INTERACTIVE COMPUTATION AND VISUALIZATION OF LEVEL-SET SURFACES: A STREAMING NARROW-BAND ALGORITHM

by

Aaron Eliot Lefohn

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

School of Computing

The University of Utah

May 2004

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Aaron Eliot Lefohn

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Ross Whitaker
	Charles Hansen
	Steven Parker

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Aaron Eliot Lefohn in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ross Whitaker
Chair, Supervisory Committee

Approved for the Major Department

Chris Johnson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Deformable isosurfaces, implemented with level-set methods, have demonstrated a great potential in visualization and computer graphics for applications such as segmentation, surface processing, and surface reconstruction. Their usefulness has been limited, however, by two problems. First, three-dimensional level sets are relatively slow to compute. Second, their formulation usually entails free parameters that can be difficult to tune correctly for specific applications. The second problem is compounded by the first. This thesis presents a solution to these challenges by describing graphics processor unit (GPU) based algorithms for solving and visualizing level-set solutions at interactive rates for volumes as large as 256^3.

Level-set techniques deform isosurfaces by solving partial differential equations (PDEs) on a voxel grid. Efficient solvers for the equations compute a solution only at those voxels on or near the isosurface. The active elements in this *narrow-band* of computation change as the level-set solution evolves. This thesis demonstrates that such *dynamic sparse-grid* computations can be efficiently solved using a streaming architecture platform–a modern graphics processor. The solution uses a multidimensional virtual memory mapping to pack the active, three-dimensional voxel data into two-dimensional texture memory on the GPU. A novel GPU-to-CPU message passing scheme quickly updates this sparse data structure as the isosurface moves.

The integration of the level-set solver with a real-time volume renderer allows a user to visualize and steer the deformable level-set surface as it evolves. The resulting isosurface can also serve as a region-of-interest specifier for the volume renderer. This thesis demonstrates the capabilities of this technology for interactive volume segmentation and visualization. This thesis also presents an evaluation of the method with a brain tumor segmentation user study.

to Karen

CONTENTS

ABSTRACT . . . . . **iv**

LIST OF FIGURES . . . . . **viii**

ACKNOWLEDGMENTS . . . . . **xi**

CHAPTERS

1. INTRODUCTION **1**

1.1 Problem Statement 1
1.2 Contributions and Results 3
1.3 Overview 3

2. TECHNICAL BACKGROUND AND RELATED WORK **5**

2.1 The Level-Set Method 5
2.2 Narrow-Band Level-Set Solvers 7
2.3 Scientific Computation on Graphics Processors 8

3. STREAMING NARROW-BAND ALGORITHM **12**

3.1 Introduction 12
3.2 A Virtual Memory Address Scheme for Sparse Computation 13
3.2.1 Traditional Virtual Memory Overview 13
3.2.2 Multidimensional Virtual Memory for GPUs 14
3.2.3 Virtual-to-Physical Address Translation 17
3.2.4 Bootstrapping the Virtual Memory System 20
3.3 Streaming Narrow-Band GPU Level-Set Solver 21
3.3.1 Initialization of Computational Domain 21
3.3.2 The Distance Transform Computation on the GPU 23
3.3.3 Level-Set Computation 24
3.3.4 GPU Implementation Details 24
3.3.5 Update of Computational Domain 25
3.4 Volume Rendering of Packed Data 26

4. SEGMENTATION APPLICATION **30**

4.1 Introduction 30
4.2 Volume Segmentation and Visualization Application 30
4.2.1 Level-Set Formulation for Segmentation 30
4.2.2 Interface and Usage 32
4.3 Performance Analysis 39

4.4 Tumor Segmentation User Study 40
4.4.1 Introduction 40
4.4.2 Methodology 41
4.4.3 Results 42

5. CONCLUSIONS 48

5.1 Summary 48
5.2 Future Work 48

APPENDICES

A. DISCRETIZATION OF THE LEVEL-SET EQUATIONS 51

B. A BRUTE-FORCE, GPU-BASED THREE-DIMENSIONAL LEVEL-SET SOLVER 54

C. GPU MEMORY ALLOCATION REQUEST GENERATION 56

D. SOFTWARE DESIGN 60

REFERENCES 72

LIST OF FIGURES

2.1 The three fundamental steps in a sparse-grid solver. Step 1 initializes the sparse computational domain. Step 2 executes the computational kernel on each element in the domain. Step 3 updates the domain if necessary. Steps 2 and 3 are repeated for each solver iteration. 8

2.2 The modern graphics processor pipeline. 10

3.1 The multidimensional virtual and physical memory spaces used in the paged virtual address system. The original problem space is V, the virtual address space. The virtual page space, V_P, is a subdivided version of V. Virtual memory pages are mapped to the physical page space, G_P, by the *page table*. The *inverse page table* maps physical pages in G_P to virtual pages in V_P. The collection of all elements in G_P constitute G, the physical memory of the hardware. 15

3.2 The virtual-to-physical address translation scheme in the multidimensional virtual memory system. A three-dimensional virtual address, VA, is first translated to a virtual page number, VPN. A page table translates the VPN to a physical page address, PPA. The PPA specifies the origin of the physical page containing the physical address, PA. The offset is then computed based from the virtual address and used to obtain the final two-dimensional physical address, PA. 16

3.3 The *substream* boundary cases used to statically resolve the conditionals arising from $3 \times 3 \times 3$ neighbor accesses across memory page boundaries. The nine *substream* cases are: interior, four edges, and four corners (a). The interior case accesses its neighbors from only three memory pages (b). The edge cases require six pages (c), and the corner cases require 12 memory pages (d). Note that for reasonably large page sizes, the more cache-friendly interior case has by far the highest number of data elements. 18

3.4 The level-set solver's use of the paged virtual memory system. All *active* pages (i.e., those that contain nonzero derivatives) in the virtual page space (a) are mapped to unique pages of physical memory (b). The inactive virtual pages are mapped to the static *inside* or *outside* physical page. Note that the only data stored on the GPU is that represented by (b). 21

3.5 The level-set embedding, ϕ, is a clamped distance transform, i.e., $|\nabla\phi|$ is nonzero near the surface model and zero elsewhere. 22

3.6 The GPU's creation of a memory allocation/deallocation request. Step A uses solver-specific data to create two buffers containing the active state of each data element and its adjacent neighbors. Step B uses automatic mipmapping to reduce the buffers from size $S[\mathrm{G}]$ to the physical page space size, $S[\mathrm{G_P}]$. Step C combines the information from the two down-sampled state buffers into an eight-bit code for each pixel. This code encapsulates whether or not each active virtual memory page and its adjacent neighbors should be enabled. In step D, the CPU reads the bit-code buffer, decodes it, and allocates/deallocates pages as requested. 26

3.7 Two pass rendering of packed volume data. In step A, a two-dimensional slice (i) is reconstructed from the physical page (packed) layout, $\mathrm{G_P}$. In step B, one or more intermediate slices between i and $i-1$ are interpolated, transformed into optical properties (via the transfer function), lit, and rendered for the current view. The next iteration begins by reconstructing slice $i+1$, replacing $i-1$, and so on. 28

3.8 Reconstruction of a slice for volume rendering the packed level-set model: (a) When the preferred slicing direction is orthogonal to the virtual memory page layout, the pages (shown in alternating colors) are draw into a pixel buffer as quadrilaterals. (b) For slicing directions parallel to the virtual page layout, the pages are drawn onto a pixel buffer as either vertical or horizontal lines. 28

4.1 The use of a curvature constraint (speed function) in the level-set computation to prevent segmentation "leaking." This example shows one slice of a three-dimensional MRI segmentation computation: (a) The spherical initialization. (b) A model expands to fill the tumor but leaks through gaps and expands into other anatomy. (c) The same scenario with a degree of curvature prevents unwanted leaking. The level set isosurface is shown in white. 31

4.2 A speed function based on image intensity causes the model to expand over regions with greyscale values within the specified range and contract otherwise. 32

4.3 A depiction of the user interface for the volume analysis application. Users interact via slice views, a three-dimensional rendering, and a control panel. 33

4.4 The actual user interface for the volume analysis application. The top left window shows the visualization of the speed function. The top right window shows a slice of the MRI source data with the current level-set solution in yellow. The lower-left window shows a volume rendering of the MRI source data (blue), the same data projected onto a clipping plane (grey), the current level-set surface (brown), and the intersection of the current level-set solution with the clipping plane (yellow). 33

4.5 Interactive level-set segmentation of a brain tumor from a $256 \times 256 \times 198$ MRI with volume rendering to give context to the segmented surface. A clipping plane (bottom) shows the user the source data, the volume rendering, and the segmentation simultaneously. The segmentation and volume rendering parameters are set by the user probing data values on the clipping plane. 35

4.6 Interactive level-set segmentation of the cerebral cortex from a $256 \times 256 \times 198$ MRI with volume rendering to give context to the segmented surface. The MRI data is also projected onto a clipping plane, on which the user can probe to control the level-set parameters. 36

4.7 The top image shows a volume rendering of a 256^3 MRI scan of a mouse thorax. Note the level set surface which is deformed to segment the liver. The bottom image shows a volume rendering of the vasculature inside the liver. Both images are rendered using the same transfer function with the level-set surface serving as a region-of-interest specifier. 37

4.8 Segmentation and volume rendering of $512 \times 512 \times 61$ three-dimensional transmission electron tomography data. The picture shows cytoskeletal membrane extensions and connexins (pink surfaces extracted with the level-set models) near the gap junction between two cells (volume rendered in cyan). 38

4.9 Sensitivity (the fraction of pixels correctly classified as inside the object boundary) results from the user study compare the interactive, GPU-based level-set segmentation tool with expert hand contouring. The results show that users of the semi-automatic tool produced segmentations that were within the error bounds of the expert hand contours in most cases. The tool also showed an overall slightly lower sensitivity, meaning that the size of the segmentations is slightly smaller. 43

4.10 Specificity (the fraction of pixels correctly classified as outside the object boundary) results from the user study compare the interactive, GPU-based level-set segmentation tool with expert hand contouring. The results show that users of the semiautomatic tool produced segmentations that were within the error bounds of the expert hand contours in most cases. The tool also showed an overall slightly higher specificity, meaning that the size of the segmentations is slightly smaller. 44

4.11 The total fraction of correctly classified pixels (combination of sensitivity and specificity) for the nine tumor cases segmented by the participating users. 45

4.12 An expert hand segmentation of a tumor from the Harvard Brigham and Women's database shows significant interslice artifacts. 47

4.13 A three-dimensional segmentation of the same tumor from one of the subjects in the user study performed using the interactive segmentation tool described in this thesis. 47

D.1 The five software layers with which the level-set segmentation application is built. In the first layer, OpenGL is used to control the GPU, Gutz defines vector, matrix, and array data structures, and Glew handles OpenGL extensions. The second layer, Glift, combines OpenGL calls into reusable object-oriented OpenGL modules. CompGPU is the third layer and encapsulates an entire render pass as a `forEach` function call. The level-set solver, level-set speed functions and visualization modules are defined in the fourth layer, and the volume segmentation application comprises the fifth layer. . 61

D.2 Class tree for the Glift, object-oriented OpenGL framework. 65

ACKNOWLEDGMENTS

I feel incredibly lucky to have spent the last two years as a graduate student in the University of Utah computer science department and specifically in the Scientific Computing and Imaging Institute (SCI). The most outstanding feature of the graphics and SCI groups at Utah is the overwhelmingly open collaborative environment. I have witnessed many impromptu discussions between students and/or faculty turn into fruitful research ideas and papers. The supportive and talented faculty and students, combined with an open, collaborative, hard-working atmosphere, creates an environment that is ripe for idea generation. Being a part of this madness for the last two years has truly changed my career and life.

I want to first specifically thank my advisor, Ross Whitaker. I greatly appreciate Ross's attraction to problems that others discount as impossible. His ability to let me work independently for long periods of time, yet answer detailed questions with little or no notice was the perfect advisorship. Ross's ability to point out subtle details, problems, and solutions was an invaluable resource. Beyond just research advising, he spent a significant amount of time improving my writing and presentation style as well as offering career advice. Ross has shown me the ideal role that an academic advisor can play in a graduate student's life. It is also worth noting that his rock climbing skills have also progressed substantially in the last two years.

I also want to thank my committee, Charles Hansen and Steven Parker, for their support and criticism throughout the project. I especially want to thank Charles Hansen, whose graphics classes inspired me to change careers, and for his help in making that career change. Chris Johnson deserves a special thanks for helping make SCI the intense, fast-moving, and inspiring place that it is. I also want to thank Chris for much career advice and support. I also recognize all of the talented members of SCI's administration, support, and media team for helping create a fantastic research environment.

My time with Peter Shirley has also had a profound impact on me. Pete's constant barrage of open-research declarations during his Image Synthesis course was a perpetual source of motivation and stimulation. Although not part of this thesis, my IEEE

Computer Graphics and Applications paper on human iris rendering was inspired by discussions with Pete. His poignant and creative criticism of my writing ability is also greatly appreciated.

I can not emphasize strongly enough the contribution of my colleagues to my research. Joe Kniss has been an integral part of my work since the beginning. Joe began as my mentor—introducing me to the world of GPU hacking. Joe has made significant contributions to all parts of this project. He is responsible for convincing me to pursue the substream idea for fragment conditional resolution. The edge-on volume rendering reconstruction idea was developed by Joe specifically for this project. He also wrote nearly all of the volume rendering code discussed in this thesis and created many of the figures found herein. I cannot thank Joe enough for his friendship, ideas, and hard work throughout my time in Utah.

Joshua Cates deserves many thanks for helping design, administer, and analyze the tumor segmentation user study. His expertise in segmentation evaluation methodology and analysis was an invaluable contribution to the application chapter of this thesis. Josh also provided the CPU-based level-set solver code to which this GPU-based solver is compared.

Milan Ikits has also contributed significantly to this work. Milan provided much input into the design of the Glift OpenGL framework. He also helped me debug much of the GPU code and was always willing to listen to a "confessional debugging" session and provide valuable feedback. Milan also contributed to many of the design discussions for the sparse solver and renderer. His Glew, OpenGL extension handler is also an integral part of the code discussed herein.

Gordon Kindlmann has a dominating and fatherly presence in the SCI lab. Gordon's rigor and mathematical savvy are regularly used to identify holes in an otherwise plausible-sounding idea. I thank him for his repeated criticism, skepticism, and brilliance. His Teem toolkit for manipulating N-dimensional raster data is also an integral part of this software. His tools have saved me weeks of coding and misery.

Milan, Gordon, and Joe were all integral in the discussions that led to the multidimensional virtual memory abstraction. I again thank the three of them for their time and insight into the problem.

Despite the fact that I rarely left the lab, I greatly appreciate Miriah Meyer's tireless attempts to get me out of the building and into the mountains. I also want to thank

Chris Wyman for sitting through repeated fragment program debugging sessions with me, and (in no particular order), other members of the graphics and SCI groups: Justin Polchlopek, Simon Premoze, Kristin Potter, David DeMarle, Xinwei Xue, Won-Ki Jeong, Shaun Ramsey, Amy Williams, Tim Miller, David Weinstein, Jason Waltman, Helen Hu, Betty Mohler, Margarita Bratkova, Brian Budge, Erik Reinhard, Suyash Awate, Mike Stark, Bill Martin, Amy Gooch, and Bruce Gooch for their friendship and help along the way.

John Owens was a great help in clarifying the substream discussion as well as helping me see the larger context of my work.

Several employees of ATI Technologies Inc. have contributed significant time to this research. Evan Hart's deep knowledge and willingness to help was critical to the success of this work. He and Jason Mitchell have provided advice and support to this project from its inception. Jeff Royal, Arcot Preetham, and Mark Segal also deserve thanks for answering questions and donating hardware to the project.

I also want to thank the providers of the volumetric data sets. Steve Lamont and Gina Sosinsky at the National Center for Microscopy and Imaging Research at UCSD provided the tomography data. Simon Warfield, Michael Kaus, Ron Kikinis, Peter Black and Ferenc Jolesz provided the MRI head data. The mouse data was supplied by the Center for In Vivo Microscopy at Duke University.

The pursuit of this research would not have been possible without the support of the following grants. The work was funded by grants ACI0089915 and CCR0092065 from the National Science Foundation, as well as Office of Naval Research grant N000140110033.

I lastly want to thank my family. My parents, Allen and Phyllis continue to be my role models for pursuing life with creativity, passion, and hard work. To the rest of my family; Li, Martin, Kevin, Milyn, Denis, Chris, Eileen, Joanne, Philip, Peter, Nancy, Sarah, and Robbie, I am deeply grateful for your inspiration and support. I especially want to thank you for being understanding of my absence throughout the duration of this work. By far the most important acknowledgement, however, goes to my wife, Karen. I owe her my deepest and most heartfelt thank you for patiently believing that the endless sleepless nights and seven day work weeks were a worthy sacrifice. I am also deeply appreciative of the times that she forcibly removed me from the lab and reintroduced me to the magic of the Southern Utah desert.

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

Surfaces define the objects in the world around us—the rough surface of a granite rock cliff, the smooth plastic of a child's toy, the complex surface of a kitchen sponge, and the dynamic water surface of a stormy sea. Scientific exploration often includes the search for surfaces that denote important boundaries. Examples include a geologist searching for oil deposits in the earth's strata, a neurosurgeon finding the exact extent of a brain tumor, and a snow avalanche forecaster discovering a dangerously weak layer in a mountain snow pack.

Just as in the physical world, surfaces are a critical component of computational science and computer graphics. There are many techniques for representing computational surface models including vertex meshes, b-spline patches, implicit surfaces, and others. The deformation of these surface representations, however, presents a number of challenges. Example applications of deformable surfaces include surface tracking in fluid simulations, image and volume segmentation, and surface processing (e.g., smoothing, sharpening, blending).

