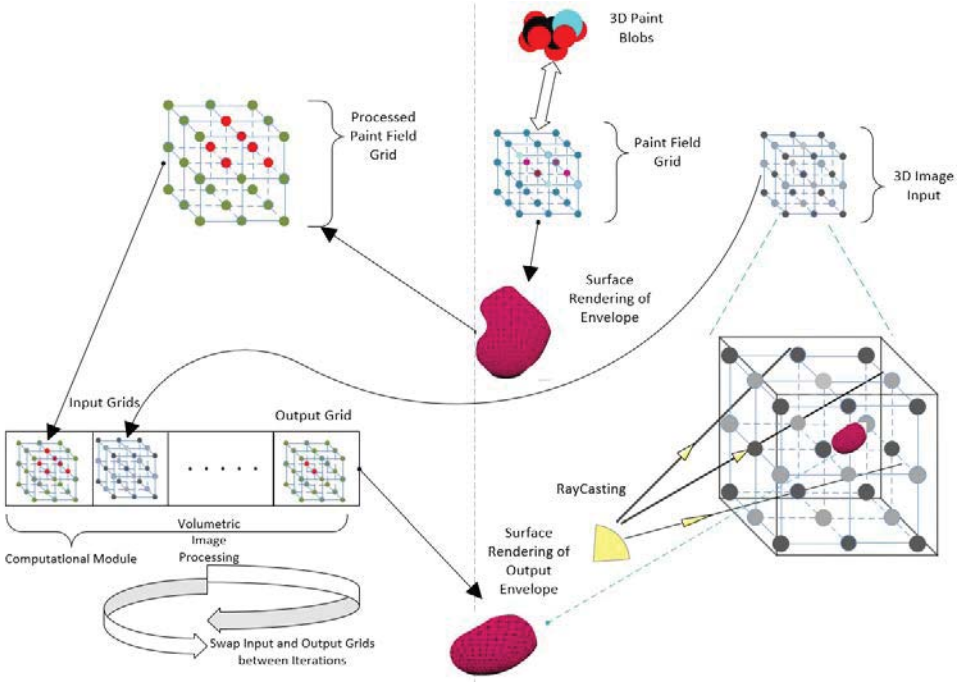


Figure 3.4: High level overview of the extended framework.

existing framework, with the same rendering options. Figure 3.5 depicts the graphics pipeline view of the extended framework.

3.2.1 Painting and Editing on 3D Slice Planes

In this thesis, we have also extended the 3D painting interface to support painting a 3D ROI on a 3D user-oriented image slice plane, also referred to as a clip-plane. In many visualization systems, a 3D clip plane is texture mapped using voxel intensity values interpolated from the input volume image. In this thesis, on the other hand, we alter the volume ray casting algorithm in the fragment shader to generate an edit plane that is rendered along with the volumetric data during the volume rendering stage (Figure 3.6), achieving the effect of a clipped volume rendering of the data. The volume ray casting is altered by computing the start of each ray from where it intersects the clip plane. These starting ray sample points ensure that everything in front of the plane is clipped away. The starting ray sample points are then used to look up the corresponding image intensity value in the volume image via interpolation. These volume image samples are mapped, using a simple transfer function, to a color and an opacity value and each corresponds to a fragment, which will appear as a screen pixel. The fragments are shaded using the normal vector of the clip plane rather than a normal vector computed from the volume image. If the fragments are mapped to an opacity equal to 1.0, the ray casting algorithm is terminated for this ray; otherwise, the ray casting algorithm continues as usual.

It is often not possible to use a TF to isolate and volume render target structures in noisy volumes,

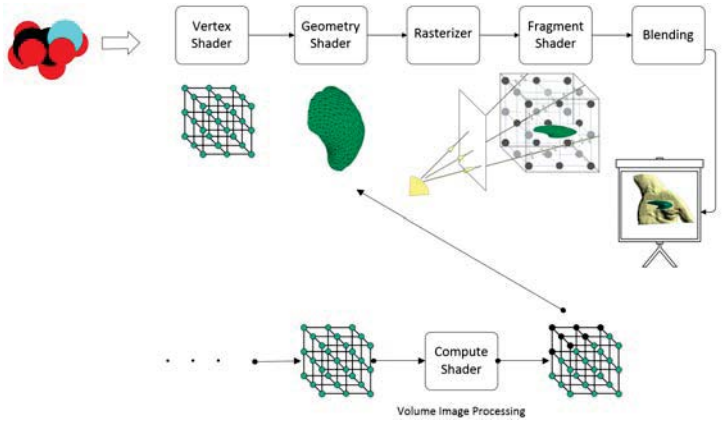
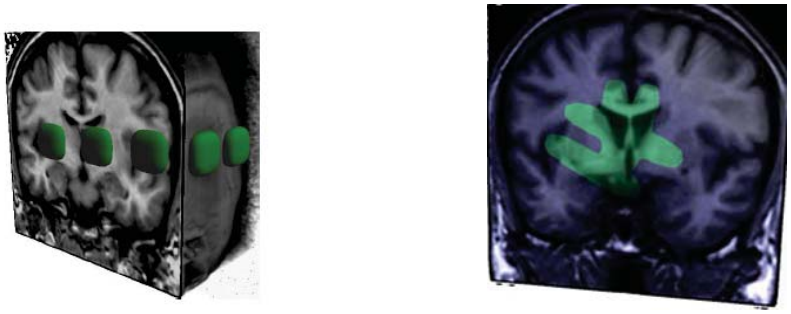


Figure 3.5: Graphics Pipeline showing the stages of the extended framework.



(a) (b)

Figure 3.6: Painting on a 3D slice plane. (a) Paint brush "tip" blob, rendered as an opaque surface, can slide along a slice plane that clips the volume rendered data. (b) Application of paint brush strokes on the slice plane. The 3D surface of the painted envelope is transparent, and a simple transfer function highlights the voxels inside the envelope and on the slice plane.

preventing the use of direct 3D volume painting. This situation also occurs when the target structure is adjacent or connected to neighboring structures with similar intensity characteristics. In these cases, the user can use the slice-plane painting approach, along with "flattened" superellipsoid paint blobs and paint strokes, to define a 3D ROI that envelopes a cross-section of a target anatomical structure. The thickness of the flattened paint blobs can be set by the user to range from a single slice thickness to many slices thick, providing the user with precise control over the envelope thickness. The user can paint thick envelopes (i.e. several image slices thick) on several cross-sectional slices of the target structure such that these slice-painted envelopes overlap. The slice-painted envelopes are automatically blended to form a single envelope tightly bounding the entire target structure. Chapter 4 presents examples of this slice-painted envelope capability.

We have also added the capability to edit a labeled 3D region, via the 3D slice plane painting approach. A segmentation algorithm, for example, generate 3D grids with labeled regions. The labeled region can be edited by using the painting interface to erase parts of the region or to add "edit" paint to

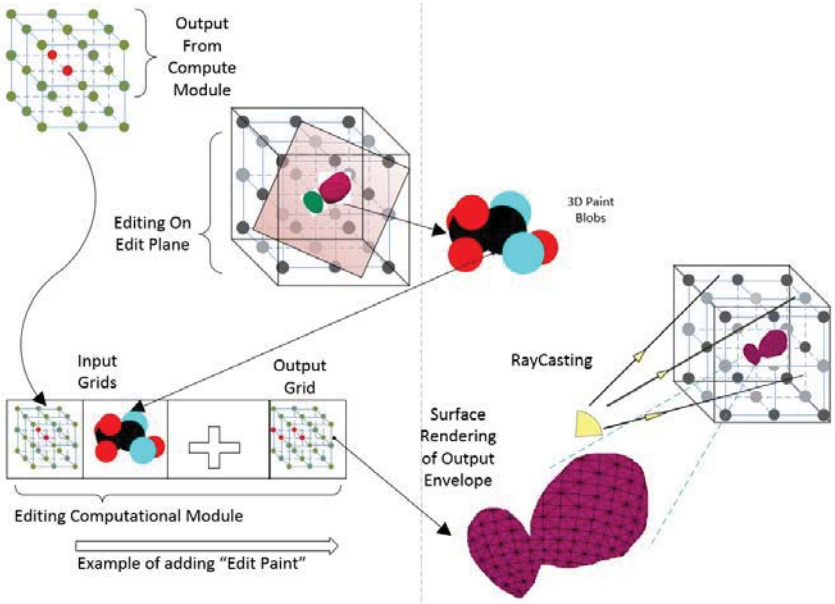


Figure 3.7: High Level Overview of editing a labelled 3D region via 3D slice paint painting.

the labeled region. An illustration of this process is shown in Figure 3.7. The left side of the dotted line depicts the slice plane painting and editing extensions. A special volume image-processing algorithm was implemented in a compute shader to perform the erasing and adding of labels to the voxels within the 3D region. This algorithm is described in Section 3.5.

As mentioned previously, the appearance of the voxels within the labeled region can be controlled with a separate transfer function, allowing these voxels to be volume rendered using a distinctive "highlight" color. The altered volume ray-casting algorithm shades the color of voxels that are on the slice plane using the normal vector of the slice plane. The color of the labeled voxels is a blend of the highlight color and the color of the voxel intensity value assigned via the transfer function. The result is the labeled region voxels appear as semi-transparent and highlighted 2D "paint". Furthermore, the user can dynamically set the superellipsoid paint blobs and paint brush strokes thickness to be just thicker than a single slice plane. The surface of the paint blob can be made completely transparent. Any voxels on the slice plane that are inside the blob can be made to appear semi-transparent and highlighted in 2D paint (Figure 3.6b). Thus, this special slice-plane rendering capability gives the user the illusion of erasing and adding 2D paint to the labeled 3D region. This visual effect is easily understandable by the user and allows the user to see the boundaries of the target structure underneath the semi-transparent 2D paint of both the labeled region and the paint blob. Corrections can be made to the labeled region on the current slice plane. The user can then continue to another oriented slice plane to make further corrections. While this slice-by-slice editing via painting approach can be tedious, especially if many slices require correcting, it is simple and allows the user to learn only a single painting interaction model.

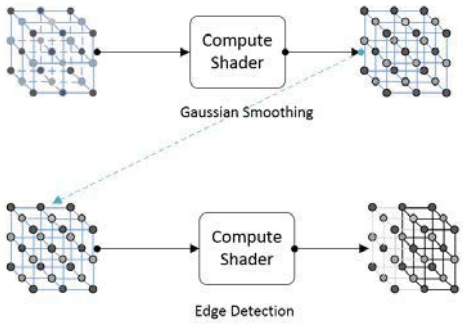


Figure 3.8: Overview of basic image processing pipeline.

3.3 Image Processing: Image Filtering

Basic volume processing algorithms, such as filtering algorithms, that are commonly used in medical imaging can be "plugged into" the extended framework as long as they are parallelizable at the 3D grid point level. The image processing filters are implemented using compute shaders. Furthermore, the filters can be cascaded - the output of one filtering stage can be used as input to the next (Figure 3.8. We have implemented Gaussian Filtering to smooth a volume image. We have also implemented edge detection using a simple image gradient magnitude calculation. The output of these cascaded image filtering stages are used as input to the level set segmentation algorithm described in the next section. We have used a modular approach in the design of our compute shaders for volume image processing algorithms. We have consistently used buffers to store 3D input and output scalar grids, where the scalar values stored at grid points can be image intensities, processed image intensities or any other field value. In particular, we have used Shader Storage Buffer Objects (55) as buffers, using their binding points as input or output hooks. By virtue of their binding point, we can use the same buffer as an input or output buffer, thereby allowing different shaders to pick up the same buffer and process them as they see fit. This strategy maximizes efficiency on the Graphics Processing Unit (GPU) by avoiding copying or moving data around. Other global filter-specific parameters can be passed into a filter compute shader, such as a kernel matrix.

3.4 Image Processing: Level Set Segmentation

We have chosen level set segmentation as our showcase volume image-processing algorithm for several reasons. Segmentation is a necessary step when exploring and visualizing noisy volume images, and is a requirement for subsequent volume image analysis. The level set algorithm is powerful and flexible and can segment geometrically and topologically complex objects. The algorithm is defined on a 3D grid and is highly parallelizable. Finally, the algorithm fits very well with the existing 3D painting interface. In this section, we describe the compute shader based implementation of the level set algorithm.

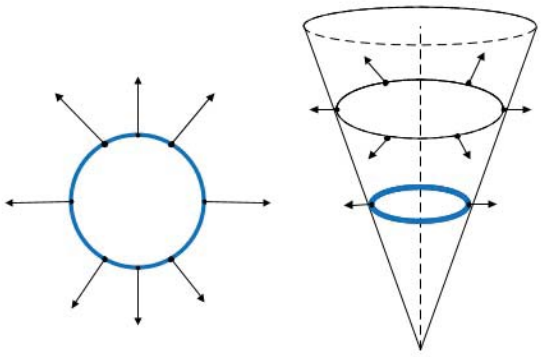


Figure 3.9: Depiction of level set evolution in 2D.

3.4.1 A Brief Look at Level Sets

In the level set method, a surface (in 3D) is defined to be the zero level set of a continuous function, $\phi(t, x, y, z)$. The movement of the level set surface is governed by an evolution equation of the level set function ϕ. The evolution equation contains terms defined as the gradient of an energy functional, where the energy functional depends on the image data. These *external energy* terms drive the level set surface towards object boundaries. There are also *internal energy* terms that are a function of ϕ only and that fundamentally act to minimize the surface area of the level set surface. At any point in time, we can recover the location of the level set surface from ϕ, by looking for points where ϕ takes on a value of zero. To illustrate this idea, in Figure 3.9 a 2D level set contour is shown. The contour is embedded in the conical function, ϕ, and evolved over time. At each time step, we can construct the embedded contour by extracting the points where ϕ is zero. The level set method, therefore, extracts the level set surface representing the boundary of the target anatomical structure and the function *phi* defines its interior. The method is initialized by creating an initial surface envelope that either loosely surrounds the target structure or is contained inside it. An initial level set function ϕ_0 is constructed from this envelope. A brief mathematical overview of the specific level set algorithm used in this thesis (28) can be found in Appendix 1.