The deformation of an explicit surface representation (e.g., vertex mesh) involves updating connectivity and parameterization information. This can be difficult as well as limit the range of possible deformations to those which do not change topological genus. Implicit surfaces, on the other hand, can easily change topological genus, split into multiple entities, and merge multiple surfaces together. The deformation technique discussed in this thesis, the level-set method, is a promising technique for modeling deformable implicit surfaces.

Level-set methods model deformable isosurfaces with a set of partial differential equations (PDEs) that act on an implicit surface voxel grid. The specific PDEs, and thus the surface deformations, are determined by the level-set application. For physically-based simulation applications, the simulation results determine the surface movement. For many

nonsimulation applications (e.g., surface processing and segmentation), level-set surface deformation is controlled by a set of free parameters.

While the level-set approach is flexible and powerful, its use can be problematic. First, level sets are relatively slow to compute. Second, the free parameters used by some applications to control surface deformation are often difficult to set. The latter problem is compounded by the first because, in many scenarios, a user must wait minutes or hours to observe the results of a parameter change.

In response to the need to accelerate level-set computations, researchers have created a number of optimization strategies. The most successful of these are the *sparse-grid* and *narrow-band* strategies which solve the level-set PDE only on the voxels near the isosurface (rather than on the entire voxel grid). Although these optimized solvers achieve significant speedups, they are still far from interactive for all but the smallest three-dimensional computations. The work in this thesis builds on these optimizations by presenting a narrow-band algorithm that runs on a modern graphics processor.

The streaming architecture of modern graphics processors provides an attractive alternate computing platform for computationally demanding problems. These specialized processors accelerate three-dimensional computer graphics computations with a combination of dedicated hardware, data-parallel computation, and high-speed memory. Although level-set algorithms exhibit the required data-parallelism to run on GPUs, the sparse and dynamic nature of the computation makes mapping them to graphics hardware difficult.

This thesis presents a solution to the above problems by presenting an efficient mapping of the level-set partial differential equations to a commodity graphics processor. This GPU-based solver runs up to 15 times faster than a highly-optimized sparse-field implementation running on a modern central processing unit (CPU). By combining the fast solver with a real-time volume renderer, a user is able to both visualize and easily control the evolving computation. This thesis presents an interactive volume segmentation application built with this new solver. The thesis also presents an evaluation user study in which brain tumors are segmented from MRI data using this new segmentation tool.

1.2 Contributions and Results

This thesis makes contributions in the fields of deformable surface processing, GPU-based streaming computation, volume visualization, and medical segmentation. The main contributions are:

- An integrated system demonstrating that level-set computations can be intuitively controlled by coupling a real-time volume renderer with an interactive solver
- An interactive volume segmentation application built with the new solver, and a user study that quantifies the effectiveness of the new tool for quickly and accurately segmenting tumors from MRI data sets
- A GPU-based three-dimensional level-set solver that is approximately 15 times faster than previous optimized solutions
- A multidimensional virtual memory scheme for GPU texture memory that supports computation on time-dependent, sparse data domains
- A message passing scheme between the GPU and CPU that uses automatic mipmap generation to create compact encoded bitcode messages
- Real-time volume rendering directly from a two-dimensional packed, sparse texture format
- Region of interest specification for the volume renderer
- Efficient computation of a volumetric distance transform on the GPU

1.3 Overview

The following chapter discusses previous work and background for level sets, GPUs and hardware-accelerated volume rendering. Chapter 3 describes the details of the streaming narrow-band solver. The first section (Section 3.2) introduces a multidimensional virtual memory system used to pack the active three-dimensional data into two-dimensional texture memory. Section 3.3 then describes the details of the streaming level-set solver in terms of the virtual memory system. That section also explains a new distance transform computation that runs efficiently on the GPU. Section 3.4 explains how the packed, two-dimensional data format is volume rendered at interactive rates. Chapter 4 describes the interactive, three-dimensional segmentation application built using the streaming

level-set solver. Section 4.3 discusses the performance of the application, and Section 4.4 presents a brain tumor segmentation an evaluation user study performed with the new tool. The conclusions in Chapter 5 summarize the work and propose future research directions in both streaming level-set solvers and graphics hardware.

CHAPTER 2

TECHNICAL BACKGROUND AND RELATED WORK

2.1 The Level-Set Method

The level-set surface deformation technique is based on an implicit surface representation. In the level-set approach, an n-dimensional manifold is embedded in a $\mathbb{R}^{n+1}$ space (i.e., a manifold with codimension one). A scalar function, $\phi(\mathbf{x}, t)$, defines the surface embedding, where $\mathbf{x} \in \mathbb{R}^{n+1}$ and t is time. The set of points on the surface at time t, $\mathbf{S}_t$, are mapped by $\phi(\mathbf{x}, t)$ such that

$$\mathbf{S}_t = \{\mathbf{x} | \phi(\mathbf{x}, t) = k\}, \tag{2.1}$$

where k is an arbitrary scalar value (often zero). It can also be said that $\mathbf{S}_t$ is the k level set of $\phi(\mathbf{x}, t)$. The discrete representation of $\phi(\mathbf{x}, t)$ is referred to as the *embedding* of the level set k. For instance, the embedding for the kth level set can be created by setting each point on a uniform grid in $\mathbf{S}_0$ to k, all points inside the surface to $\phi(\mathbf{x}, 0) > k$ and all grid points outside to $\phi(\mathbf{x}, 0) < k$. The signed distance from the k isosurface is often used for the embedding, $\phi(\mathbf{x}, t)$, but it is not a requirement of the technique.

The embedding, $\phi(\mathbf{x}, t)$, evolves with the surface, and the relationship is given by the first-order, partial differential equation

$$\frac{\partial \phi(\mathbf{x}, t)}{\partial t} = -\nabla \phi(\mathbf{x}, t) \cdot \mathbf{v}(\mathbf{x}, t), \tag{2.2}$$

where $\mathbf{v}(\mathbf{x}, t)$ describes the velocity of the surface at point $\mathbf{x}$ at time t. Within this framework one can implement a wide range of deformations by defining an appropriate $\mathbf{v}(\mathbf{x}, t)$. This velocity term is often a combination of several other terms, including data-dependent terms, geometric terms (e.g., curvature), and others. In many applications, these velocities introduce free parameters, and the proper tuning of those parameters is critical to making the level-set model behave in a desirable manner. Equation 2.2 is the

general form of the level-set equation, which can be tuned for wide variety of problems and which motivates the architecture of the new solver.

Although the proposed solver addresses the solution to Equation 2.2, this thesis restricts the discussion to a special form of Equation 2.2 that is suitable for the segmentation application described in Chapter 4. This special case of Equation 2.2 occurs when $\mathbf{v}(\mathbf{x},t) = \mathrm{G}(\mathbf{x},\mathrm{t})\mathbf{n}(\mathbf{x},\mathrm{t})$, where $\mathbf{n}(\mathbf{x},t)$ is the surface normal and $\mathrm{G}(\mathbf{x},\mathrm{t})$ is a scalar field, which is refereed to as the *speed* of the level set. In this case Equation 2.2 becomes

$$\frac{\partial\phi(\mathbf{x},t)}{\partial t} = -|\nabla\phi(\mathbf{x},t)|\mathrm{G}(\mathbf{x},\mathrm{t}). \tag{2.3}$$

Equation 2.3 describes a surface motion in the direction of the surface normal, and thus the volume enclosed by the surface expands or contracts, depending on the sign and magnitude of $\mathrm{G}(\mathbf{x},\mathrm{t})$. The remainder of this thesis uses an abbreviated notation by assuming the spatial and temporal variability of $\phi(\mathbf{x},t)$, $\mathrm{G}(\mathbf{x},\mathrm{t})$, and $n(\mathbf{x},t)$ are understood. These quantities are thus referred to as ϕ, G, and n respectively.

The mean curvature of the level sets of ϕ, H, (hereafter referred simply as curvature) is commonly used as a level-set speed function (i.e., G). Because applying curvature flow to a surface minimizes surface area, curvature is often combined with data-based speed terms to smooth out an otherwise rough or noisy surface solution. A convex surface under pure curvature flow will converge to the n-sphere and finally a single point [14]. The mean curvature of ϕ is defined as

$$H = c_n\nabla\cdot\frac{\nabla\phi}{|\nabla\phi|}, \tag{2.4}$$

where, if n is the dimensionality of the surface, $c_n = 1/(n-1)$.

There is a special case of Equation 2.2 in which the surface motion is strictly inward or outward. In such cases the PDE can be solved somewhat efficiently using the *fast marching method* [44] and variations thereof [11]. However, this case covers only a very small subset of interesting speed functions. In general, this work in this thesis is concerned with solutions that allow the model to expand and contract as well as include a curvature term.

The initial estimation of ϕ is propagated forward in time using finite forward differences. The gradient magnitudes are computed with the up-wind scheme [34]. To guarantee a stable solution, the upwind scheme approximates $\nabla\phi$ using one-sided derivatives that are always in the *up-wind* direction of the propagating surface. The largest

allowable time step, $\triangle t$, is inversely proportional to the maximum speed at a given time, t. about max value for $\triangle t$ is used. Given that $\frac{\partial \phi}{\partial t}$ is defined by Equation 2.3 and the general update equation is

$$\phi(\mathbf{x}, t + \triangle t) = \phi(\mathbf{x}, t) + \triangle t \frac{\partial \phi}{\partial t}, \tag{2.5}$$

the level set update equation is

$$\phi(\mathbf{x}, t + \triangle t) = \phi(\mathbf{x}, t) + \triangle t F |\nabla \phi|. \tag{2.6}$$

The details of estimating $\nabla \phi$ and H are presented in Appendix A.

2.2 Narrow-Band Level-Set Solvers

Efficient algorithms for solving the more general equation rely on the observation that at any one time step the only parts of the solution that are important are those adjacent to the moving surface (near points where $\phi = 0$). This observation places level-set solvers as part of a larger class of solvers that efficiently operate on time-dependent, sparse computational domains—i.e., a subset of the original problem domain. However, in order to take advantage of the sparse nature of level-set solutions, algorithms must maintain a somewhat consistent *level-set density* (i.e., $\nabla \phi$), which is defined as the number of level sets per unit volume. If the level-set density becomes too low (spread out) it can become difficult to efficiently isolate the computation to the desired interface. Alternatively, a level-set density that becomes too high (close together) can cause aliasing and numerical problems. To address this, level-set solvers must manage the motion of the level sets, their density, and the position of the model relative to the desired computational domain. In general time-dependent, sparse algorithms maintain proper motion and density by iterating on the three steps shown in Figure 2.1.

Two of the most common CPU-based level-set solver techniques are the *narrow-band* [1] and *sparse-field* [57] methods. Both approaches limit the computation to a narrow region near the isosurface yet store the complete computational domain in memory. The narrow-band approach implements the initialization and update steps in Figure 2.1 (Steps 1 and 3) by updating the embedding, ϕ, on a band of 10-20 pixels around the model, using a signed distance transform implemented with the fast marching method [44]. The band is reinitialized whenever the model (defined as a particular level set) approaches the edge. In contrast, the sparse-field method traverses the complete domain only during

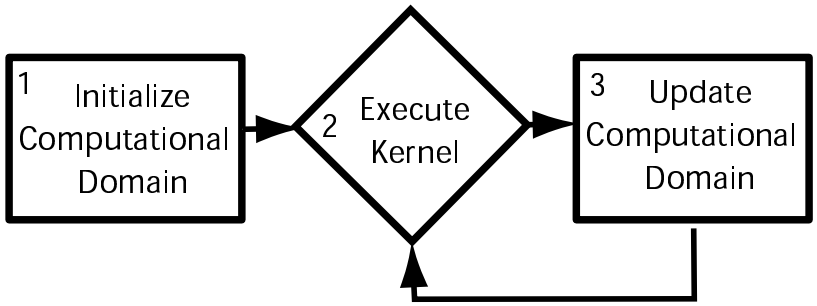


Figure 2.1. The three fundamental steps in a sparse-grid solver. Step 1 initializes the sparse computational domain. Step 2 executes the computational kernel on each element in the domain. Step 3 updates the domain if necessary. Steps 2 and 3 are repeated for each solver iteration.

the initialization step of the algorithm in Figure 2.1. The sparse-field approach keeps a linked list of active data elements. The list is incrementally updated via a distance transform after each iteration. A similar strategy is described in Peng et al. [36]. Even with this very narrow band of computation, update rates using conventional processors on typical resolutions (e.g., 256^3 voxels) are not interactive. This is the motivation behind the GPU-based, streaming narrow-band solver presented in this thesis.

2.3 Scientific Computation on Graphics Processors

Graphics processing units have been developed primarily for the computer gaming industry, but over the last several years researchers have come to recognize them as a low cost, high performance computing platform. Two important trends in GPU development, increased programmability and higher precision arithmetic processing, have helped to foster new nongaming applications.

For many data-parallel computations, graphics processors outperform central processing units (CPUs) by more than an order of magnitude because of their parallel *streaming* architecture [35] and dedicated high-speed memory. In the streaming model of computation, arrays of input data are processed identically by the same computation *kernel* to produce output data streams. The GPU takes advantage of the data-level parallelism inherent in the streaming model by having identical processing units execute the computation in parallel.

Although streaming architectures such as GPUs share a data-parallel design with

Cray-like vector computers [41] and massively parallel SIMD computers such as the Connection Machine system [16], they have several important differences. In contrast to vector architectures, which compute a single instruction on many data elements; the computation kernel in a streaming architecture may consist of many (possibly thousands) of instructions and use temporary registers to hold intermediate values. In contrast to Connection-Machine-like computers that contain thousands of small processing elements, each with their own small memory; GPUs use a relatively small number of processing elements (e.g., 8) that each have access to global memory as well as a small number of local registers (e.g., 32). Lastly, in addition to these architectural differences, the ubiquity and low price of GPUs (e.g., less than $500 U.S. dollars for a GPU in contrast to millions of dollars for a vector super-computer) means that millions of users and programmers have access to the platform. This large number of users makes the development of GPU-based algorithms especially warranted at this time.

Currently GPUs must be programmed via graphics APIs such as OpenGL [43] or DirectX [33]. Therefore all computations must be cast in terms of computer graphics primitives such as vertices, textures, texture coordinates, etc. Figure 2.2 depicts the computation pipeline of a typical GPU. Vertices and texture coordinates are first processed by the vertex processor. The rasterizer then interpolates across the primitives defined by the vertices and generates *fragments* (i.e., pixels). The fragment processor applies textures and/or performs computations that determine the final pixel value. A *render pass* is a set of data passing completely through this pipeline. It can also be thought of as the complete processing of a stream by a given kernel.

Grid-based computations are solved by first transferring the initial data into texture memory. The GPU performs the computation by rendering graphics primitives that access this texture. In the simplest case, a computation is performed on all elements of a two-dimensional texture by drawing a quadrilateral that covers the same number of grid points (pixels) as the texture. Memory addresses that identify each fragment's data value as well as the location of its neighbors are given as texture coordinates. A fragment program (the kernel) then uses these addresses to read data from texture memory, perform the computation, and write the result back to texture memory. A three-dimensional grid is processed as a sequence of two-dimensional slices. This computation model has been used by a number of researchers to map a wide variety of computationally demanding problems to GPUs. Examples include matrix multiplication, finite element methods, and

multi-grid solvers [13, 26, 50]. All of these examples demonstrate a homogeneous sequence of operations over a densely populated grid structure.

Strzodka et al. [40] were the first to show that the level-set equations could be solved using a graphics processor. Their solver implements the two-dimensional level-set method using a time-invariant speed function for flood-fill-like image segmentation, without the associated curvature. Their solver did not take advantage of the sparse nature of the level-set PDEs and therefore performs only marginally better than a highly-optimized sparse-field CPU implementation. The work in this thesis extends their work to three dimensions, adds in the second-order curvature computation, and significantly optimizes the solver by implementing a narrow-band solver on the GPU.

This thesis presents a GPU computational model that supports *time-dependent, sparse grid* problems. These problems are difficult to solve efficiently with GPUs for two reasons. The first is that in order to take advantage of the GPU's parallelism, the streams being processed must be large, contiguous blocks of data, and thus grid points near the level-set surface model must be *packed* into a small number of textures. The second difficulty is that the level set moves with each time step, and thus the packed representation must readily adapt to the changing position of the model. This requirement is in contrast to the recent sparse matrix solvers [4, 25] and previous work on rendering with compressed data [3, 24]. In the two sparse-matrix solvers[4, 25], a packed texture scheme is used to efficiently compute sparse matrix-vector multiplications as well as compute values of

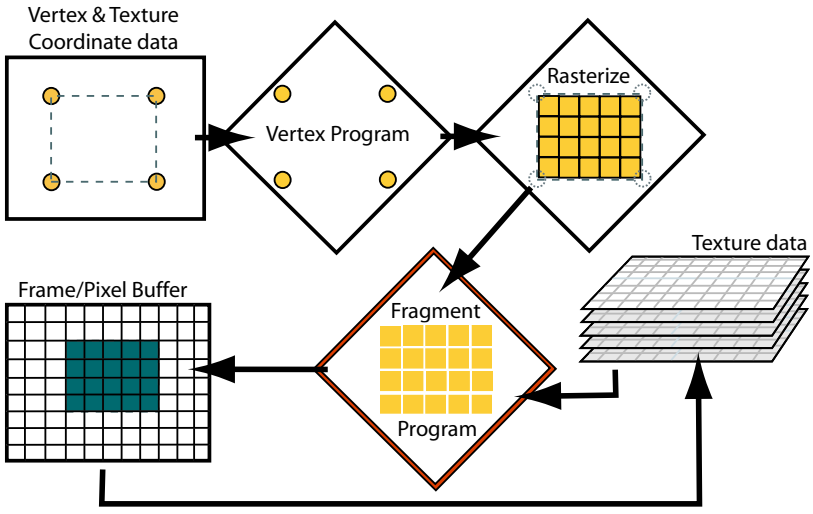


Figure 2.2. The modern graphics processor pipeline.

the sparse matrix elements on the GPU. The scheme is static, however, in the sense that the nonzero matrix elements must be identified before the computation begins. Recent work by Sherbondy et al. [47] describes an alternative time-dependent, sparse GPU computation model which is discussed in Chapter 4.3.