3.4.2 Initial Level Set Construction

As the level set function evolves, it can develop shocks or very sharp corners, which cause numerical inaccuracies in a software implementation of the evolution function. A common technique to deal with this problem is to initialize the function ϕ as a signed distance function and then re-initialize it periodically during the evolution. We follow the level set formulation described in Li et al. (28) which eliminates the need for this re-initialization process. In this thesis, we use the painted envelope to construct our initial level set function. Recall that the envelope is defined as a blended set of paint

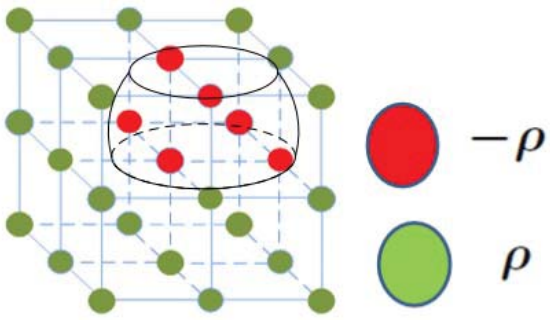


Figure 3.10: Illustration of initial level set. Values inside the VOI are marked as $-\rho$ and outside the VOI are marked as $+\rho$, where ρ is a constant (28).

blobs, where each blob is represented using a superellipsoid implicit function. From Li et al. (28) the initial level set function, ϕ_0, is defined as:

$$\phi_0(x,y,z) = \begin{cases} -\rho & \text{if}(x,y,z) \in \Omega_0 - \partial\Omega_0 \\ 0 & (x,y,z) \in \partial\Omega_0 \\ \rho & \Omega - \Omega_0, \end{cases} \tag{3.1}$$

where Ω is the volumetric domain, Ω_0 is a subset of the volumetric domain containing all points inside the painted envelope and $\partial\Omega_0$ is the set of all points exactly on the boundary of Ω_0 (i.e. the painted envelope boundary surface). As in (28), in our implementation we set ρ to a value of 6. The level set field function ϕ (including the initial level set function ϕ_0) is sampled at points on a regular 3D grid and is referred to as the ϕ-grid (and ϕ_0-grid) in this thesis. Typically the ϕ-grid dimensions are set equal to the input volume image dimensions. That is, to initialize the ϕ-grid, at each ϕ-grid point, we determine if the point is inside, outside, or on the painted envelope boundary using a point inside-outside function defined in McInerney and Faynshteyn (32). This initial level set function construction process is implemented in a compute shader, *levelset_init.cs*. The compute shader accepts the ϕ-grid as input as well as the array of paint blobs defining the painted envelope.

3.4.3 Overview of Level Set Implementation

The level set segmentation algorithm uses edge image features to determine if the evolving level set surface has reached the boundary of the target structure. Input volume images are commonly convolved with a smoothing filter to remove noise before performing edge detection. We use the Gaussian Filtering and edge detection compute shaders, described previously, to compute the edge detected image. The level set evolution is computed inside another compute shader, *updatephi.cs*. It takes an input ϕ-grid, along with the edge detected intensity grid outputted from the edge detection compute shader. It then

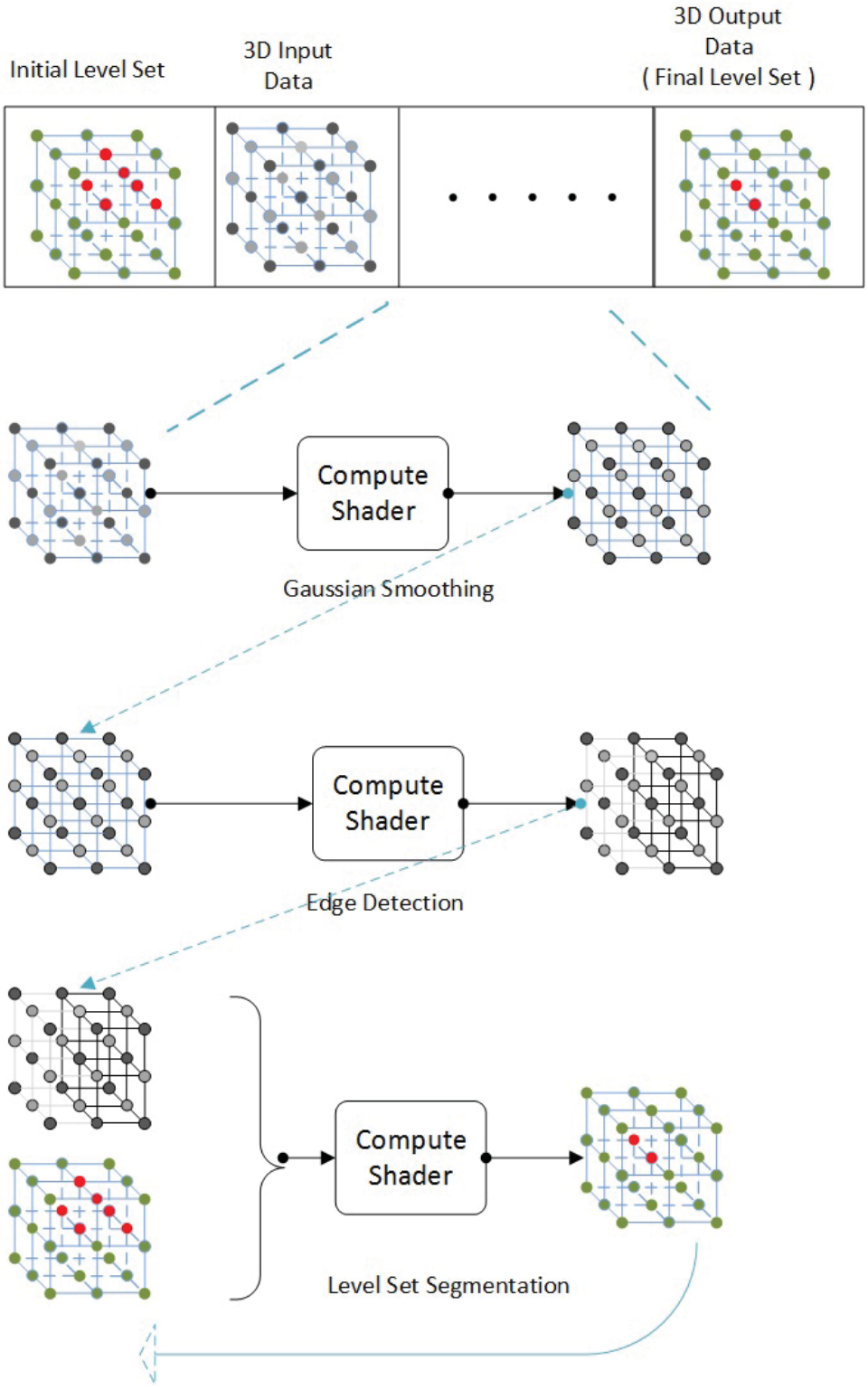


Figure 3.11: Overview of level set segmentation-processing pipeline.

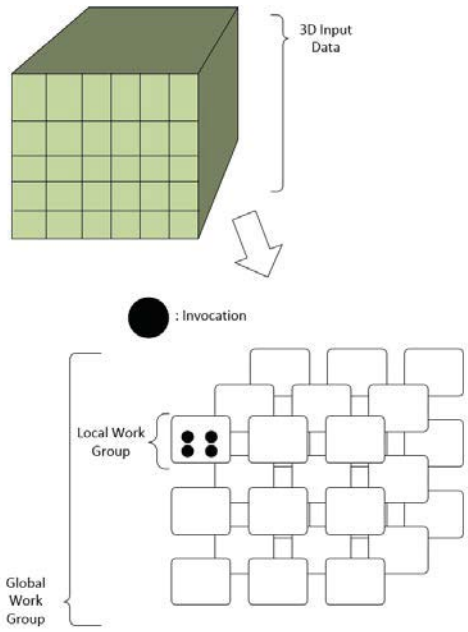


Figure 3.12: Parallel computing in a compute shader. A computational problem on a 3D grid, exhibiting a parallel pattern, is broken down and computed in parallel on compute shader threads.

outputs a grid containing updated values of the function ϕ referred to as ϕ-gridOut. Initially, the input ϕ-grid is the ϕ_0-grid (i.e. the ϕ-grid at time $t = 0$). Given the ϕ-grid at time t, the shader computes the evolved ϕ-gridOut at time $t + \Delta t$ using Equation 1.5, described in Appendix A. As mentioned previously, we use buffer binding indices as hooks to interchange the input and output ϕ-grid buffers, which avoids copying or moving the grids. That is, the input ϕ-grid becomes the output grid and the ϕ-gridOut becomes the input grid in the next temporal step (i.e. iteration).