CHAPTER 3

STREAMING NARROW-BAND ALGORITHM

3.1 Introduction

This chapter describes GPU-based streaming algorithms for computing and visualizing the solution of the three-dimensional level-set partial differential equations. This new solver is 10 to 15 times faster than a highly-optimized CPU-based sparse-field implementation.

The first step toward creating a highly optimized GPU-based level-set solver was to create a brute force solution [30]. The details of this solver are given in Appendix B. This solver computes the level-set PDE at all voxels in the volume and is a direct extension of the two-dimensional work of Strzodka et al. [40]. This basic GPU-based, three-dimensional level-set solver runs one to two times faster than a highly optimized sparse-field CPU-based solver [52]. While this is not an impressive speedup, it is worth nothing that the GPU-based solver performs approximately 10 times more calculations than the optimized CPU-based one. As such, a narrow-band/sparse-field GPU-based solver should theoretically be able to achieve a 10–20 times speedup.

The proposed streaming, narrow-band level-set solver realizes these speedups by efficiently leveraging the capabilities of modern GPUs. The algorithm packs the active computational domain into two-dimensional texture memory, solves the three-dimensional, level-set PDE directly on this packed format, and quickly updates the packed data after each solver iteration.

The design of the streaming narrow-band algorithm takes into account several computational limitations of modern GPUs as well as the goal of interactive performance. First, the data-parallel computation model requires *homogeneous operations* on the entire computational domain. Second, memory constraints require an efficient algorithm to process *and store* only the active domain on the computational processor (i.e., the GPU). Third, GPUs do not support *scatter* write operations [38], and lastly, the communica-

tion bandwidth between the CPU and GPU is insufficient to allow transmission of any significant portion of the computational domain.

Section 3.2 describes a multidimensional virtual address scheme that efficiently maps the time-varying three-dimensional data into a two-dimensional texture. Section 3.3 then explains the details of the GPU-based level-set solver. In addition to explaining how the multidimensional virtual memory scheme is used in the solver, the section also introduces a new distance transform computation that can be efficiently performed on the GPU. The direct volume rendering of the deforming level-set surface is explained in Section 3.4.

3.2 A Virtual Memory Address Scheme for Sparse Computation

Remapping the computational domain (a subset of a volume) to take advantage of the GPU's capabilities has the unfortunate effect of making the computational kernels extremely complicated—that is difficult to design, debug, and modify. The kernel programmer must take the physical memory layout into consideration each time the kernel addresses memory. Other researchers have successfully remapped computational domains to efficiently leverage the GPU's capabilities [4, 13, 25, 38], but they invariably describe these complex kernels in terms of the physical memory layout. This section presents a solution to this problem that allows kernel programmers to access memory as if it were stored in the original (computational) domain—irrespective of its physical layout on the GPU. The solution is an extension to the virtual memory systems used in modern operating systems.

3.2.1 Traditional Virtual Memory Overview

Nearly all modern operating systems contain a virtual memory system [48]. The purpose of virtual memory is to give the programmer the illusion that the application has access to a contiguous memory address space, while allowing the operating system to allocate memory for each process on demand, in manageable increments, from whatever physical resources happen to be available. Note that there are two meanings of virtual memory. The first is the mapping from a logical address space to a physical address space. The second is the mechanism for mapping logical memory onto a physical memory hierarchy (e.g., main memory, disk, etc). For this discussion, virtual memory only refers to the former definition.

Virtual memory works by adding a level of indirection between physical memory and

the memory accessed by an application. Most conventional virtual memory systems divide physical and virtual memory into equally sized *pages*. The data addressed by an application's contiguous virtual address space will often be stored in many, disconnected physical memory pages. A *page table* tracks the mapping from virtual to physical memory pages. When an application requests memory, the system allocates physical memory pages and updates the page table.

When an application accesses memory via a virtual address, the system must first perform a virtual-to-physical address translation. The virtual address, VA, is first converted to a virtual page number, VPN. The system uses the page table to convert the VPN to a physical page address, PPA. The PPA is the physical address of the first element in a page. Finally, the memory system obtains the physical address, PA, by adding the PPA to the offset, OFF. The OFF is the linear distance between the virtual address and the beginning of the virtual page which contains it. The address computation is

$$\begin{array}{lcl} \text{VPN} & \leftarrow & \frac{\text{VA}}{S[\text{P}]} \\ \text{PPA} & \leftarrow & \text{PageTable(VPN)} \\ \text{OFF} & \leftarrow & \text{mod}(\text{VA}, \text{S[P]}) \\ \text{PA} & \leftarrow & \text{PPA} + \text{OFF}, \end{array} \tag{3.1}$$

where S[P] is the size of a memory page.

3.2.2 Multidimensional Virtual Memory for GPUs

The virtual memory system used in the proposed solver is a multidimensional extension of the traditional virtual memory system described in Section 3.2.1. This section begins by defining a general multidimensional virtual memory system and then describes details specific to the GPU implementation.

Traditional virtual memory systems use one-dimensional virtual and physical address spaces. While it is possible to generalize the algorithms described in Section 3.2.1 to an N-dimensional virtual address space and an M-dimensional physical address space, the practicalities of GPUs and the nature of the level-set problem space dictate the values of N and M. Specifically, GPUs are optimized to process two-dimensional memory regions ($M = 2$), while volumetric level-set computations are defined on a three-dimensional domain ($N = 3$). The design also make the simplifying assumption that virtual and physical pages are identical in dimension and size. Thus, the virtual space is not partitioned equally in all axes: two-dimensional pages must be stacked in three-dimensional to populate the problem domain as seen in Figure 3.1.

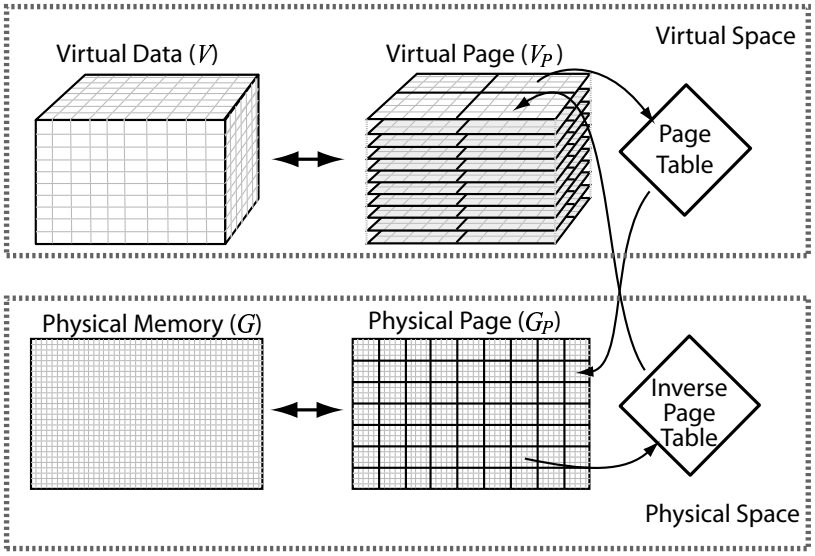


Figure 3.1. The multidimensional virtual and physical memory spaces used in the paged virtual address system. The original problem space is V, the virtual address space. The virtual page space, $\mathrm{V_P}$, is a subdivided version of V. Virtual memory pages are mapped to the physical page space, $\mathrm{G_P}$, by the *page table*. The *inverse page table* maps physical pages in $\mathrm{G_P}$ to virtual pages in $\mathrm{V_P}$. The collection of all elements in $\mathrm{G_P}$ constitute G, the physical memory of the hardware.

The discussion of the various address spaces involved in the multidimensional virtual address scheme requires a concise notation. To begin, the space of K-length vectors of integers is notated as $\mathbb{Z}^K$. The set of all voxels in the three-dimensional problem domain is the virtual address space, which is defined as $\mathrm{V} \subset \mathbb{Z}^3$. Each of the virtual memory pages is a set of contiguous voxels in V; the space of all virtual pages is $\mathrm{V_P}$ (Figure 3.1). Similarly, the physical address space, $\mathrm{G} \subset \mathbb{Z}^2$, is subdivided into pages to form the physical page space, $\mathrm{G_P}$. The elements within a virtual or physical page are addressed identically using elements of $\mathrm{P} \subset \mathbb{Z}^2$. In addition, a size operator is defined for the two-dimensional and three-dimensional spaces described above. For X in $\{\mathrm{V}, \mathrm{V_P}, \mathrm{G}, \mathrm{G_P}\}$, $S[X]$ is a two-vector or three-vector (according to the dimension of X) giving the number of elements along each axis of the space X. Note that $S[\mathrm{V_P}] = \mathrm{S[V]/S[P]}$ and $S[\mathrm{G_P}] = \mathrm{S[G]/S[P]}$ (using component-wise division). The level-set solver system uses pages of size $S[\mathrm{P}] = (16, 16)$. This size represents a good compromise between a tight fit to the narrow computational domain and the overhead of managing and computing pages. Empirical results validate this choice.

Virtual-to-physical address translation in an N-dimensional virtual memory system works analogously to the one-dimensional algorithm. Virtual addresses are now three-dimensional position vectors in V and physical addresses are two-dimensional vectors in G. The page table is a three-dimensional table that returns two-dimensional physical page addresses. With these multidimensional definitions in mind, Equation 3.1 still applies to the vector-valued quantities. Figure 3.2 shows an example multidimensional address translation.

For the level-set solver in this thesis, the multidimensional virtual memory system is implemented in part by the CPU and in part by the GPU. The CPU manages the page table, handles memory allocation/deallocation requests, and translates VPNs to PPAs. The GPU issues memory allocation/deallocation requests and computes physical addresses. The design further divides the GPU tasks between the various processors on the GPU. The fragment processor creates memory allocation/deallocation requests. The address translation implementation uses the vertex processor and rasterizer to compute all PAs. Sections 3.2.3 and 3.2.4 describe the architectural and efficiency reasons for assigning the various virtual memory tasks to specific processors.

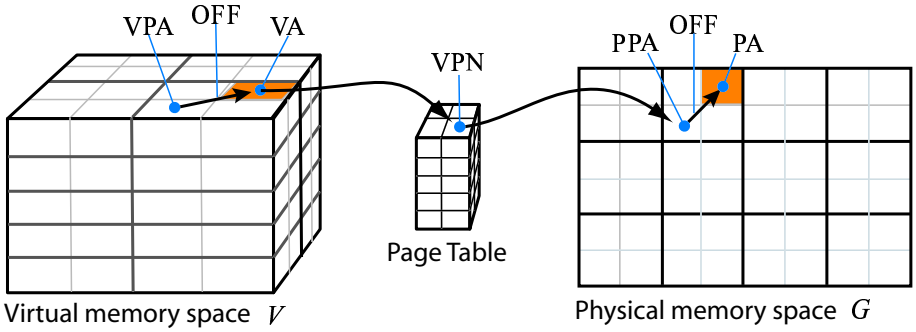


Figure 3.2. The virtual-to-physical address translation scheme in the multidimensional virtual memory system. A three-dimensional virtual address, VA, is first translated to a virtual page number, VPN. A page table translates the VPN to a physical page address, PPA. The PPA specifies the origin of the physical page containing the physical address, PA. The offset is then computed based from the virtual address and used to obtain the final two-dimensional physical address, PA.

3.2.3 Virtual-to-Physical Address Translation

This section explains the details of the virtual-to-physical address scheme used in the GPU-based virtual address system. Because the translation algorithm is executed each time the kernel accesses memory, its optimization is fundamental to the success of the method.

The simplest and most general way to implement the virtual-to-physical address translation for a GPU-based virtual memory system is to directly implement the computation in Equation 3.1 and store the page table on the GPU as a three-dimensional texture. A significant benefit of this approach is that it is completely general. Unfortunately, without dedicated memory-management hardware to accelerate the translation, this scheme suffers from several efficiency problems. First, the page table lookup means that a *dependent* texture read is required for each memory access. A dependent texture is defined as using the result of one texture lookup to index into another. This may cause a significant loss in performance on current GPUs. Second, storing the page table on the GPU consumes limited texture memory. The third problem is that a divide, modulus, and addition operation are required for each memory access. This consumes costly and limited fragment program instructions. Note that Section 3.2.4 discusses other problems with storing the page table on the GPU related to the limited capabilities of current GPU architectures.

The solver avoids the memory and computational inefficiencies that arise from storing the page table on the GPU by examining the pattern of virtual addresses required by the application's fragment program. In the case of the level-set solver, the fragment programs use virtual addresses within only a $3 \times 3 \times 3$ neighborhood of each active data element. This means that each active memory page will access only adjacent virtual memory pages (Figure 3.3). Moreover, the remainder of this section shows that this simplified translation case makes it possible to lift the entire address translation from the fragment processor to the vertex processor and rasterizer. The decision to reconstruct virtual neighborhoods on-the-fly rather than duplicate data lying on page boundaries is an important aspect of the system. The design choice was made to meet our original goals of minimizing memory usage, minimizing memory traffic, and maintaining square 16×16 memory pages.

Once the solver resolves the virtual addresses used by a fragment program, it can determine which virtual pages each active page will access. With this *relative page* information, the GPU can perform the virtual-to-physical address translation without

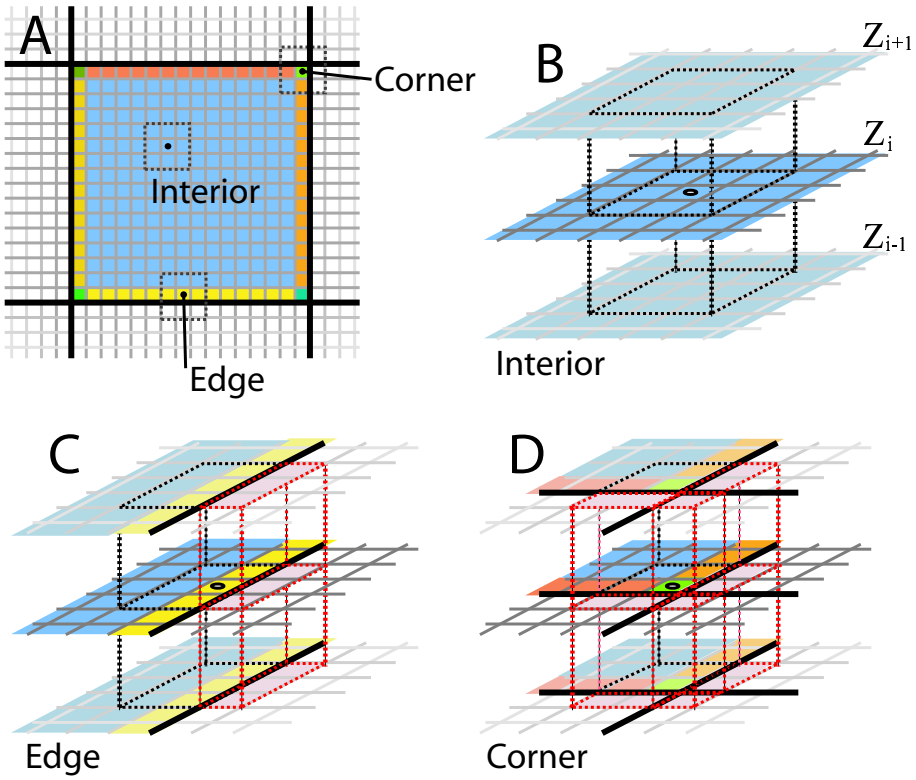


Figure 3.3. The *substream* boundary cases used to statically resolve the conditionals arising from $3 \times 3 \times 3$ neighbor accesses across memory page boundaries. The nine *substream* cases are: interior, four edges, and four corners (a). The interior case accesses its neighbors from only three memory pages (b). The edge cases require six pages (c), and the corner cases require 12 memory pages (d). Note that for reasonably large page sizes, the more cache-friendly interior case has by far the highest number of data elements.

a page table in texture memory. The CPU makes this possible by sending the PPAs for all required pages to the GPU as texture coordinates. The GPU can then use the relative neighbor offset vectors to decide which adjacent page contains the requested value (see Figure 3.3(a)).

The GPU's task of deciding which adjacent page contains a specific neighbor value unfortunately requires a significant amount of conditional logic. This logic must classify each data element into one of nine boundary cases: one of the four corners, one of the four edges, or an interior element (see Figure 3.3). Unfortunately current fragment processors do not support conditional execution. This logic could alternatively be encoded into a texture; however, this would again force the use of an expensive dependent texture read.

Just as statically resolving virtual addresses allowed the solver to optimize the GPU computation, all active data elements can be preclassified into the nine boundary cases. The result is that all memory addresses used in each case will lie on the same pages relative to each active page (see Figure 3.3). In other words, the memory-page-locating logic has been statically resolved by preclassifying data elements into their respective boundary cases. The data elements for these *substream* cases are generated by drawing unique geometry for each case. The corner substream cases are represented as points, the edges as lines, and the interior regions as quadrilaterals.

Kapasi et al. [18] describe an efficient solution to conditional execution in streaming architectures. Their solution is to route stream elements to different processing elements based on the code branch. Substreams are merely a static implementation of this data routing solution to conditional execution. The advantage is that the computation kernel run on each substream contains no conditional logic and is optimized specifically for that case. The solution additionally gains from optimized cache behavior for the most common, interior, case (77% of the data points in a 16×16 page). The interior data elements require only three memory pages to access all neighbors (Figure 3.3(b)). In comparison, reading all neighbors for an edge element requires loading six pages (Figure 3.3(c)). The corner cases require 12 pages from disparate regions of physical memory(Figure 3.3(d)). The corner cases account for less than 2% of the active data elements.

With the use of substreams, the GPU can additionally optimize the address computation by computing physical addresses with the vertex processor rather than the fragment processor. Because all data elements (i.e., fragments) use exactly the same relative memory addresses, the offset and physical address computation steps of Equation 3.1 can be generated by interpolating between substream vertex locations. The vertex processor and rasterizer can thus perform the entire address translation. This optimization distributes computational load to underutilized processing units and reduces the number of limited and expensive fragment instructions.

The algorithm described above is a highly optimized address translation scheme for evaluating neighborhoods of $3 \times 3 \times 3$. Many applications, however, require the use of larger neighborhoods. The substream and vertex processor optimizations described above will, to a limited extent, generalize to neighborhoods larger than $3 \times 3 \times 3$. To process larger neighborhoods, a separate set of substreams would need to be generated for each *layer* of grid points adjacent to memory page boundaries. Theoretically, neighborhoods

as large as one half of the page size could be processed with the technique, although there may be a neighborhood size beyond which the cost of splitting the computation into many small substreams outweighs the benefits. A more general technique, such as performing the full address computation in the fragment stage (as described at the beginning of this section), may be more advantageous for processing large neighborhoods.

3.2.4 Bootstrapping the Virtual Memory System

This section describes the steps required to initialize the GPU virtual memory system. To begin, the application specifies the page size, S[P], the virtual page space size, $S[V_P]$, and the fundamental data type to use (i.e., 32-bit floating point, 16-bit fixed point, etc.). The virtual memory system then allocates an initial physical memory buffer on the GPU. It also creates a page table, an inverse page table, a geometry engine, and a stack of free pages on the CPU. The decision to place the aforementioned data structures on the CPU is based on the efficiency concerns described in Section 3.2.3 as well as GPU architectural restrictions. These restrictions include: the GPU's lack of random write access to memory, lack of writable three-dimensional textures, lack of dynamically sized output buffers, and limited GPU memory.

The page table is defined to store a `MemoryPage` object that contains the vertices and texture coordinates required by the GPU to access the physical memory page. The inverse page table is designed to store a VPN vector for each active physical page. Figure 3.2 shows these mappings. Note that the page table and inverse page table were referred to as the *unpacked map* and *packed map* respectively in Lefohn et al. [29].

The vertices and texture coordinates stored in the `MemoryPage` object are actually pointers into the geometry engine. The geometry engine has the capability of quickly rendering (i.e., processing) any portion of the physical memory domain. Thus the geometry engine must generate the substreams for the set of active physical pages. The last initialization step is the creation of the free-page stack. The virtual memory system simply pushes all physical pages (i.e., pointers to `MemoryPage` objects) defined by the geometry engine onto a stack.

The application issues GPU physical memory allocation and deallocation requests to the virtual memory system. Upon receiving a virtual page request, the system pops a physical page from the free-page stack, updates the page tables, and returns a `MemoryPage` pointer to the application. The reverse process occurs when the application deallocates a virtual memory page.

The level-set solver generates memory page allocation and deallocation requests after each solver iteration based on the form of the current solution. Section 3.3.5 describes how the solver uses the GPU to efficiently create these memory requests.

3.3 Streaming Narrow-Band GPU Level-Set Solver

This section explains how the GPU level-set solver implementation uses the virtual memory system described in Section 3.2 to create an efficient streaming narrow-band solver. The full details of the level-set equations are not given here, but are instead found in Appendix A.

3.3.1 Initialization of Computational Domain

The solver begins by initializing the sparse computational domain (Step 1 in Figure 2.1). An initial level-set volume is passed to the level-set solver by the level-set application. The sparse domain initialization involves identifying active memory pages in the input volume, allocating GPU memory for each active page, then sending the initial data to the GPU (Figure 3.4).

The solver identifies active virtual pages by checking each data element for a nonzero derivative value in any of the six cardinal directions. If any element in a page contains nonzero derivatives, the entire page is activated. The initialization code then requests a

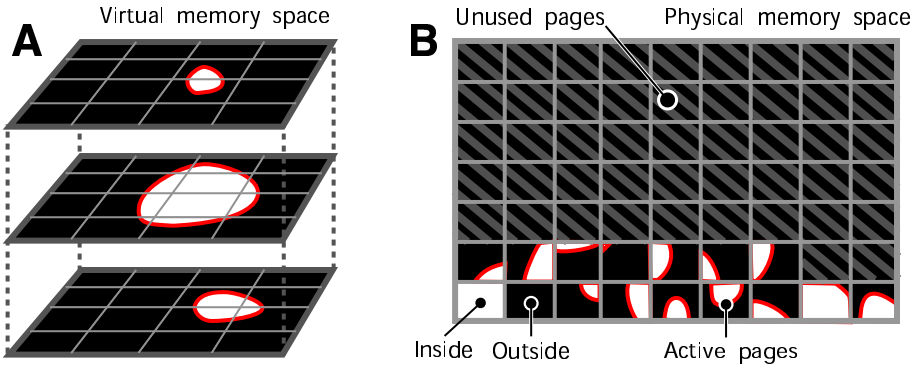


Figure 3.4. The level-set solver's use of the paged virtual memory system. All *active* pages (i.e., those that contain nonzero derivatives) in the virtual page space (a) are mapped to unique pages of physical memory (b). The inactive virtual pages are mapped to the static *inside* or *outside* physical page. Note that the only data stored on the GPU is that represented by (b).

GPU memory page from the virtual memory system for each active page. The level-set data is then *drawn* into GPU memory using the vertex locations in each `MemoryPage` object.

This scheme is effective only because the input level-set volume is assumed to be a clamped distance transform—meaning that regions on or near the isosurface have nonzero gradients while regions outside or inside the surface have gradients of zero (see Figure 3.5). The outside voxels have a value of zero (black) and the inside ones have a value of one (white). The algorithm described in Section 3.3.2 describes how the clamped distance transform is maintained during the level-set computation.

The inactive virtual pages do not need to be represented in physical memory. If an active data element queries an inactive value, however, an appropriate value needs to be returned. Because all inactive regions are either uniformly black or white, the system handles this boundary condition problem by defining a special, inactive page state. A virtual page in this state is mapped to one of two *static* physical pages. One of these static pages is black, representing regions outside of the level-set surface. The other static page is white and represents regions inside the level-set surface. The page table contains these many-to-one mappings, but the inverse page table does not store a valid entry for the static pages. Note that this boundary problem could have alternatively been solved using single pixels instead of entire pages; however, this lack of uniformity in memory page sizes would have complicated the page table representations. Alternatively, the problem

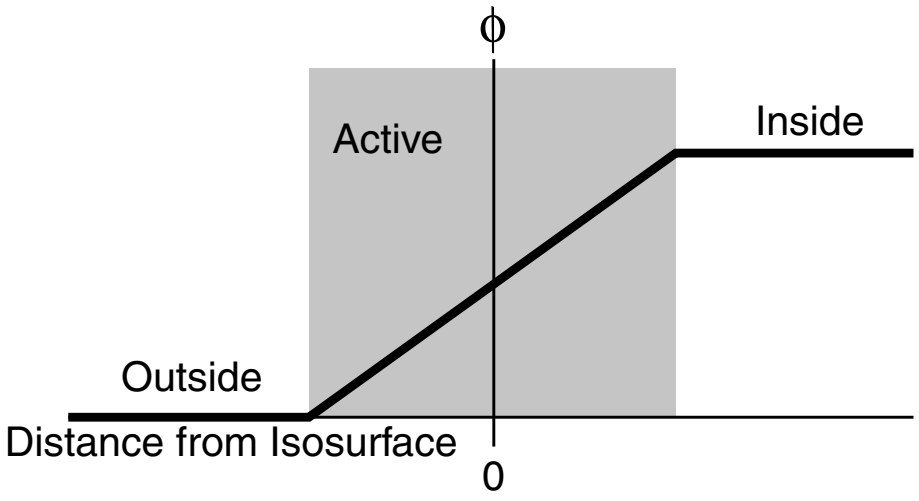


Figure 3.5. The level-set embedding, ϕ, is a clamped distance transform, i.e., $|\nabla\phi|$ is nonzero near the surface model and zero elsewhere.

could have been solved by creating substreams for the active elements on the boundary of the active set; however, this would have unnecessarily added a number of additional substream render passes to the computation.

3.3.2 The Distance Transform Computation on the GPU

The GPU-based level-set solver borrows ideas from both the narrow-band and sparse-field algorithms, but implements a new solution that conforms to the architectural restrictions of GPUs. Both of these previous, CPU-based, methods maintain a distance-transform embedding (i.e., manage level-set density) by a series of heterogeneous operations that are not particularly efficient on the GPU. In order to solve this problem, the streaming level-set method maintains a distance-transform embedding by introducing an additional speed term $\mathrm{G_r}$ into the level-set PDE Equation 2.3. This additional speed term *pushes* the level sets of ϕ, either closer together or farther apart, so that they resemble an appropriately scaled clamped distance transform (CDT). The CDT has a constant level-set density within a predefined band and ensures that voxels near the isosurface have finite derivatives while those farther away have gradient magnitudes of zero. As described in the proceeding section, the identification of zero-derivative regions is critical for an efficient solver implementation. This *rescaling* speed term, $\mathrm{G_r}$, is computed as

$$\mathrm{G_r} = \phi \mathrm{g}_\phi - \phi|\nabla\phi|, \tag{3.2}$$

where g_ϕ is the target gradient magnitude within the computational domain. This target parameter is set based on the numerical precision of the level-set data. By setting g_ϕ sufficiently high, numerical errors caused by underflow can easily be avoided. It is important to note that $\mathrm{G_r}$ is strictly a numerical construct; it does not affect the movement of the zero level set, i.e., the surface model. This embedding-rescaling computation is similar to the technique discussed in Fedkiw et al. [12].

In conclusion, Equation 3.2 has the following three properties. First $\mathrm{G_r}$ is proportional to ϕ (i.e., $\mathrm{G_r}$ approaches zero as ϕ approaches zero), and therefore adding $\mathrm{G_r}$ to the speed terms in the level-set computation will not move the level-set surface (assuming $k = 0$). Second, because the up-wind scheme [34] maintains monotonicity in the embedding, no new extrema will be created. As such, the clamping properties of the original embedding will be maintained. Lastly the fixed point of $\mathrm{G_r}$ is the distance transform scaled by g_ϕ.

3.3.3 Level-Set Computation

The GPU next computes the level-set computation (Step 2 of the sparse algorithm, Figure 2.1). The details of the level-set discretization used by the level-set solver are given in Appendix A. This section gives a high-level overview of the computation. The level-set update proceeds in the following steps:

A Compute 1st and 2nd partial derivatives.
B Compute N level-set speed terms.
C Update level-set PDE.

The derivative passes in Step A above use the substream-based, virtual-to-physical address scheme described in Section 3.2.3. The derivatives are computed in nine substream passes, each of which outputs to the same four buffers. The speed function computations in Step B are application-dependent. Example speed terms include the curvature computation described in Equation 2.4, the rescaling term described in Equation 3.2, and the data-dependent term described in Equation 4.1. There will be zero or more render passes for each speed function. The level-set update (Step C) is simply the up-wind scheme described in Appendix A, which is computed in a single pass. Note that additional GPU memory must be allocated to store the intermediate results accumulated in Steps A and B before they are consumed in Step C. The solver also performs register allocation of temporary buffers to minimize GPU memory usage.

3.3.4 GPU Implementation Details

The level-set solver and volume renderer are implemented in programmable graphics hardware using vertex and fragment programs on the ATI Radeon 9800 GPU. The programs are written in the OpenGL ARB_vertex_program and ARB_fragment_program assembly languages.

Several details related to render pass output buffers are critical to the performance of the level-set solver. First is the ability to output multiple, high-precision 4-tuple results from a fragment program. Writing 16 scalar outputs from a single render pass enables the solver to perform the expensive three-dimensional neighborhood reconstruction only once and use the gathered data to compute the derivatives in a single pass. Second, the solver avoids the expensive change between render targets [23] (i.e., pixel buffers) by allocating a single pixel buffer with many *render surfaces* (front, back, aux0, etc.) and using each surface as a separate output buffer.

Lastly, there is a subtle speed-versus-memory trade-off that must be carefully considered. Because the physical-memory texture can be as large as 2048^2, storing intermediate results (e.g., derivatives, speed values, etc.) during the computation can require a large amount of GPU memory. This memory requirement can be minimized by performing the level-set computation in subregions. The intermediate buffers must then be only the size of the subregion. This partitioning does reduce computational efficiency; however, and so the subregions are made as large as possible. The solver currently use 512^2 subregions when the level-set texture is 2048^2 and use a single region when it is smaller.

3.3.5 Update of Computational Domain

After each level-set update, the solver determines which virtual pages need to be added-to or removed-from the active domain. The solver accomplishes this by aggregating gradient information from all elements in each active page. The GPU must compute this information because the level-set solution exists only in physical memory. The active set must be updated by the CPU, however, because the page table and geometry engine exist in CPU main memory. In addition, the amount of information passed from the GPU to the CPU must be kept to a minimum because of the limited bandwidth between the two processors. This section gives an overview of an algorithm that works within these constraints. Appendix B explains the full details of the algorithm.

The GPU creates a memory allocation/deallocation request by producing a small image (of size $S[\mathrm{G_P}]$) with a single-byte pixel per physical page. The value of each pixel is a bit code that encapsulates the activation or deactivation state of each page and its six adjacent neighbors (in $\mathrm{V_P}$). The CPU reads this small (< 64kB) message, decodes it, and submits the allocation/deallocation requests to the virtual memory system (Figure 3.6).

The GPU creates the bit-code image by first computing two, four-component neighbor information buffers of size $S[\mathrm{G}]$ (Step A of Figure 3.6). This computation uses the previously-computed, one-sided derivatives of ϕ to identify the required active pages. A page must be activated if it contains elements with nonzero gradient magnitudes. The automatic mipmapping GPU feature is used to down-sample the resulting buffers (i.e., aggregate data samples) to the page-space image (Step B in Figure 3.6). The final GPU operation combines the active page information into the bit code (Step C in Figure 3.6). A fragment program performs this step by emulating a bit-wise OR operation via conditional addition of powers of two. Finally, in step D of Figure 3.6, the CPU reads this message

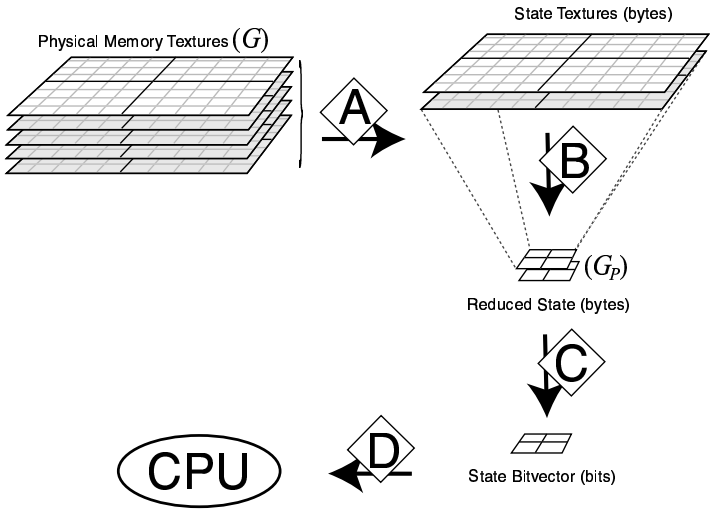


Figure 3.6. The GPU's creation of a memory allocation/deallocation request. Step A uses solver-specific data to create two buffers containing the active state of each data element and its adjacent neighbors. Step B uses automatic mipmapping to reduce the buffers from size $S[\mathrm{G}]$ to the physical page space size, $S[\mathrm{G_P}]$. Step C combines the information from the two down-sampled state buffers into an eight-bit code for each pixel. This code encapsulates whether or not each active virtual memory page and its adjacent neighbors should be enabled. In step D, the CPU reads the bit-code buffer, decodes it, and allocates/deallocates pages as requested.

from the GPU.

Note that the use of automatic mipmapping places some restrictions on the maximum tile size due to quantization rounding errors that arise when down-sampling 8-bit values. This limitation can be relaxed by using a 16-bit fixed-point data type. Alternatively, floating-point values can be used if the down-sampling is performed with fragment program passes instead of automatic mipmapping.

3.4 Volume Rendering of Packed Data

The direct visualization of the level set evolution is important for a variety of level-set applications. For instance, in the context of segmentation, direct visualization allows a user to immediately assess the quality and accuracy of the pending segmentation and steer the evolution toward the desired result. Volume rendering [10, 31, 42] is a natural choice for visualizing the level-set surface model, because it does not require an intermediate geometric extraction, which might limit interactivity. If one were to use

marching cubes, for instance, a distinct triangle mesh would need to be created (and rendered) for each iteration of the level-set solver. The implemented solver, therefore, includes a volume renderer, which produces a full three-dimensional (transfer-function based) volume rendering of the evolving level set on the GPU [22].

For rendering the evolving level-set model, the new renderer use a variant of traditional two-dimensional texture based volume rendering [6]. The renderer modifies the conventional approach to render the level-set solution directly from the packed physical memory layout, which is physically stored in a single two-dimensional texture. Because the level-set data and physical page configurations are dynamic, it would be inefficient to precompute and store the three separate versions of the data, sliced along cardinal views, as is typically done with two-dimensional texture approaches. Instead the renderer reconstructs these views each time the volume is rendered. Note that this new technique also enables volume rendering from a dataset stored in a *single* set of two-dimensional slices.