The final output of the level set segmentation algorithm, ϕ-gridOut, contains scalar values. This output grid can be sent to a geometry shader that executes the Marching Cubes algorithm and generates a mesh of triangles representing the zero level set surface. As mentioned in Section 3.2, this envelope surface can be treated in exactly the same manner as a user painted envelope - it can rendered together with the volume rendered data and voxels inside the level set surface can be volume rendered using a separate transfer function. Figure 3.4.3 illustrates the level set segmentation processing pipeline. This figure expands on the section labeled "Computational Module" from Figure 3.4.

3.4.4 Compute Shader Implementation

We begin this section with a brief description of the types of memory available on the GPU. A GPU contains a few streaming multiprocessors (SM), which in turn contain a number of streaming processors or cores. An image-processing problem is generally broken down and performed on the smallest computational unit such as a voxel in a 3D grid or a pixel in a 2D grid. Computation on these units is performed by launching threads, with each thread processing a single unit (such as a voxel). Threads (also called

invocations) are grouped into a block (also called a local work-group) and a Streaming Multiprocessor (SM) can handle one or more blocks concurrently. A thread on the SM runs on a core in that SM. Each SM has a small number of registers, which store thread specific variables. Data is loaded from the Central Processing Unit (CPU) to the GPU and stored in global memory, which is typically quite large but has a relatively low bandwidth. Upon launch, each thread loads the data it requires from the global memory into its local registers. Upon finishing the computation, the data is written out to global memory. Each SM contains a small amount of memory (between 16KB and 48KB) which is available to the threads on that SM to share data efficiently with other threads in the same block. If threads within a block read the same data, the data can be read in once and collectively shared by housing it on the shared memory of the SM.

Modern GPUs have a general L1 and L2 cache to speed up reads from global memory. Cores on a SM are meant to execute the same instruction at the same time. However, in practice, threads are grouped together and the group executes the same instruction. That group of threads is called a *warp*, where the warp size on recent architecture is 32. Performance is optimized when a warp, running on a SM, uses data with nearby addresses. If a warp of threads is stalled, another warp can be immediately executed to hide the latency, swapping out the first one. Stalls can occur due to data cache access delays and instruction delays. If global memory access introduces some latency, this can be covered up by putting enough warps on each SM.

One approach to optimizing the performance of a volume image-processing algorithm that requires few iterations and that requires that each processed grid point access neighbor grid points, is an overlapping tile method. This method attempts to take advantage of the shared memory on a local work-group. Loading a "tile" of data grid points (e.g. an 8 x 8 x 8 region of grid point) that surrounds a smaller tile of grid points (e.g. 4 x 4 x 4) on the shared memory allows the threads to efficiently lookup values stored at neighbor grid points efficiently.

Another approach, used in this thesis for the level set segmentation compute shader, uses a simpler scheme. Shader Buffer Objects storing 3D grids of scalar values are stored as a contiguous one-dimensional array on the GPU. In a thread, we are able to use the thread id to compute a 3D grid point position. This grid point position can be flattened into a linear one-dimensional index into the 3D grid buffer. The thread then executes the level set evolution equation for its assigned grid point. Individual threads are grouped in a local work-group or block. The local work-groups together make up the larger global work-group (Figure 3.4.4). Shader Buffer Objects are stored in the L2 Cache and each thread looks up its required data from these buffer objects.

Stopping Level Set Evolution

In Kuo et al. (25), the authors mention several criteria for stopping the level set evolution. A stopping condition is typically evaluated after each iteration of the segmentation algorithm. We use a simple but often effective stopping condition in this thesis. The volume of the segmented 3D region is denoted V and the difference in the volume between the previous and current iteration is denoted ΔV. The evolution is stopped when $\Delta V/V$ falls below a small threshold value of 0.005. In keeping with the philosophy of

minimizing CPU-GPU intercommunication after the data is loaded on to the GPU for computation, we have allowed the extra iterations to run through on computationally empty threads once the stopping criterion is achieved.

We have utilized atomic operations, supported as of OpenGL 4.3, to implement the simple stopping condition. Atomic operations can be safely performed by shader threads running simultaneously where the threads are attempting to write to the same memory location. Atomic operations write to (or read from) memory uninterrupted; if multiple threads attempt to access the same location simultaneously, they will be serialized. We use atomic operations that operate on a special stopping condition buffer that stores the previous and current volume of the segmented region, as well as the number of grid points that have been processed.

When a thread begins executing at a current time step t (i.e. iteration), it checks the stopping condition ($\Delta V/V < 0.005$). If the condition is met, the thread returns. Otherwise the thread uses an atomic add operation to add 1 to the number of processed grid points. The thread then executes the level set evolution equation for its assigned grid point. If the ϕ function field value for this grid point is less than 0, we use the atomic add operation to add 1 to the current segmented region volume; that is, the number of voxels (i.e. grid points) inside the segmented region is used as a measure of the region's volume. When the thread finishes executing the evolution equation, it checks the number of processed grid points in the stopping condition buffer and determines if all grid points have been processed. If so, this thread sets the previous segmented region volume equal to the current volume and then resets the current volume and number of processed grid points to 0.

3.5 Editing a Labeled 3D Region

This section provides implementation details for the editing functionality. Editing the segmented region has been provided in the form of erasing and adding operations, followed by optional local blending or smoothing. These operations are implemented in special compute shaders. As mentioned in Section 3.2, the user can use the painting interface to erase or add labels to the labeled region that is output from the segmentation algorithm in the form of the ϕ-grid. The edit compute shaders accept the ϕ-grid as input as well as the array of paint blobs defining the edit region. In editing mode, the edit compute shaders use the painted blobs' inside-outside function to determine if a ϕ-grid point is inside the envelope formed by the blobs. When adding to a labeled region, the grid point values inside the painted blobs are set to $-\rho$, labeling them as part of the segmented region. Conversely, when erasing part of the labeled region, the grid point values inside the painted blobs are set to $+\rho$.

In Museth et al. (33), the authors define editing operators via the speed term in the general level set equations. Employing editing operators based on level sets has advantages such as avoiding boundary surface self-intersection issues and easily coping with topological genus changes. In this thesis, we use a simple version of the constructive solid geometry operations mentioned in Museth et al. (33). A remove or erase operation is analogous to the cut away operation in Constructive Solid Geometry (CSG) and the difference operation in set theory. Similarly, an add operation is equivalent to a union operation in

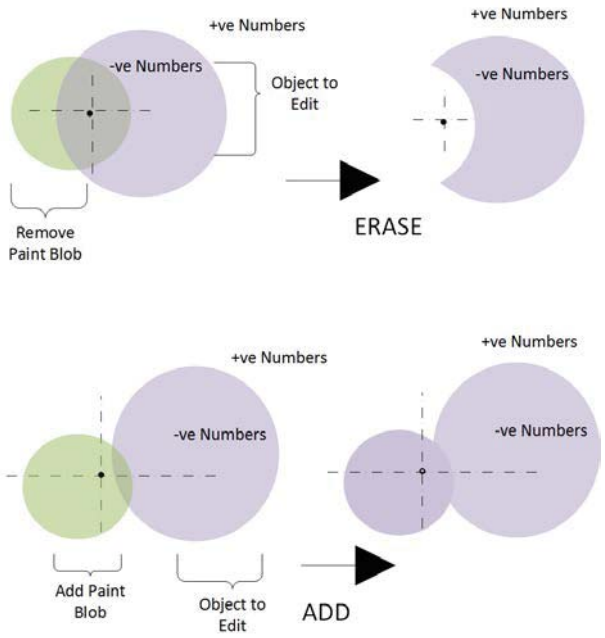


Figure 3.13: Conceptual Diagram of Adding and Removing Paint

CSG or set theory.

Both the erase and add operations are simply evaluated at the ϕ-grid points. For the add operation, at each ϕ-grid point, we read its field value and store it at the corresponding output grid point. Using array of painted blobs and blobs' inside-outside function, we then check if this output grid point is inside the painted region; if so, it is overwritten with a $-\rho$ value, making it a part of the labeled region. For the erase operation, at each ϕ-grid point, we we read its field value and set the corresponding output grid point to the same value. We then check if this output grid point is inside the painted region and if its value is less than 0 (i.e. indicating it is currently part of the labeled region); if so, it is overwritten with a $+\rho$ value, removing it from the labeled region. Our editing operators are parallelized on the ϕ-grid and operate in real-time.

Chapter 4

Results and Validation

We have performed a series of tests of the compute shader-based level set segmentation algorithm, using several synthetic and real data sets. Since this thesis is primarily concerned with the compute shader implementation and integration of volume image processing algorithms into the existing framework, we focus on demonstrating a working segmentation algorithm and some measurements of its performance, rather than on a formal analysis of segmentation accuracy and efficiency. A formal analysis of accuracy, as well as optimization of the GPU implementation and performance comparisons to alternative GPU level set implementations, is the subject of future work. For this reason, we rely upon informal visual analysis to validate the correctness of the segmentation output. For tests using synthetic data sets, we visually inspect the 3D rendering of the initial level set envelope and the final envelope to determine the degree of segmentation success. For tests using real data sets, we make use of the 3D slice plane capability and visually inspect slices containing the target anatomical structures as well as the segmentation "paint" to assess segmentation accuracy. We also present some results on the performance of our GPU-based level set implementation by comparing it to an equivalent CPU MATLAB implementation. This comparison also serves only to validate the correctness of GPU implementation; that is, we expect significant speedup over the MATLAB implementation. Finally we present tests of our 3D slice-plane based 3D painting facility to demonstrate its potential for initializing and editing the level set segmentation algorithm.

4.1 Segmenting Synthetic Data Sets

In the first series of tests, we demonstrate a working level set segmentation algorithm using a synthetic "cloverleaf" volume image. Voxel values inside the cloverleaf are smoothly graded inside and outside voxel values are set to 0. The inside values change smoothly from 200 to 205. We use several sizes of the data set, including 128 x 128 x 128 voxels and 256 x 256 x 256 voxels. Since the 3D image contains a single object, we painted an initial envelope surrounding the cloverleaf directly in 3D. Three paint brush strokes were required. We then triggered the level set segmentation algorithm with a key press. In Figures 4.1a,b we show two views of the cloverleaf and the initial painted envelope. Figures 4.1c,d

(a) (b) (c) (d)

Figure 4.1: Segmentation algorithm run on a a 256 x 256 x 256 cloverleaf dataset. (a),(b) Initial painted level set surface. (c),(d) Final segmentation result.

show two views after the segmentation has run for 100 iterations. The approximate time required for the segmentation was 39 seconds. The result is visually very accurate. For the smaller data set (i.e. 128 x 128 x 128 voxels) only 50 iterations were required to generate an accurate result, requiring approximately 2.4 seconds. The parameter settings[1] were μ=0.02, γ=5,λ=5, ϵ=1.5, τ=5

(a) (b) (c)

Figure 4.2: Segmentation algorithm run on a 168 x 160 x 92 CT image of a human vertebra phantom. (a) Initial painted level set surface. (b),(c) Final segmentation result.

In a second set of tests we use a 168 x 160 x 92 CT volume image of a human vertebra phantom (Figures 4.2. This test demonstrates the topological flexibility of level set surfaces. We painted an initial envelope without holes, directly in 3D, that surrounds the vertebra. The level set segmentation algorithm correctly captures the topology of the vertebra. The segmentation ran for 200 iterations. The approximate time required for the segmentation was 11 seconds. The result is visually very accurate. The parameter settings were μ=0.02, γ=5, λ=5, ϵ=1.5, τ=5. Note that the user is able to paint more accurate envelopes matching the topology of the target structure, resulting in fewer required iterations.

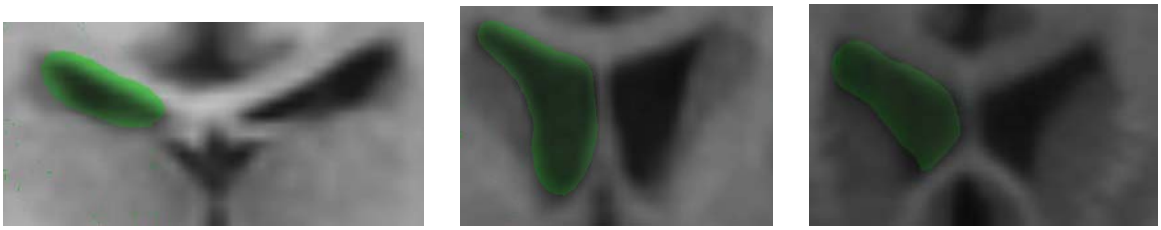
[1]See Li et al. (28) for a description of all parameters.

4.2 Segmenting Real Data Sets

(a) (b) (c)

Figure 4.3: Segmenting the lateral ventricle in a 240 x 240 x 192 MRI brain image. (a) Initial 3D slice painted level set surface. (b) Final segmentation result. (c) Volume rendering of manually segmented ventricle.

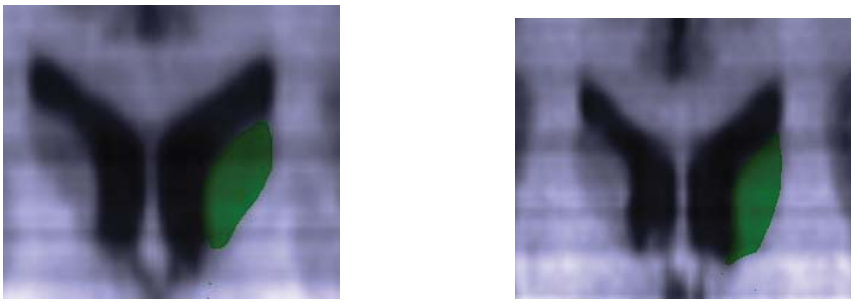
We ran our segmentation algorithm on a 240 x 240 x 192 MRI volume image of the brain. Segmentation of structures in MRI scans is often challenging due to noise, the similar voxel intensities of neighboring structures and the complexity of the target structure shape. In these two examples we segment the lateral ventricle and the caudate nucleus. In the first example we used the 3D slice painting facility and painted an envelope on several slices containing the lateral ventricle (Figure 4.3). The parameter settings were μ=0.02, γ=5,λ=5, ϵ=1.5, τ=5. The segmentation ran for 75 iterations and required approximately 14 seconds. In Figure 4.4, several 3D slice views are used to show the accuracy of the segmentation.