The volume rendering algorithm utilizes a two pass approach for reconstruction and rendering. Figure 3.7 illustrates the steps involved. An additional off-screen buffer caches two reconstructed neighboring slices containing the level set solution and its gradient (Figure 3.7 A). During the rendering phase arbitrary slices along the preferred slice direction are interpolated from these neighboring slices (Figure 3.7 B). Once all interpolated slices between slice i and $i-1$ are rendered and composited, the next slice $(i+1)$ is reconstructed. This newly reconstructed slice replaces the cached slice, $i-1$. The GPU then renders and composites the interpolated slices (i.e., those between slice $i+1$ and i). This pattern continues until all slices have been reconstructed and rendered.

When the preferred slice axis, based on the viewing angle, is orthogonal to the virtual memory page layout, the renderer reconstructs two-dimensional slices of the level set solution and its gradient using a textured quadrilateral for each page, as shown in Figure 3.8 A. On the other hand, if the preferred slice direction is parallel to the virtual page layout, the algorithm renders a row or column from each page using textured line primitives, as in Figure 3.8 B. In both cases, slices are reconstructed into a pixel buffer which is bound as a texture in the rendering pass. These slices are reconstructed at the same resolution as level set solution. For efficiency, the renderer reuses data wherever possible. For instance, lighting for the level-set surface, evaluated in the rendering phase, uses gradient vectors computed during the level-set update stage.

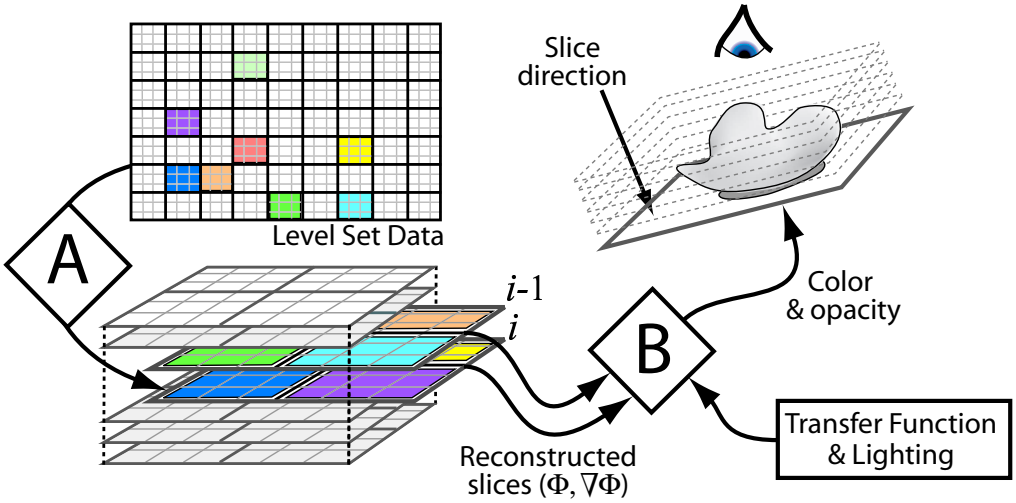


Figure 3.7. Two pass rendering of packed volume data. In step A, a two-dimensional slice (i) is reconstructed from the physical page (packed) layout, G_P. In step B, one or more intermediate slices between i and $i-1$ are interpolated, transformed into optical properties (via the transfer function), lit, and rendered for the current view. The next iteration begins by reconstructing slice $i+1$, replacing $i-1$, and so on.

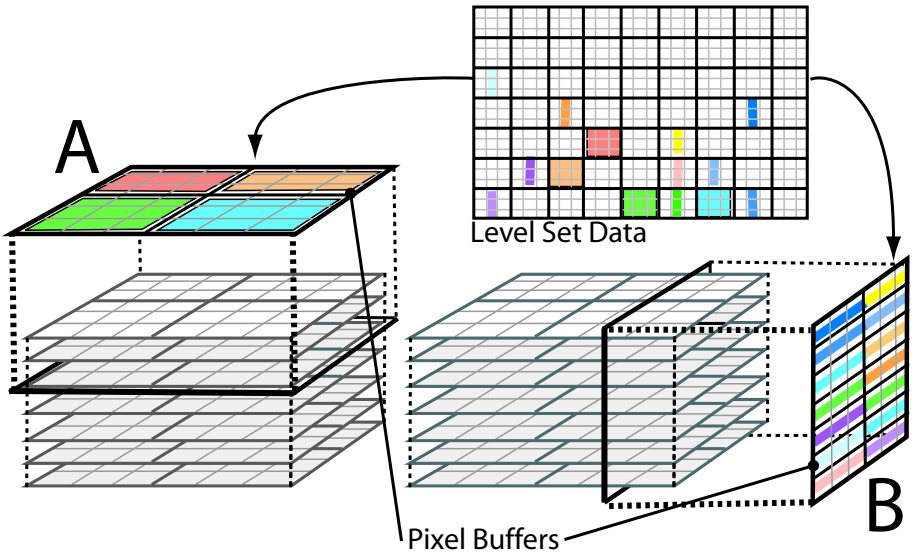


Figure 3.8. Reconstruction of a slice for volume rendering the packed level-set model: (a) When the preferred slicing direction is orthogonal to the virtual memory page layout, the pages (shown in alternating colors) are draw into a pixel buffer as quadrilaterals. (b) For slicing directions parallel to the virtual page layout, the pages are drawn onto a pixel buffer as either vertical or horizontal lines.

In the rendering phase, the algorithm leverages the hardware's bilinear filtering for in-plane interpolation of the reconstructed level set slice. Trilinear interpolation of an arbitrary slice between two adjacent reconstructed slices is accomplished by combining them, *i.e.* performing linear interpolation along the preferred slice direction, in the fragment program. This same fragment program also evaluates the transfer function and lighting for the interpolated data.

CHAPTER 4

SEGMENTATION APPLICATION

4.1 Introduction

Segmentation is an important part of volume visualization and analysis. In the IEEE Visualization 2002 panel entitled "Volume Rendering in Medical Applications", Bill Lorensen of General Electric Research and Development made an important observation about volume rendering: "Its time to move beyond pretty pictures and move more toward image analysis." With the rising importance of quantitative volume analysis, will come an increased role for tools that utilize visualization to achieve better quantitative results.

This chapter describes such a tool; an interactive volume segmentation and visualization application that uses the GPU-based level-set solver and volume rendering techniques described in Chapter 3. Section 4.2 describes the details of the application, while Section 4.3 presents a performance analysis of the system. Section 4.4 presents a user study uses that evaluates the effectiveness of the interactive segmentation tool for medical segmentation.

4.2 Volume Segmentation and Visualization Application

4.2.1 Level-Set Formulation for Segmentation

For segmenting volume data with level sets, the speed usually consists of a combination of two terms [32, 56]

$$\frac{\partial \phi}{\partial t} = |\nabla \phi| \left[\alpha D(\bar{x}) + (1-\alpha) \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} \right], \tag{4.1}$$

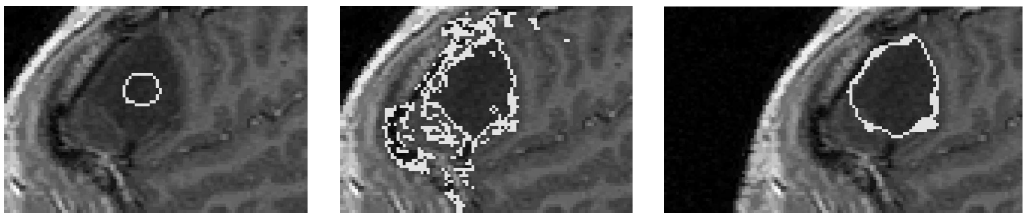
where D is a data term that forces the model to expand or contract toward desirable features in the input data (which is also called the *source* data), the term $\nabla \cdot (\nabla \phi / |\nabla \phi|)$ is the mean curvature H of the surface, which forces the surface to have less area (and remain smooth), and $\alpha \in [0, 1]$ is a free parameter that controls the degree of smoothness in the solution.

This combination of a data-fitting speed function with the curvature term is critical to the application of level sets to volume segmentation. Most level-set data terms D from the segmentation literature are equivalent to well-known algorithms such as isosurfaces, flood fill, or edge detection when used without the smoothing term (i.e., $\alpha = 1$). The smoothing term alleviates the effects of noise and small imperfections in the data, and can prevent the model from leaking into unwanted areas (Figure 4.1). In the context of volume analysis, the level-set surface models provide several capabilities that complement volume rendering: local, user-defined control; smooth surface normals for better rendering of noisy data; and a closed surface model, which can be used in subsequent processing or for quantitative shape analysis.

For the work in this thesis the segmentation application uses a simple speed function to demonstrate the effectiveness of *interactivity* and *real-time visualization* in level-set solvers. The speed function created for this work depends solely on the greyscale value input data I at the point $\bar{x}$:

$$D(I) = \epsilon - |I - T|, \tag{4.2}$$

where T controls the brightness of the region to be segmented and ϵ controls the range of greyscale values around T that could be considered inside the object. In this way a model situated on voxels with greyscale values in the interval $T \pm \epsilon$ will expand to enclose that voxel, whereas a model situated on greyscale values outside that interval will contract to exclude that voxel. The speed term is gradual, as shown in Figure 4.2, and thus the effects of the D diminish as the model approaches the boundaries of regions with greyscale levels within the $T \pm \epsilon$ range, and the effects of the curvature term will be relatively larger. This



(a) (b) (c)

Figure 4.1. The use of a curvature constraint (speed function) in the level-set computation to prevent segmentation "leaking." This example shows one slice of a three-dimensional MRI segmentation computation: (a) The spherical initialization. (b) A model expands to fill the tumor but leaks through gaps and expands into other anatomy. (c) The same scenario with a degree of curvature prevents unwanted leaking. The level set isosurface is shown in white.

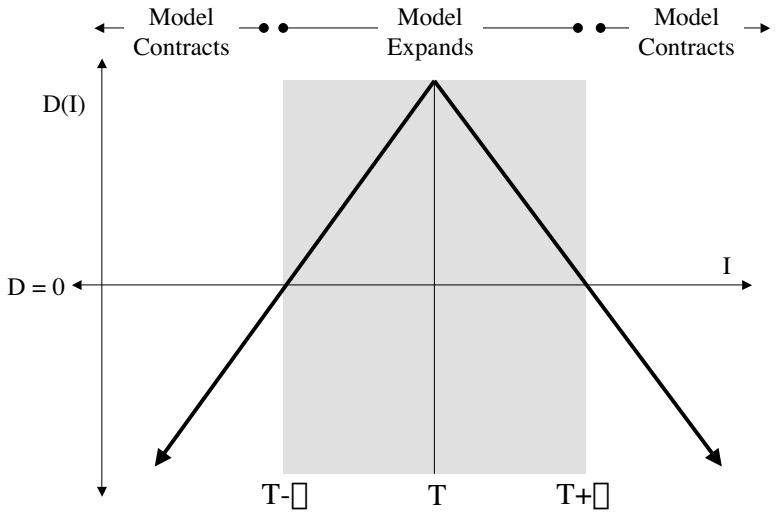


Figure 4.2. A speed function based on image intensity causes the model to expand over regions with greyscale values within the specified range and contract otherwise.

choice of D corresponds to a simple, one-dimensional statistical classifier on the volume intensity [28].

To control the model a user specifies three free parameters, T, ϵ, and α, as well as an initialization. The user generally draws a spherical initialization inside the region to be segmented. Note that the user can alternatively initialize the solver with a preprocessed (thresholded, flood filled, etc.) version of the source data.

4.2.2 Interface and Usage

The application in this thesis consists of a graphical user interface (GUI) that presents the user with two slice viewing windows, a volume renderer, and a control panel (Figures 4.3 and 4.4). Many of the controls are duplicated throughout the windows to allow the user to interact with the data and solver through these various views. Two and three-dimensional representations of the level-set surface are displayed in real time as it evolves.

The first two-dimensional window displays the current segmentation as a yellow line overlaid on top of the source data. The second two-dimensional window displays a visualization of the level-set speed function that clearly delineates the positive (blue) and negative (black) regions. The first window can be probed with the mouse to accomplish three tasks: set the level set speed function, set the volume rendering transfer function, and draw three-dimensional spherical initializations for the level-set solver. The first two

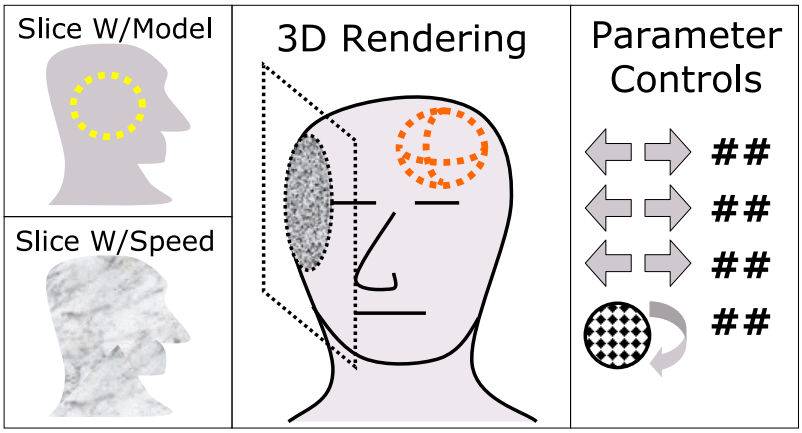


Figure 4.3. A depiction of the user interface for the volume analysis application. Users interact via slice views, a three-dimensional rendering, and a control panel.

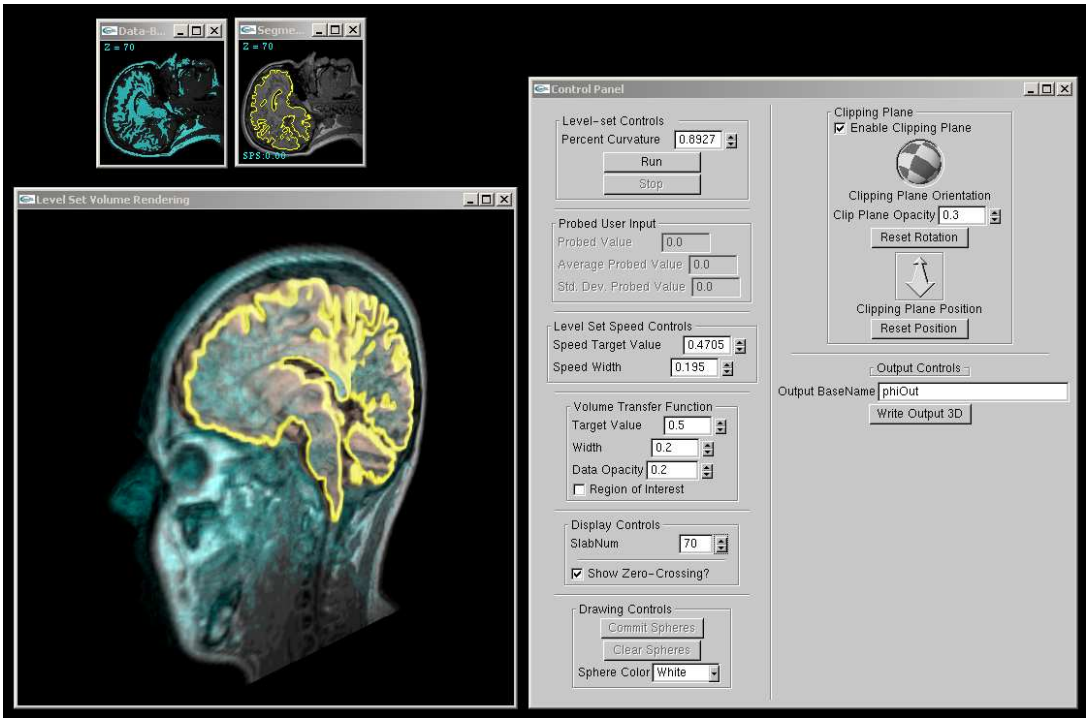


Figure 4.4. The actual user interface for the volume analysis application. The top left window shows the visualization of the speed function. The top right window shows a slice of the MRI source data with the current level-set solution in yellow. The lower-left window shows a volume rendering of the MRI source data (blue), the same data projected onto a clipping plane (grey), the current level-set surface (brown), and the intersection of the current level-set solution with the clipping plane (yellow).

are accomplished by accumulating an average and variance for values probed with the cursor. In the case of the speed function, the T is set to the average and ϵ is set to the standard deviation. Users can modify these values, via the GUI, while the level set deforms. The spherical drawing tool is used to initialize and/or edit the level-set surface. The user can place either white (model on) or black (model off) spheres into the system.

The volume renderer displays a three-dimensional reconstruction of the current level-set isosurface (see Chapter 3.4) as well as the input data. In addition, an arbitrary clipping plane, with texture-mapped source data, can be enabled via the GUI (Figure 4.4). Just as in the slice viewer, the speed function, transfer function, and level-set initialization can be set through probing on this clipping plane. The crossing of the level-set isosurface with the clipping plane is also shown in bright yellow.

The volume renderer uses a two-dimensional transfer function to render the level set surface and a three-dimensional transfer function to render the source data. The level-set transfer function axes are intensity and distance from the clipping plane (if enabled). The transfer function for rendering the original data is based on the source data value, gradient magnitude, and the level-set data value. The latter is included so that the level set model can function as a region-of-interest specifier. All of the transfer functions are evaluated on-the-fly in fragment programs rather than in lookup tables. This approach permits the use of arbitrarily high-dimensional transfer functions, allows run-time flexibility, and reduces memory requirements [23]. The GUI has additional controls for starting/stopping the solver, enabling a region-of-interest volume rendering mode, setting opacity of the volume and clipping plane, and saving the three-dimensional segmentation to file.