(a) (b) (c)

Figure 4.4: Slice-by-Slice examination of the segmentation of the lateral ventricle in a 240 x 240 x 192 MRI brain image. In (a),(b),(c) user scans through the slices to visually verify the segmentation result.

We also segmented the caudate nucleus from the brain image. This structure is challenging to segment due to its proximity to other structures with similar intensity characteristics. The initial and final level set surfaces are shown in Figure 4.6. In addition two slice views are shown in Figure 4.5. The 100 iterations of the evolution took approximately 25.7 seconds with parameters μ=0.02, γ=5, λ=5, ϵ=1.5, τ=5. Our implementation of the level set algorithm currently uses simple Gaussian smoothed gradient magnitude edges to stop the level set evolution. More accurate edges may lead to more accurate segmentations. We are currently investigating GPU-based median filtering (59; 7; 47; 39) combined with more sophisticated edge detectors.



(a) (b)

Figure 4.5: Slice-by-Slice examination of the caudate nucleus segmentation. In (a) and (b) different slices are shown and the user scans through the slices for a visual check of the segmentation.

(a) (b)

Figure 4.6: Segmentation of the caudate nucleus from a 3D MR brain image. (a) Initial 3D slice painted level set surface. (b) Final segmentation result.

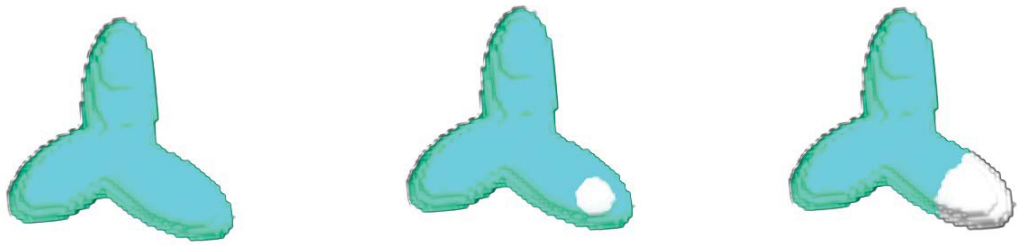
4.3 Editing

In this section we demonstrate the 3D slice plane editing facility. In Figure 4.7 we show a segmented cloverleaf object and manually erase the segmentation in the lower right region of the cloverleaf. Figure 4.8 shows an example of adding to a segmentation. In both case a thin cylindrical brush tip was used and only a single slice was affected. The user may optionally increase the size or thickness of the brush tip and paint on several slices at once. Figure 4.9 shows an example of editing the caudate nucleus segmentation on a 3D slice of the MR brain data set. We can see some segmentation "leakage" into the ventricle in the upper right portion of the caudate so it is erased. In Figure 4.3 a portion of the lateral ventricle is corrected using a small paint brush tip.

4.4 SpeedUp

We measured the wall clock time[2] of our GPU-based level set segmentation algorithm on a synthetic clover leaf data set. Three sizes of the data set were used: 64 x 64 x 64, 128 x 128 x 128 and 256 x 256 x 256. We then compared these times to an equivalent CPU-based MATLAB implementation. The GPU

[2]In practical computing, wall clock time or real-world time is the actual time, usually measured in seconds, that a program takes to run or to execute an assigned task



(a) (b) (c)

Figure 4.7: Editing by erasing a portion of the cloverleaf segmentation on a 3D slice plane using a thin cylindrical paint brush tip.

(a) (b) (c)

Figure 4.8: Editing on a 3D slice by adding to the segmentation of the vertebra.

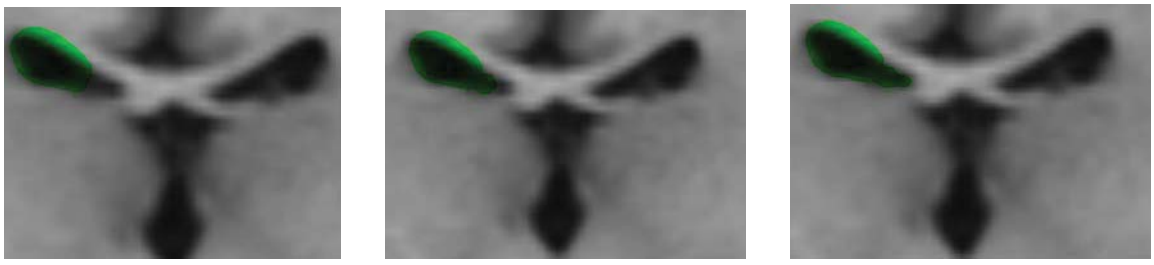
is a Nvidia GTX 570M with 1.5 GDDR5 random access memory (RAM), 7 streaming multiprocessors each with 48 cores. The CPU is an Intel i7-2670QM with 16 GB of RAM. The MATLAB code is not compiled with an optimized C++ compiler so we expected significant speedup from the GPU implementation. Table 4.1 summarizes the results. The considerable performance difference between the two implementations, coupled with the visual evaluation of segmentation accuracy, strongly suggests that the level set segmentation algorithm benefits greatly from a data parallel implementation.

Table 4.1: Segmentation Time Comparison of CPU versus GPU (100 iterations)

Dimension of grid	64 x 64 x 64	128 x 128 x 128	256 x 256 x 256
CPU	62.19 sec	665.7 sec	5386.29 sec
GPU	1.08 sec	5.1 sec	39.22 sec

(a) (b)

Figure 4.9: Erasing paint (i.e. the segmentation labels) on a 3D slice of a caudate nucleus segmentation.



(a) (b) (c)

Figure 4.10: A portion of the lateral ventricle segmentation is corrected using a small paint brush tip.

Chapter 5

Conclusions

Interactively exploring, visualizing and analyzing 3D medical images is a complex, integrated task. Quickly generating insightful views of these often massive and noisy data sets requires a 3D interaction model, real-time contextual visualization techniques and real-time image processing techniques that are seamlessly and tightly integrated into a single rendering pipeline. With the advent of powerful graphics hardware and GPU programming capabilities, real-time rendering of large 3D data sets is now possible. Furthermore, with the onset of GPGPU programming capabilities, real-time processing of these data sets is also within reach. However, until recently, the seamless integration of the two into a rendering pipeline was problematic. The release of compute shaders for GPGPU programming has potentially provided a solution to this integration problem. This thesis demonstrates the potential of compute shaders for creating a complete, integrated software framework for interactive 3D medical image visualization and processing that optimizes the use of the massively parallel computational and rendering power of modern graphics hardware. The compute shader based processing capabilities are designed such that the performance of the processing algorithms will naturally continue to improve as the number of GPU cores increases through graphics hardware evolution. The inherent design of compute shaders, as well as our use of a 3D grid based shader program interface supports flexible addition or replacement of processing algorithms. The extended framework uses a single intuitive painting interface for all selection, initialization and editing interactions with the data. The extended painting interface also supports processing of noisy volume images by integrating a 3D slice plane view directly with the volume rendered view.