The interactive level-set solver and volume rendering system is demonstrated with the following three data sets: a brain tumor MRI (Figure 4.5, 4.6), an MRI scan of a mouse (Figure 4.7), and transmission electron tomography data of a gap junction (Figure 4.8). In all of these examples a user interactively controls the level-set surface evolution and volume rendering via the multiview interface. The initializations for the tumor and mouse were drawn via the user interface while the gap junction solution was seeded with a thresholded version of the source data.

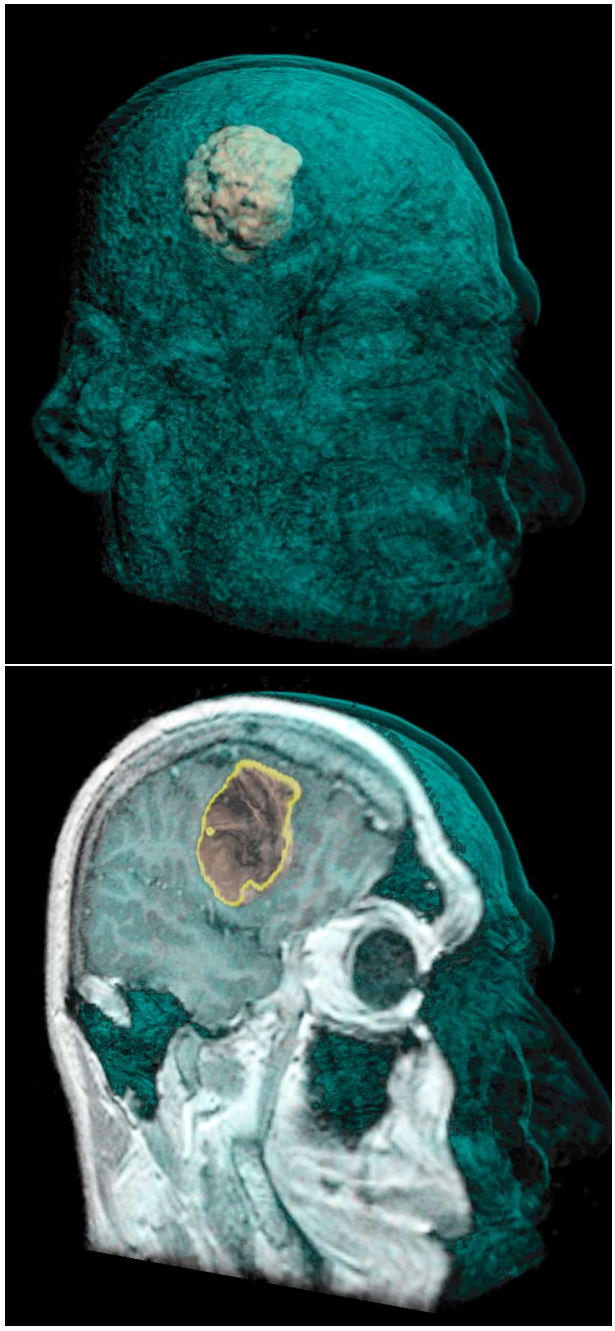


Figure 4.5. Interactive level-set segmentation of a brain tumor from a $256 \times 256 \times 198$ MRI with volume rendering to give context to the segmented surface. A clipping plane (bottom) shows the user the source data, the volume rendering, and the segmentation simultaneously. The segmentation and volume rendering parameters are set by the user probing data values on the clipping plane.

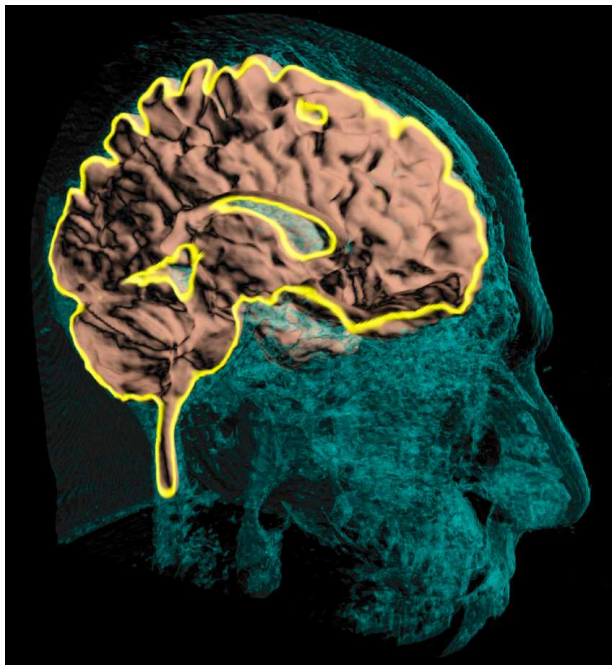


Figure 4.6. Interactive level-set segmentation of the cerebral cortex from a 256×256×198 MRI with volume rendering to give context to the segmented surface. The MRI data is also projected onto a clipping plane, on which the user can probe to control the level-set parameters.

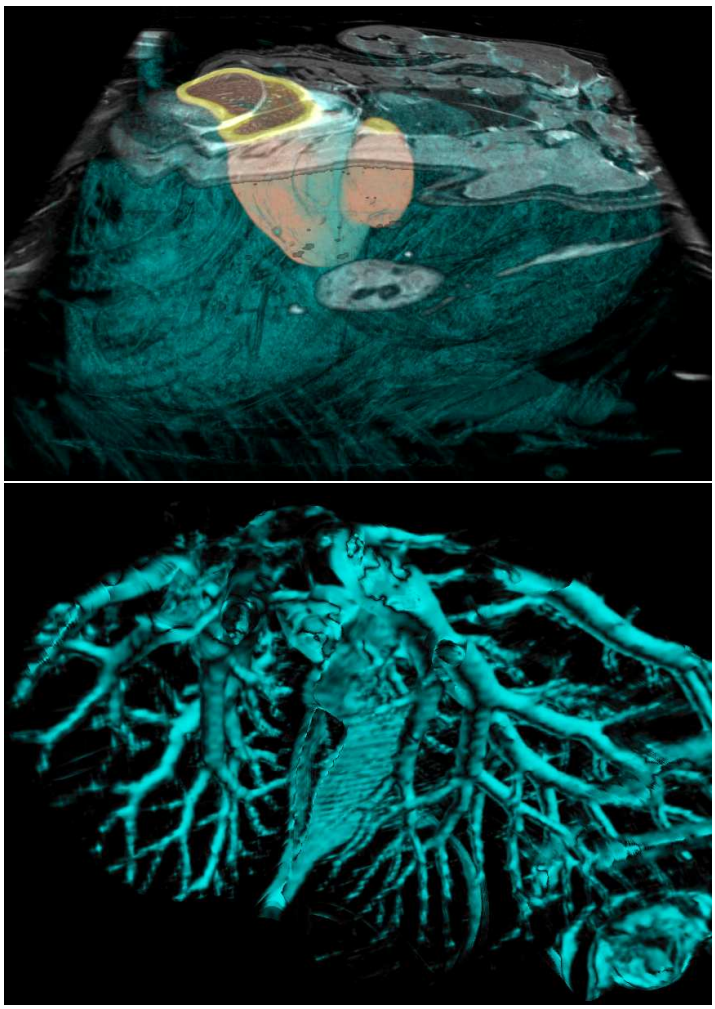


Figure 4.7. The top image shows a volume rendering of a 256^3 MRI scan of a mouse thorax. Note the level set surface which is deformed to segment the liver. The bottom image shows a volume rendering of the vasculature inside the liver. Both images are rendered using the same transfer function with the level-set surface serving as a region-of-interest specifier.

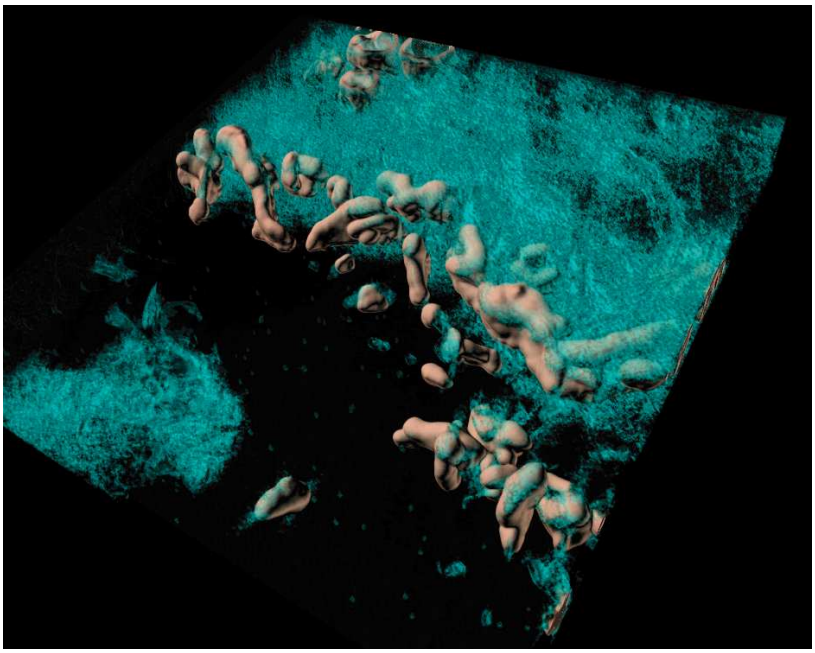


Figure 4.8. Segmentation and volume rendering of $512 \times 512 \times 61$ three-dimensional transmission electron tomography data. The picture shows cytoskeletal membrane extensions and connexins (pink surfaces extracted with the level-set models) near the gap junction between two cells (volume rendered in cyan).

4.3 Performance Analysis

The GPU-based level-set solver achieves a speedup of 10–15 times over a highly-optimized, sparse-field, CPU-based implementation [52]. The user study presented in Section 4.4 demonstrates that the new solver runs interactive rates for the tumor segmentations performed in the study. Interactivity is defined here as being fast enough that the segmentation times are almost entirely based on user time rather than solver time. Alternatively, users generally regard a solver running at rates greater than steps per second as interactive.

All benchmarks were run on an Intel Xeon 1.7 GHz processor with 1 GB of RAM and an ATI Radeon 9800 Pro GPU. All timings include the complete computation, i.e., both the virtual memory system update and the level-set computation are included. For a $256 \times 256 \times 175$ volume, the level-set solver runs at rates varying from 70 steps per second for the tumor segmentation to 3.5 steps per second for the final stages of the cortex segmentation (Figure 4.5). In contrast, the CPU-based, sparse field implementation ran at 7 steps per second for the tumor and 0.25 steps per second for the cortex segmentation.

The speed of the solver is approximately 80% dependent on the core clock rate of the GPU, 15% dependent on the GPU's memory speed and only 5% dependent on the speed of the AGP bus. These dependency measures were obtained by measuring the solver's computation rate while changing the GPU's core and memory clock speeds [51] and by changing the speed of the AGP bus. These and other profiling techniques are described by NVIDIA [9]. Note that the 80% dependence on core clock speed and 15% dependence on memory speed indicate that the speed of the solver will continue to improve as GPUs increase in speed and/or add additional computational elements.

The speed of the solver is bound almost entirely by the fragment stage of the GPU. In addition, the speed of the solver scales linearly with the number of active voxels in the computation. Creation of the bit vector message consumes approximately 15% of the GPU arithmetic and texture instructions, but for most applications the speedup over a dense GPU-based implementation far eclipses this additional overhead.

The amount of texture memory required for the level-set computation is proportional to the surface area of the level-set surface—i.e., the number of active pages. Tests have shown that for many applications, only 10%-30% of the volume is active. To take full advantage of this savings, the total size of physical memory, $S[\mathrm{G}]$, must increase when the number of allocated pages grows beyond the capacity of the currently allocated physical

memory. The current implementation performs only static allocation of the maximum physical memory space, but future versions could easily realize the above memory savings. Chapter 5 discusses changes to GPU display drivers that will facilitate the implementation of this feature.

In comparison to the depth-culling-based sparse volume computation presented by Sherbondy et al. [47], the packing scheme presented herein guarantees that very few wasted fragments are generated by the rasterization stage. This is especially important for sparse computations on large volumes—where the rasterization and culling of unused fragments could consume a significant portion of the execution time. In addition, the packing strategy can process the entire active data set simultaneously, rather than slice-by-slice. This improves the computationally efficiency by taking advantage of the GPU's deep pipelines and parallel execution. The packing algorithm should also be able to process larger volumes, due to the memory savings discussed above. The packing algorithm, however, does incur overhead associated with maintaining the packed tiles, and more experimentation is necessary to understand the circumstances under which each approach is advantageous. Furthermore, they are not mutually exclusive, and Chapter 5 discusses the possibility of using depth culling in combination with the packed representation.

4.4 Tumor Segmentation User Study

4.4.1 Introduction

This section presents a evaluation study of the GPU-based level-set segmentation application [7, 27]. More than simply evaluating the GPU-based tool with respect to CPU-based applications, the study shows that the combination of interactivity, visualization, and level-set computation creates a tool that is more general and faster than previously existing options.

The purpose of the user study was to determine if the new level-set solver system can produce volumetric delineations of brain tumor boundaries comparable to those done by experts (e.g., radiologists or neurosurgeons) using traditional hand-contouring. The GPU-based segmentation application is applied to the problem of brain tumor segmentation using data from the *Brain Tumor Segmentation Database*, which is made available by the Harvard Medical School at the Brigham and Women's Hospital (HBW) [19, 54]. The HBW database consists of 10 three-dimensional 1.5T MRI brain tumor patient datasets selected by a neurosurgeon as a representative sampling of a larger clinical database. For

each of the 10 cases, there are also four independent expert hand segmentations of one randomly selected two-dimensional *slice* in the region of the tumor.

The user study consists of nine tumor cases: three meningioma (cases 1-3) and six low grade glioma (4-6, 8-10). One case, number 7, was omitted because a quick inspection showed it that its intensity structure was too complicated to be segmented by the proposed tool—such a problem remains as future work. The data used in the study was *not preprocessed*, and there are no hidden segmentation parameters—all system parameters were set by the users in real time, as they interacted with the data and the models.

Five users were selected from among University of Utah staff and students and trained briefly to use the software. Each user was asked to delineate the full, three-dimensional boundaries of the tumor in each of the nine selected cases. The users were given no time limit and their time to complete each tumor segmentation was recorded. None of the participating users were experts in reading radiological data. The goal of the study was not to test for tumor recognition (tissue classification), but rather to test whether parameters could be selected for the segmentation algorithm to produce a segmentation which mimics those done by the experts. To control for tumor recognition, we allowed each user to refer to a single slice from an expert segmentation. Users were told to treat this hand segmentation slice as a guide for understanding the difference between tumor and nontumor tissue. The underlying assumption is that an expert would not need such an example.

4.4.2 Methodology

The study considers three factors in evaluating the new segmentation tool [53]: validity of the results (accuracy), reproducibility of the results (precision), and efficiency of the method (time). To quantify accuracy a ground truth is established from the expert segmented slices using the STAPLE method [55]. This method is essentially a sophisticated averaging scheme that accounts for systematic biases in the behavior of experts in order to generate a fuzzy ground truth (W) for each case. The ground truth segmentation values for each case are represented as an image of values between zero and one that indicates the probability of each pixel being in the tumor. The STAPLE method also gives sensitivity and specificity parameters (p and q respectively) for each expert and each case. Sensitivity is the fraction of pixels correctly classified as lying inside the object boundary, and specificity is the fraction of pixels correctly classified as lying outside the

object boundary. Each subject generates a binary segmentation which, compared against the ground truth, gives values to obtain p and q for that subject. A third metric is also considered for the analysis, *total correct fraction* which is the total number of correctly classified pixels (weighted by W) as a percentage of the total size of the image.

To assess interoperator precision in segmentations, the study uses the metric proposed by [53], which consists of pairwise comparisons of the cardinality of the intersection of the positive classifications divided by the cardinality of the union of positive classifications. To analyze efficiency, the study calculates the average total time (user time plus processing time) taken for a segmentation.

4.4.3 Results

For a typical segmentation of a tumor using the new tool a user scrolls through slices until they find the location of the tumor. With a mouse, the user queries intensity values in the tumor and sets initial values for the parameters T and ϵ based on those intensity values. They initialize a sphere near or within the tumor and initiate deformation of that spherical model. As the model deforms the user scrolls through slices, observing its behavior and modifying parameters. Using the immediate feedback they get on the behavior of the model, they continue modifying parameters until the model boundaries appear to align with those of the tumor. In a typical 5-minute session, a user will modify the model parameters between 10 and 30 times.

Figures 4.9, 4.10, and 4.11 show graphs of average p, q, and c values for the experts and the users in the study. Error bars represent the standard deviations of the associated values for the experts and the users in the study.

The performance of the experts and the users varies case by case, but in almost all cases the performance of the users was within the range of performances of the experts. The average correct fraction of the users was better than the experts in 4 out of 9 cases. A general trend is that the participating users tended to underestimate the tumor relative to the experts, as indicated by lower values of p. This is consistent with other experiences with hand segmentations and level-set models—with hand contouring users tend to overestimate structures, and with level sets the curvature term tends to reduce the size of convex structures [8].