Further improvements and additional capabilities can be made to the framework. Firstly, the GPU level set implementation is currently un-optimized and can be improved considerably using techniques similar to Roberts et al.(43). Secondly, while the painting interface provides slice-by-slice editing that supports post-processing of noisy volume image segmentation, it can be labour intensive if many slices require editing. One strategy to reduce or, in some cases, eliminate this editing phase would be the ability to interactively create “barriers” that reinforce target structure boundaries in boundary regions with no edge features. This capability could be carried out during the 3D slice painting of the initial target structure envelope. The user could flip to a “barrier-paint” mode and create thin 3D regions that

are then used to modify the edge detected image used by the level set segmentation algorithm. Another useful improvement would be the ability to segment as you paint. That is, thick "flattened" paint blobs and brush strokes can be painted on a 3D slice plane. This painted envelope is bounded by two parallel 3D slice planes, one on either side of the envelope. The level set segmentation algorithm can be executed and the level set surface is constrained by the two planes. This constrained segmentation would allow the user to quickly paint, segment and render a "chunk" of a target structure around the current slice plane and then continue to another slice plane to segment the next chunk. This segment-as-you-paint strategy may improve the volume exploration work-flow. Other possibilities for future work are the addition of alternative segmentation algorithms such as 3D active surface techniques (20; 49) and random walks (9; 16; 57).

Appendix 1

Level Set Formulation

This appendix provides some mathematical details of the level set segmentation algorithm, specifically that of the variational formulation used by Li et al. (28). The reader is referred to Li et al. (28) for details. Level sets are implicitly defined deformable surfaces defined as the zero level set $\{(x, y, z)|\phi(t, x, y, z) = 0\}$, of the level set function ϕ. Li et al. define an energy functional $\mathcal{E}(\phi)$, where the functional can be viewed as a representation of the energy of the deformable surface and the final shape of the surface corresponds to the minimum of this energy. The energy functional consists of two terms:

$$\mathcal{E}(\phi) = \mu\mathcal{P}(\phi) + \mathcal{E}_m(\phi). \tag{1.1}$$

where $\mu > 0$ is a parameter controlling the effect of the penalty term $\mathcal{P}(\phi)$ and $\mathcal{E}_m(\phi)$ is an external energy that drives the motion of the zero level surface. The term $\mathcal{P}(\phi)$ is used to penalize the deviation of ϕ from the signed distance function and is defined as:

$$\mathcal{P}(\phi) = \int_\Omega \frac{1}{2}(|\nabla\phi| - 1)^2 dxdydz \tag{1.2}$$

From the calculus of variations, the evolution equation

$$\frac{\partial\phi}{\partial t} = -\frac{\partial\mathcal{E}}{\partial\phi} \tag{1.3}$$

is the gradient flow that minimizes the functional $\mathcal{E}$. The external energy functional $\mathcal{E}_m(\phi)$ consists of two terms, both of which incorporate edge indicator function g defined for an image I:

$$g = \frac{1}{1 + |\nabla G_\sigma * I|^2}, \tag{1.4}$$

where G_σ is the Gaussian kernel with standard deviation σ. The edge indicator function stops the evolution of the level set surface on the boundaries of the target structure. Equation (1.3) can be approximated using finite differences, where spatial partial derivatives $\frac{\partial\phi}{\partial x}$, $\frac{\partial\phi}{\partial y}$, and $\frac{\partial\phi}{\partial z}$ are approximated

by central differences, and the temporal partial derivative $\frac{\partial \phi}{\partial t}$ is approximated by a forward difference. The result is an iterative difference equation:

$$\phi_{i,j,k}^{t+1} = \phi_{i,j,k}^{t} + \tau L(\phi_{i,j,k}^{t}) \tag{1.5}$$

where τ is a time step and $L(\phi_{i,j,k}^{t})$ is an approximation to the right hand side of equation 1.3.

Bibliography

[1] Meisam Aliroteh and Tim McInerney. Sketchsurfaces: Sketch-line initialized deformable surfaces for efficient and controllable interactive 3d medical image segmentation. In *Proceedings of the 3rd International Conference on Advances in Visual Computing - Volume Part I*, ISVC'07, pages 542–553, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[3] Stefan Bruckner and M. Eduard Gröller. Instant volume visualization using maximum intensity difference accumulation. *Comput. Graph. Forum*, 28(3):775–782, 2009.

[4] Joshua E. Cates, Aaron E. Lefohn, and Ross T. Whitaker. Gist: An interactive gpu-based level-set segmentation tool for 3d medical images. *Medical Image Analysis*, 8:217–231, 2004.

[5] H. L. J. Chen, F. F. Samavati, M. C. Sousa, and J. R. Mitchell. Sketch-based volumetric seeded region growing. In *Proceedings of the Third Eurographics Conference on Sketch-Based Interfaces and Modeling*, SBM'06, pages 123–130, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[6] Hung-Li Jason Chen, Faramarz F. Samavati, and Mario Costa Sousa. Gpu-based point radiation for interactive volume sculpting and segmentation. *The Visual Computer*, 24(7-9):689–698, Jul 2008.

[7] Wei Chen, M. Beister, Y. Kyriakou, and M. Kachelries. High performance median filtering using commodity graphics hardware. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4142–4147, Oct 2009.

[8] CIBC, 2014. ImageVis3D: An interactive visualization software system for large-scale volume data. Scientific Computing and Imaging Institute (SCI), Download from: http://www.imagevis3d.org.

[9] Maxwell D Collins, Jia Xu, Leo Grady, and Vikas Singh. Random walks based multi-image segmentation: Quasiconvexity results and gpu-based solutions. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1656–1663. IEEE, 2012.

[10] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 65–74, New York, NY, USA, 1988. ACM.

[11] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, GI '08, pages 73–80, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[12] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M. LaConte. Medical image processing on the {GPU} past, present and future. *Medical Image Analysis*, 17(8):1073 – 1094, 2013.

[13] L. Faynshteyn. Context-preserving volumetric data set exploration using a 3D painting metaphor. Master's thesis, Dept. of Computer Science, Ryerson University, Toronto, ON, Canada, 2012.

[14] L. Faynshteyn and T. McInerney. Context-preserving volumetric data set exploration using a 3d painting metaphor. In *Advances in Visual Computing - 8th International Symposium, ISVC, 2012, Rethymnon, Crete, Greece, July 16-18, 2012, Revised Selected Papers, Part I*, pages 336–347, 2012.

[15] Thomas Fogal and Jens Kruger. Tuvok, an Architecture for Large Scale Volume Rendering. In *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization*, November 2010.

[16] Leo Grady, Thomas Schiwietz, Shmuel Aharon, and Rdiger Westermann. Random walks for interactive organ segmentation in two and three dimensions: Implementation and validation. In JamesS. Duncan and Guido Gerig, editors, *Medical Image Computing and Computer-Assisted Intervention MICCAI 2005*, volume 3750 of *Lecture Notes in Computer Science*, pages 773–780. Springer Berlin Heidelberg, 2005.

[17] Markus Hadwiger, Patric Ljung, Christof Rezk Salama, and Timo Ropinski. Advanced illumination techniques for gpu-based volume raycasting. In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, pages 2:1–2:166, New York, NY, USA, 2009. ACM.

[18] Markus Hadwiger, Christian Sigg, Henning Scharsach, Khatja Bhler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[19] Ghassan Hamarneh, Johnson Yang, Chris Mcintosh, and Morgan Langille. 3D live-wire-based semi-automatic segmentation of medical images. In *Proceedings of SPIE Medical Imaging: Image Processing 5747*, pages 1597–1603, 2005.

[20] Zhiyu He and Falko Kuester. Gpu-based active contour segmentation using gradient vector flow. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Ara V. Nefian, Meenakshisundaram Gopi, Valerio Pascucci, Jiri Zara, Jose Molineros, Holger Theisel, and Thomas Malzbender, editors, *ISVC (1)*, volume 4291 of *Lecture Notes in Computer Science*, pages 191–201. Springer, 2006.

[21] Frank Heckel, Jan H. Moltz, Christian Tietjen, and Horst K. Hahn. Sketch-based editing tools for tumour segmentation in 3d medical images. *Computer Graphics Forum*, 32(8):144–157, 2013.

[22] Takashi Ijiri and Hideo Yokota. Contour-based interface for refining volume segmentation. *Comput. Graph. Forum*, 29(7):2153–2160, 2010.

[23] Yan Kang, Klaus Engelke, and Willi A. Kalender. Interactive 3d editing tools for image segmentation. *Medical Image Analysis*, pages 35–46, 2004.

[24] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Hsien-Chi Kuo, Maryellen L. Giger, Ingrid S. Reiser, John M. Boone, Karen K. Lindfors, Kai Yang, and Alexandra Edwards. Level set segmentation of breast masses in contrast-enhanced dedicated breast ct and evaluation of stopping criteria. *J. Digital Imaging*, 27(2):237–247, 2014.

[26] Aaron E. Lefohn, Joshua E. Cates, and Ross T. Whitaker. Interactive, gpu-based level sets for 3d segmentation. In Randy E. Ellis and Terry M. Peters, editors, *MICCAI (1)*, volume 2878 of *Lecture Notes in Computer Science*, pages 564–572. Springer, 2003.

[27] Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, May 1988.

[28] Chunming Li, Chenyang Xu, Changfeng Gui, and Martin D. Fox. Level set evolution without reinitialization: A new variational formulation. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), 20-26 June 2005, San Diego, CA, USA*, pages 430–436, 2005.

[29] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987*, pages 163–169, 1987.

[30] Yuancheng Luo and R. Duraiswami. Canny edge detection on nvidia cuda. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, June 2008.

[31] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[32] T. McInerney and L. Faynshteyn. Interactive volume of interest selection using a superellipsoid paintbrush. Manuscript in preparation, 2015.

[33] Ken Museth, David E. Breen, Ross T. Whitaker, and Alan H. Barr. Level set surface editing operators. *ACM Trans. Graph.*, 21(3):330–338, 2002.

[34] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[35] Stanley Osher and James A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on hamilton-jacobi formulations. *JOURNAL OF COMPUTATIONAL PHYSICS*, 79(1):12–49, 1988.

[36] Shigeru Owada, Frank Nielsen, and Takeo Igarashi. Volume catcher. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 111–116, New York, NY, USA, 2005. ACM.

[37] Lei Pan, Lixu Gu, and Jianrong Xu. Implementation of medical image segmentation in cuda. In *Information Technology and Applications in Biomedicine, 2008. ITAB 2008. International Conference on*, pages 82–85, May 2008.

[38] Eric Penner. Three-dimensional medical image visualization techniques on modern graphics processors. Master's thesis, University of Calgary, Calgary, Alberta, Canada, 2009.

[39] Gilles Perrot, Stéphane Domas, and Raphaël Couturier. Fine-tuned high-speed implementation of a gpu-based median filter. *J. Signal Process. Syst.*, 75(3):185–190, June 2014.

[40] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[41] Jrg-Stefan Prani, Timo Ropinski, and Klaus Hinrichs. Uncertainty-aware guided volume segmentation. pages 1358–1365, 2010.

[42] N. Pühringer. Sketch-based modelling for volume visualization. Master's thesis, Vienna University of Technology, 2009.

[43] Mike Roberts, Jeff Packer, Mario Costa Sousa, and Joseph Ross Mitchell. A work-efficient gpu algorithm for level set segmentation. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 123–132, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[44] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the Symposium on Data Visualisation 2003*, VISSYM '03, pages 231–238, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[45] Daniel Ruijters, Bart M. ter Haar Romeny, and Paul Suetens. Gpu-accelerated elastic 3d image registration for intra-surgical applications. *Computer Methods and Programs in Biomedicine*, 103(2):104 – 112, 2011.

[46] Martin Rumpf and Robert Strzodka. Level set segmentation in graphics hardware. In *Proceedings of IEEE International Conference on Image Processing (ICIP'01)*, volume 3, pages 1103–1106, 2001.

[47] Ricardo M. Sánchez and Paul A. Rodríguez. Highly parallelable bidimensional median filter for modern parallel programming models. *J. Signal Process. Syst.*, 71(3):221–235, June 2013.

[48] Henning Scharsach. Advanced gpu raycasting. In *In Proceedings of CESCG 2005*, pages 69–76, 2005.

[49] Jrme Schmid, Jos Antonio Iglesias Guitin, Enrico Gobbetti, and Nadia Magnenat-Thalmann. A gpu framework for parallel segmentation of volumetric images using discrete deformable models. *Vis. Comput.*, 27(2):85–95, February 2011.

[50] Andreas Schwarzkopf, Thomas Kalbe, Chandrajit Bajaj, Arjan Kuijper, and Michael Goesele. Volumetric nonlinear anisotropic diffusion on gpus. In *Scale Space and Variational Methods in Computer Vision - Third International Conference, SSVM 2011, Ein-Gedi, Israel, May 29 - June 2, 2011, Revised Selected Papers*, pages 62–73, 2011.

[51] Ramtin Shams and R.A. Kennedy. Efficient histogram algorithms for nvidia cuda compatible devices. In *Proceedings of International Conference on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, 2007.

[52] Ramtin Shams, Parastoo Sadeghi, Rodney Kennedy, and Richard Hartley. Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images. *Comput. Methods Prog. Biomed.*, 99(2):133–146, August 2010.

[53] O. Sharma, Qin Zhang, F. Anton, and C. Bajaj. Multi-domain, higher order level set scheme for 3d image segmentation on the gpu. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2211–2216, June 2010.

[54] Anthony Sherbondy, Mike Houston, and Sandy Napel. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *in IEEE Visualization*, pages 171–176, 2003.

[55] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[56] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[57] Andrew Top, Ghassan Hamarneh, and Rafeef Abugharbieh. Active learning for interactive 3d image segmentation. In *Proceedings of the 14th International Conference on Medical Image Computing and Computer-assisted Intervention - Volume Part III*, MICCAI'11, pages 603–610, Berlin, Heidelberg, 2011. Springer-Verlag.

[58] V. Vineet and P.J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, June 2008.

[59] Ivan Viola, Armin Kanitsar, and Eduard Grller. Hardware-based nonlinear filtering and segmentation using high-level shading languages. In *in Proceedings of IEEE Visualization*, pages 309–316, 2003.

[60] XinXin Wang and B.E. Shi. Gpu implemention of fast gabor filters. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 373–376, May 2010.

[61] Xiaoru Yuan, Nan Zhang, Minh X. Nguyen, and Baoquan Chen. Volume cutout. *The Visual Computer*, 21(8-10):745–754, 2005.