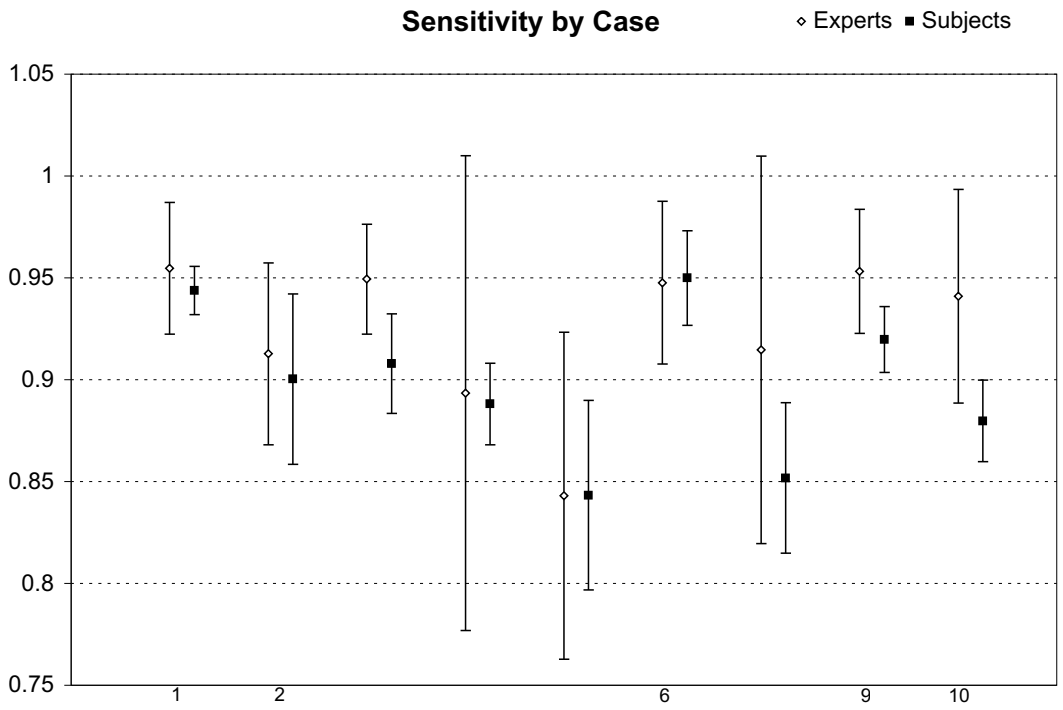


Figure 4.9. Sensitivity (the fraction of pixels correctly classified as inside the object boundary) results from the user study compare the interactive, GPU-based level-set segmentation tool with expert hand contouring. The results show that users of the semi-automatic tool produced segmentations that were within the error bounds of the expert hand contours in most cases. The tool also showed an overall slightly lower sensitivity, meaning that the size of the segmentations is slightly smaller.

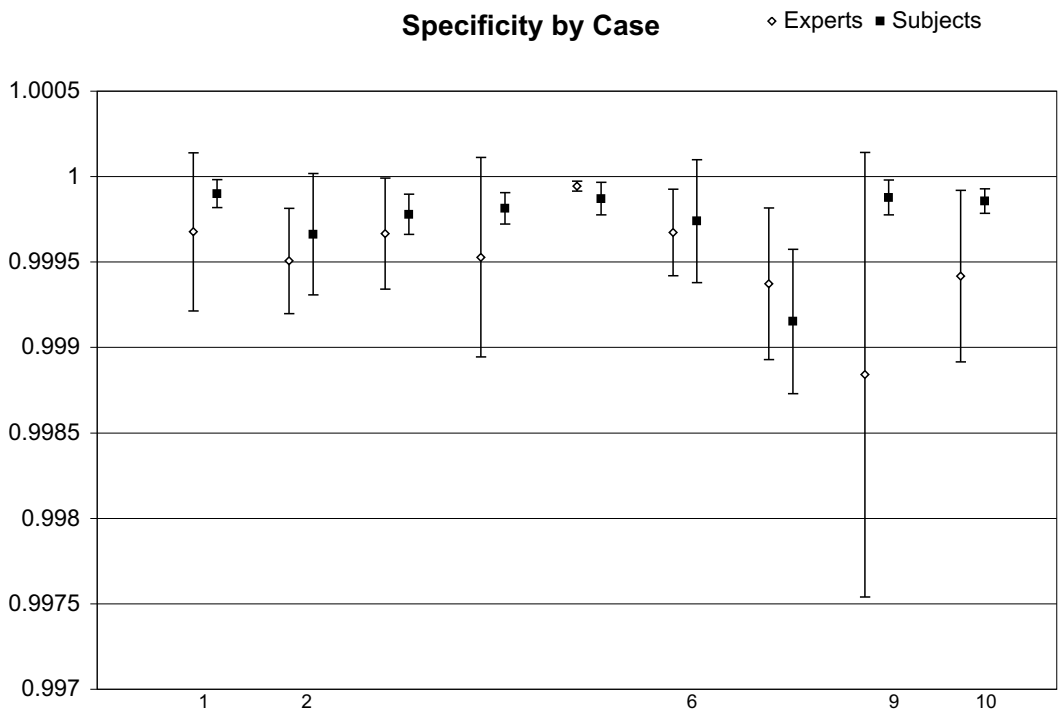


Figure 4.10. Specificity (the fraction of pixels correctly classified as outside the object boundary) results from the user study compare the interactive, GPU-based level-set segmentation tool with expert hand contouring. The results show that users of the semiautomatic tool produced segmentations that were within the error bounds of the expert hand contours in most cases. The tool also showed an overall slightly higher specificity, meaning that the size of the segmentations is slightly smaller.

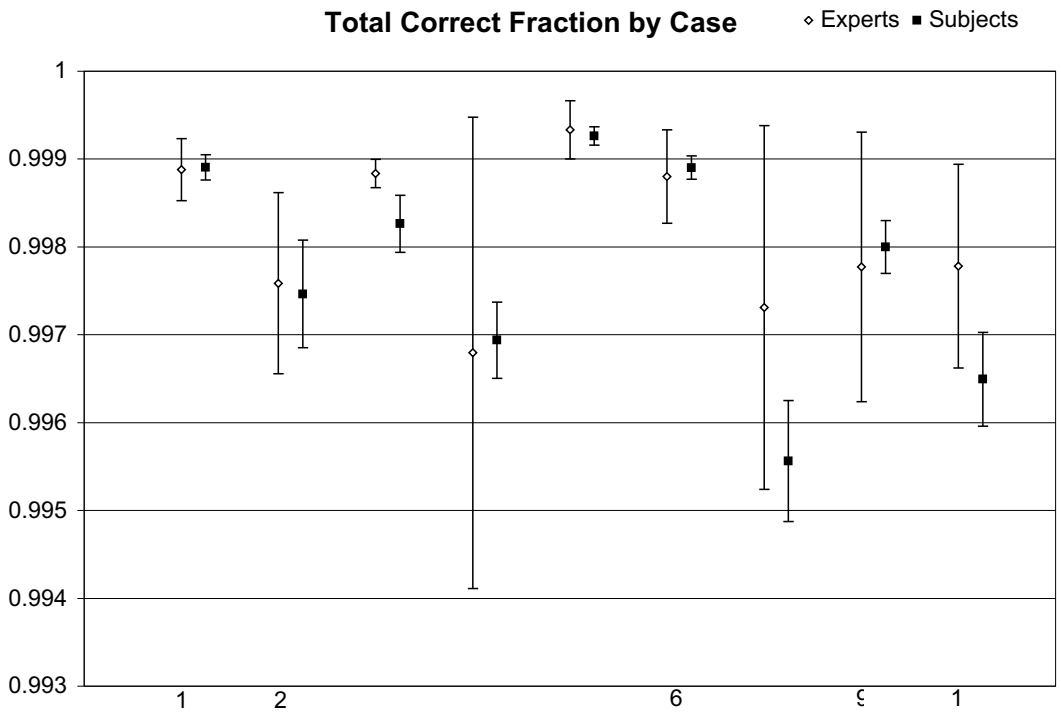


Figure 4.11. The total fraction of correctly classified pixels (combination of sensitivity and specificity) for the nine tumor cases segmented by the participating users.

The segmentations in the user study show a much higher degree of precision than the expert hand segmentations. Mean precision [53] across all users and cases was $94.04\% \pm 0.04\%$ while the mean precision across all experts and cases was $82.65\% \pm 0.07\%$. Regarding efficiency, the average time to complete a segmentation (all users, all cases) was 6 ± 3minutes. Only $5\% - 10\%$ of this time is spent processing the level-set surface. This compares favorably with the 3-5 hours required for a typical three-dimensional segmentation done by hand.

The accuracy and precision of subjects using the new tool compares well with the automated brain tumor segmentation results of Kaus, et al. [19], who use a superset of the same data used in the study. They report an average correct volume fraction of $99.68\% \pm 0.29\%$, while the average correct volume fraction obtained by the participating users was $99.78\% \pm 0.13\%$. Their method required similar average operator times (5-10 minutes), but unlike the proposed method their classification approach required subsequent processing times of approximately 75 minutes. That method, like many other segmentation methods discussed in the literature, includes a number of hidden parameters, which were not part of their analysis of timing or performance.

These quantitative comparisons with experts pertain to a only single two-dimensional slice that was extracted from the three-dimensional segmentations. This is a limitation due to the scarcity of expert data. Experience shows that computer-aided segmentation tools perform relatively better for three-dimensional segmentations because the hand contours typically show signs of interslice inconsistencies and fatigue. Figures 4.12 and 4.13 respectively show a segmentation by an expert with hand contouring and a segmentation done by one of the users of the GPU-based level-set segmentation tool.

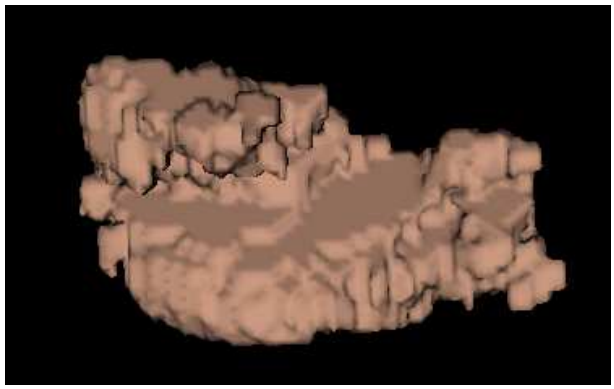


Figure 4.12. An expert hand segmentation of a tumor from the Harvard Brigham and Women's database shows significant interslice artifacts.

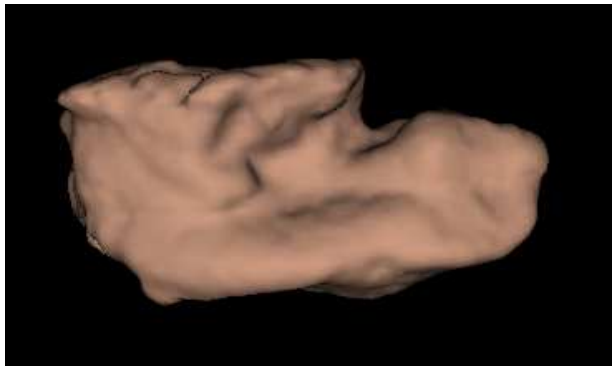


Figure 4.13. A three-dimensional segmentation of the same tumor from one of the subjects in the user study performed using the interactive segmentation tool described in this thesis.

CHAPTER 5

CONCLUSIONS

5.1 Summary

This thesis demonstrates a new tool for interactive volume exploration and analysis that combines the quantitative capabilities of deformable isosurfaces with the qualitative power of volume rendering. By efficiently leveraging programmable graphics hardware, the level-set solver operates approximately 15 times faster than previous solutions and is therefore interactive for moderately sized volumes (e.g., 128^3–256^3). This mapping relies on an efficient multidimensional virtual memory system to implement a time-dependent, sparse computation scheme. The memory mappings are updated via a novel GPU-to-CPU message passing algorithm. The GPU renders the level-set surface model directly from this packed texture format. This new rendering technique also enables full volume rendering from volume data stored as a *single* set of two-dimensional slices. The interactive segmentation tool is evaluated by means of a brain tumor segmentation user study. The study shows that, when compared to the segmentations produced by expert hand contouring, users of the new tool are able to quickly produce more precise and equivalently accurate segmentations.

5.2 Future Work

Future extensions and applications of the level-set solver include the processing of multivariate data as well as the application of the solver to other level-set problems. Examples include surface reconstruction, surface processing, and surface tracking in computational fluid dynamics simulations. Many of these extensions involve changing only the speed functions. Additions to the user interface, such as three-dimensional paint operations into the parameter volumes, may also enable an additional level of control over the computation. Additionally, the system described in this thesis enforces memory coherence at the granularity of a 16×16 memory page. This property might be interesting to apply to CPU-based sparse computations. The local memory access patterns, lack of

conditionals, and absence of pointer dereferences might result in a CPU-based solution that outperforms current sparse-field methods.

There are multiple improvements that could be made to the memory and computational efficiency of the solver. First, it may be worth achieving an even narrower band of computation around the level-set model. This is possible by using depth culling to avoid computation on inactive elements within each active page [47]. Implementing this depth culling requires a GPU memory model in which an arbitrary number of data buffers can access a single depth buffer. The second optimization is to allow the total amount of available physical memory to change at run time and grow to the limits of GPU memory. This requires spreading physical memory across multiple two-dimensional textures (i.e., creating a three-dimensional physical memory space). The proposed *super buffer* [37] OpenGL extension supports both of these proposed optimizations.

The GPU virtual memory abstraction also indicate promising future research. I am currently beginning work on a more general virtual memory implementation that fully abstracts N-dimensional GPU memory. The goal is to provide an API that allows a GPU application programmer to specify an optimal physical and virtual memory layout for their problem, then write the computational kernels irrespective of the physical layout. The kernels will specify memory accesses via abstract memory access interfaces, and an operating-system-like layer will replace these memory access calls with the appropriate address translation code. This layer should also optimize computational kernels by automatically mapping portions of the kernel to the vertex processor and rasterizer, generate substreams where appropriate, and perform other optimizations.

This thesis presents an effective solution for solving time-dependent narrow-band partial differential equations on the GPU. As the emerging field of general purpose computation on GPUs (GPGPU) moves forward, one of the most challenging questions is, "How general should the programming model become?" It is inevitable and desirable that the programming model for GPUs be lifted to a higher level. It is also critical, however, that the forthcoming abstractions not hinder effective use of the underlying hardware. For example, the fully general multidimensional virtual memory scheme proposed above may already be too general to guarantee efficient program execution. It is possible that the best high-level GPU programming solution will be domain-specific infrastructures (e.g., a framework for solving discrete partial differential equations). It is also possible, however, that a highly-optimizing programming language or framework for GPUs can be

defined by precisely defining the inherent restrictions required for efficient computation on streaming architectures.

APPENDIX A

DISCRETIZATION OF THE LEVEL-SET EQUATIONS

A.1 Introduction

This appendix describes the discretization of equation 2.3 and the curvature computation. Equation 2.3 is discretized using the *up-wind* scheme [34] and compute the curvature of the level-set surface using the *difference of normals* method [58].

A.2 Level-Set Discretization

To begin Equation A.1 describes the finite difference derivatives required for the level-set update and curvature computation. The neighborhood, u, from which these derivatives are computed is specified with the numbering scheme

6	7	8
3	4	5
0	1	2

. (A.1)

Note that 4 denotes the center pixel, and $u_i^{\pm z}$ represents the i^{th} sample on the slice above or below the current one. The derivatives of the level-set embedding, ϕ, are then defined as

$$
\begin{array}{lcllcl}
D_x & = & (u_5 - u_3)/2 & D_x^{+z} & = & (u_5^{+z} - u_3^{+z})/2 \\
D_y & = & (u_7 - u_1)/2 & D_x^{-z} & = & (u_5^{-z} - u_3^{-z})/2 \\
D_z & = & (u_4^{+z} - u_4^{-z})/2 & D_y^{+x} & = & (u_8 - u_2)/2 \\
D_x^{+} & = & u_5 - u_4 & D_y^{-x} & = & (u_6 - u_0)/2 \\
D_y^{+} & = & u_7 - u_4 & D_y^{+z} & = & (u_7^{+z} - u_1^{+z})/2 \\
D_z^{+} & = & u_4^{+z} - u_4 & D_y^{-z} & = & (u_7^{-z} - u_1^{-z})/2 \\
D_x^{-} & = & u_4 - u_3 & D_z^{+x} & = & (u_5^{+z} - u_5^{-z})/2 \\
D_y^{-} & = & u_4 - u_1 & D_z^{-x} & = & (u_3^{+z} - u_3^{-z})/2 \\
D_z^{-} & = & u_4 - u_4^{-z} & D_z^{+y} & = & (u_7^{+z} - u_7^{-z})/2 \\
D_x^{+y} & = & (u_8 - u_6)/2 & D_z^{-y} & = & (u_1^{+z} - u_1^{-z})/2. \\
D_x^{-y} & = & (u_2 - u_0)/2 &&&
\end{array}
\tag{A.2}
$$

Curvature is then computed using the above derivatives. The two normals, $\mathbf{n}^+$ and $\mathbf{n}^-$, are computed by

$$\mathbf{n}^{+} = \begin{bmatrix} \frac{D_x^+}{\sqrt{(D_x^+)^2+\left(\frac{D_y^{+x}+D_y}{2}\right)^2+\left(\frac{D_z^{+x}+D_z}{2}\right)^2}} \\ \frac{D_y^+}{\sqrt{(D_y^+)^2+\left(\frac{D_x^{+y}+D_x}{2}\right)^2+\left(\frac{D_z^{+y}+D_z}{2}\right)^2}} \\ \frac{D_z^+}{\sqrt{(D_z^+)^2+\left(\frac{D_x^{+z}+D_x}{2}\right)^2+\left(\frac{D_y^{+z}+D_y}{2}\right)^2}} \end{bmatrix} \tag{A.3}$$

and

$$\mathbf{n}^{-} = \begin{bmatrix} \frac{D_x^-}{\sqrt{(D_x^-)^2+\left(\frac{D_y^{-x}+D_y}{2}\right)^2+\left(\frac{D_z^{-x}+D_z}{2}\right)^2}} \\ \frac{D_y^-}{\sqrt{(D_y^-)^2+\left(\frac{D_x^{-y}+D_x}{2}\right)^2+\left(\frac{D_z^{-y}+D_z}{2}\right)^2}} \\ \frac{D_z^-}{\sqrt{(D_z^-)^2+\left(\frac{D_x^{-z}+D_x}{2}\right)^2+\left(\frac{D_y^{-z}+D_y}{2}\right)^2}} \end{bmatrix} \tag{A.4}$$

respectively. The components of the divergence from equation 2.4 are then computed as

$$\frac{\partial \mathbf{n}_x}{\partial x} = \mathbf{n}_x^+ - \mathbf{n}_x^-, \tag{A.5}$$

$$\frac{\partial \mathbf{n}_y}{\partial y} = \mathbf{n}_y^+ - \mathbf{n}_y^-, \tag{A.6}$$

and

$$\frac{\partial \mathbf{n}_z}{\partial z} = \mathbf{n}_z^+ - \mathbf{n}_z^-, \tag{A.7}$$

Finally, Equation A.8 estimates H:

$$H = \frac{1}{2}\left(\frac{\partial \mathbf{n}_x}{\partial x} + \frac{\partial \mathbf{n}_y}{\partial y} + \frac{\partial \mathbf{n}_z}{\partial z}\right). \tag{A.8}$$

The upwind approximation to $\nabla\phi$ is then computed using D_x^+, D_y^+, D_z^+, D_x^-, D_y^-, and D_z^-. To begin,

$$\nabla\phi_{\max} = \begin{bmatrix} \sqrt{max(D_x^+,0)^2 + max(-D_x^-,0)^2} \\ \sqrt{max(D_y^+,0)^2 + max(-D_y^-,0)^2} \\ \sqrt{max(D_z^+,0)^2 + max(-D_z^-,0)^2} \end{bmatrix} \tag{A.9}$$

is computed followed by

$$\nabla\phi_{\min} = \begin{bmatrix} \sqrt{min(D_x^+,0)^2 + min(-D_x^-,0)^2} \\ \sqrt{min(D_y^+,0)^2 + min(-D_y^-,0)^2} \\ \sqrt{min(D_z^+,0)^2 + min(-D_z^-,0)^2} \end{bmatrix}. \tag{A.10}$$

The final choice of $\nabla\phi$ is defined by

$$\nabla\phi = \begin{cases} \|\nabla\phi_{\max}\|_2 & \text{if } F > 0 \\ \|\nabla\phi_{\min}\|_2 & \text{otherwise} \end{cases}, \tag{A.11}$$

where F is the linear combination of all speed functions (e.g. mean curvature, the rescaling term G_r, etc). Section 4.2.1 describes the speed terms used in the level-set segmentation application.

The last step in the upwind scheme computes $\phi(t + \triangle t)$ by

$$\phi(t + \triangle t) = \phi(t) + \triangle t F|\nabla\phi|. \tag{A.12}$$

APPENDIX B

A BRUTE-FORCE, GPU-BASED THREE-DIMENSIONAL LEVEL-SET SOLVER

B.1 Design

This appendix covers the design of a brute-force, GPU-based three-dimensional level-set solver designed for the ATI Radeon 8500 GPU. The solver computes the level-set PDE (see Appendix A at each voxel, i.e., it is *not* a narrow-band solver. The solver also includes segmentation speed functions described in Section 4.2, including the second-order curvature term.

The level-set volume is stored in a set of two-dimensional slices (i.e., pbuffer textures). This memory arrangement is dictated by the fact that GPUs support only two-dimensional output buffers. As such, the PDE computation is performed on a slice-by-slice basis. Memory usage is slightly optimized by packing the scalar slices into the RGB channels of the RGBA pbuffers. In addition to saving texture memory, this also reduces the number of costly render target swaps by a factor of three.

B.1.1 Computation Overview

The three-dimensional solver requires seven render passes per slab to compute the mean curvature, and a total of 16 render passes per slab to compute an entire time step update. For a $256x256x175$ data set, this means that 2800 render passes are required to update the entire volume a single PDE time step. Pseudocode for the solver is shown below, using function-call-like syntax to represent render passes. The partitioning of the computation into render passes is dictated by the number of available texture inputs, temporary registers, and fragment program instructions.

```
for(int t=0; t < numSteps; t++) {
  for(int z=0; z < numSlabs; z++) {
```

```
    // Compute two sets of 4-vec derivatives
    Tex2D d1    = deriv1( phi[z] );                          // 1
    Tex2D d2    = deriv2( phi[mz], phi[pz] );                // 2

    // Compute Curvature
    Tex2D d3    = deriv3( phi[mz], phi[z], phi[pz] );        // 3
    Tex2D d4    = deriv4( phi[mz], phi[z], phi[pz] );        // 4
    Tex2D d5    = deriv5( phi[mz], phi[z], phi[pz] );        // 5
    Tex2D d6    = deriv6( phi[mz], phi[z], phi[pz] );        // 6
    Tex2D cx    = curvX( d1, d3, d4, normalizeLUT );         // 7
    Tex2D cxy   = curvY( d1, d3, d5, cx, normlalizeLUT );    // 8
    Tex2D curv  = curvZ( d1, d2, d6, cxy, normalizeLUT );    // 9

    // Sum the speed functions
    Tex2D speed = sumSpeed( curv, G );                       // 10

    // Upwind Computation
    Tex2D minG1 = minGrad1( d1, d2 );                        // 11
    Tex2D minG2 = minGrad2( minG1, d1, d2 );                 // 12
    Tex2D maxG  = maxGrad( d1, d2 );                         // 13
    Tex2D gMag1 = gradMag1( minG2, maxG, speed,              // 14
                            phi[z], l2NormLUT );
    Tex2D gMag2 = gradMag2( gMag1 );                         // 15

    // Do PDE timestep update
    Tex2D phi[z] = phiUp( gMag2, multScaleLUT );             // 16
  }
}
```

As discussed in Chapter 3, this full-volume solver is only one to two times faster than a sparse-field CPU-based implementation [52]. The GPU-based solver, however, is performing approximately 10 times more computations.

APPENDIX C

GPU MEMORY ALLOCATION REQUEST GENERATION

C.1 Introduction

This appendix describes the details of the GPU memory allocation/deallocation request scheme used by the GPU virtual memory system. The algorithm is described first in terms of an abstract client solver. Section C.1.2 presents the client-specific details in terms of the level-set solver client.

C.1.1 General Allocation Request Algorithm

The allocation request algorithm consists of the following steps:

A GPU computes VPN of requested active pages
B GPU compresses active-page request
C CPU reads compressed request image
D CPU decodes active-page request
  a Issues memory allocation/deallocation requests
  b Updates page tables and geometry engine
  c Calls client's `ReleasePage` function
  d Calls client's `InitNewPage` function

Steps A and B create the set of requested active virtual pages. This set serves as the memory allocation/deallocation request to the CPU. The CPU then calls the client's `ReleasePage` function for each newly deallocated page before deallocating the page. Similarly, the CPU calls the client's `InitNewPage` function for each newly activated page.

In Step A, the GPU uses client-specific data to create two auxiliary RGBA (i.e. 4-tuple) buffers that hold eight *true* or *false* (e.g., 255 or 0) values for each active data element (Figure 3.6). The first six values represent whether or not the virtual page in each of the six cardinal directions should be active for the next pass. The seventh value indicates if the active page itself should be active, and the eighth value is free to be used by the client. The level-set solver client uses the eighth value to determine if a newly

deactivated memory page is inside or outside the level-set surface. This eight-dimensional, active-page information vector, J, is thus J = (+x, −x, +y, −y, +z, −z, self, clientSpecific), where the first six elements refer to relative neighbor offsets in the virtual page space, V_P.

The eight-value code, J, is computed in eight substream passes followed by a single standard (i.e., entire memory page) pass. The substream passes compute whether the in-plane adjacent memory pages needs to be active (i.e., the edge-adjacent pages $(+x, -x, +y, -y)$). Each substream pass computes the value of a client-specified function, `IsNeighborActive`, across the page boundary orthogonal to the page edge being rendered and writes the boolean result to the corresponding output component of J. The second computation calls `IsNeighborActive` for the pages above and below the active one. Note, however, that because the neighboring pages are *face-adjacent*, this computation is performed at all data elements in the page instead of just the edges. The computation also writes a *true* value to the J component representing the active page itself if the client's `IsSelfActive` function returns true. The value of the eighth bit is filled by the result of the client's `IsEighthBitTrue` function.

Step B of the allocation-request algorithm is to compress the two, J buffers into a small ($\leq$ 64kB) active-page message. This compressed message serves as the memory allocation/deallocation request that is sent to the CPU. The compression is accomplished by rendering a quadrilateral of size $S[G_P]$ with the *automatic mipmapping* option enabled on the neighbor-information buffers. The render pass also uses a fragment program designed to create a bit code at each pixel value. Each pixel in the resulting small image corresponds to a physical memory page. The value of each pixel contains an eight-bit code of the same form as the eight-value code produced in step A (i.e., the J vector). This eight-bit code completely determines if the memory page and/or any of its six cardinal neighbors in virtual page space are to be active on the next pass.

The automatic mipmapping performs a box-filter averaging of the values written in Step A. The result is that if *any* data element in the memory page set a value to *true* (i.e., 255) in Step A, the down-sampled value will also be true (i.e., nonzero). The fragment program inspects these down-sampled values. It sets the corresponding bit in the output value to true for each nonzero input. The bits are set via an emulated bitwise OR operation. Current fragment processors do not support bitwise operations, but an OR is emulated by conditionally adding power-of-two values to the output value.

In Step C, the CPU reads the bit-code message from the GPU. Step D begins by the CPU wrapping the message buffer with a bit-vector accessor. The resulting bit vector is a linear representation of the physical page space, G_P, where each byte represents the information for a page. Two auxiliary bit-vectors are allocated—each a bit-addressed, linear representation of the virtual memory page space, V_P. The first is the `newActiveSet` bit vector, and the second is the client-specific `eighthBitSet` bit vector. After the allocation message is decoded, a true bit in the `newActiveSet` bit vector will denote an active virtual page.

The CPU then decodes the bit-vector message. For each 8-bit sequence, the current linear index is converted to a physical page number (PPN). The inverse page table then converts the PPN to a VPN. Because each bit in the bit-code message represents an offset direction from the current virtual page, the decoder can easily reconstruct the VPN for each neighbor of each active page. The decoder then reads the seven spatial page bits. It then computes the VPN for the page represented by each true bit and sets the corresponding bit in the `newActiveSet` bit vector to true. If the eighth bit is true, the `eighthBitSet` is set to true for the corresponding virtual page.

The virtual memory system next determines which virtual memory pages to deallocate and which to allocate. The set of newly deactivated pages is constructed by performing a set-subtraction of the `newActiveSet` from the `oldActiveSet`. The set of pages that need to be allocated for the next pass is created by computing the opposite set difference. Each deallocated memory page is pushed onto a stack of free memory pages. The page table are updated based on the client's implementation of `ReleasePage` function. Each newly activated page is mapped to a physical memory location by popping a page from the free page stack. The physical page is mapped in the page tables and the geometry engine is appropriately updated. The new physical memory is then initialized via the client's `InitNewPage` implementation.

C.1.2 Level-Set Solver Implementation Details

For Step A of the update algorithm described in section C.1.1, the level-set solver defines the functions `IsNeighborActive` and `IsSelfActive`. The `IsNeighborActive` reads the previously computed, one-side derivative that crosses a page boundary onto a specific neighbor. The function returns true if the derivative is nonzero. The `IsSelfActive` function returns true if *any* of the six, cardinal, one-sided derivatives are nonzero. The

level-set solver simply writes the value of the level-set embedding to the eighth data value. This is used to determine if a newly deactivated page is inside or outside of the level-set surface. The `IsEigthBitTrue` function used by the fragment program in Step B returns true if the eighth data value is greater than zero. If a page becomes inactive, it is guaranteed to be either all black or all white. The down-sampled level-set embedding for the page will thus be either pure black or pure white.

The `eighthBitSet` used in the bit-code message decoding stage (Step D) is used to determine if a newly deactivated memory page is inside or outside the level-set surface. If the bit for the page is true, then the page is inside the surface. Otherwise it is outside. This information is used by the solver's `ReleasePage` function to map deactivated pages to the correct *static* physical page (white or black). These static mappings ensure that derivatives across boundaries of the active domain are correct.

The solver's `InitNewPage` function initializes newly allocated physical memory. The memory is initialized to either white or black depending on the inside/outside setting in the page table entry. Note that no level-set data are transferred to accomplish the update. The entire level-set solution resides *only* on the GPU for the duration of the computation. The current implementation also has to send pre-computed speed pages to the GPU when new pages are added. This could be optimized for many speed functions, however, by computing the function on the GPU.

APPENDIX D

SOFTWARE DESIGN

D.1 Introduction

This appendix describes the software infrastructure on which the GPU-based level-set solver and visualization system is built. The first section gives an overview of the layered structure of the code. The proceeding sections give detailed documentation for each of these layers.

D.2 Design Overview

The GPU-based level-set solver and visualization system are built using five separate software layers. Figure D.1 shows these five layers and the libraries within each layer. The lowest layer includes low-level data structures and the OpenGL interface for controlling the GPU. The second layer provides object-oriented abstractions for GPU-specific operations, while the third layer encapsulates the operations necessary to execute an entire render pass. The fourth layer includes the level-set solver code and visualization modules described in Chapter 3. Lastly the fifth layer encompasses the volume segmentation application described in Chapter 4.

The first layer is comprised of three software libraries: The OpenGL three-dimensional graphics API [43], a utility library called *Gutz*, and an OpenGL management utility called *Glew* [17]. The OpenGL routines issue commands to the GPU and pass data between the CPU and GPU. The Gutz library contains core utilities such as vectors, arrays, and matrices. Milan Ikits' Glew library greatly simplifies the handling of the various OpenGL versions and extensions.

The second layer is an object-oriented abstraction around OpenGL called *Glift*. Glift uses OpenGL and Gutz to provide a framework for writing modular, re-usable OpenGL code.

The third layer, CompGPU, uses Glift objects to abstract a GPU render pass as a function-object (functor). CompGPU enables programmers to write render passes

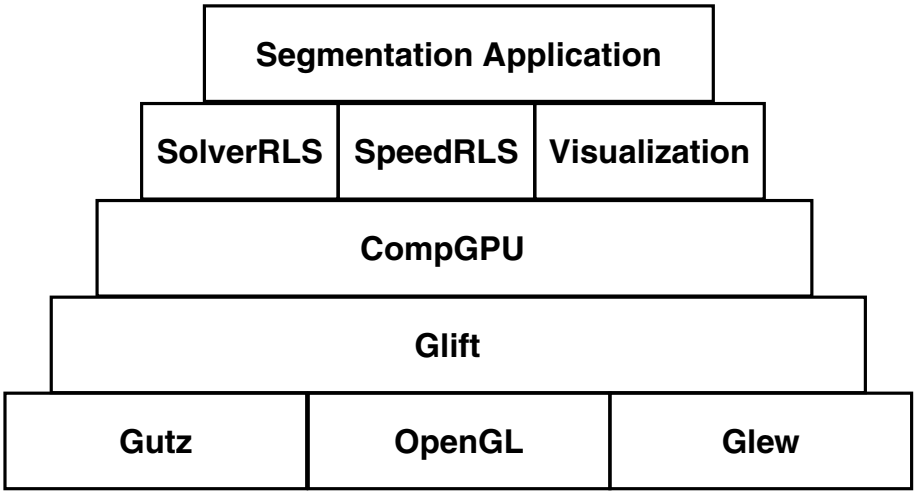


Figure D.1. The five software layers with which the level-set segmentation application is built. In the first layer, OpenGL is used to control the GPU, Gutz defines vector, matrix, and array data structures, and Glew handles OpenGL extensions. The second layer, Glift, combines OpenGL calls into reusable object-oriented OpenGL modules. CompGPU is the third layer and encapsulates an entire render pass as a `forEach` function call. The level-set solver, level-set speed functions and visualization modules are defined in the fourth layer, and the volume segmentation application comprises the fifth layer.

using function-call-like syntax. The fourth software layer contains the level-set solver and visualization modules. Both of these libraries make extensive use of Gutz, Glift, and CompGPU objects. The solver and visualization objects communicate with each other via several predefined interfaces.

The fifth and final layer is the interactive level-set segmentation application. This layer creates an instance of the level-set solver and configures it for volume segmentation. The segmentation application also instantiates the visualization modules. The application specifies the graphical user interface (GUI) using Glut [20] and Glui [39]. Note that the fourth and fifth layers make extensive use of Gordon Kindlmann's Nrrd library [21] for raster data manipulation. This includes file I/O, data resampling, resizing, slicing, cropping, tiling, etc.

D.3 The First Layer

D.3.1 OpenGL

The system uses the OpenGL graphics API to control the graphics processor. The details of the API are described in other sources [43] and will not be repeated here. OpenGL calls set the state of the graphics board and display drivers. This low-level of programming is error-prone and leads to non-reusable code. This is the motivation for the Glift abstraction layer.

D.3.2 Glew

The Glew library [17] (OpenGL Extension Wrangler) greatly simplifies the many versions of OpenGL and the large number of OpenGL extensions. It also unifies the use of all OpenGL features across multiple computational platforms. Glew was created and is maintained by Milan Ikits and will not be discussed in detail in this thesis. In brief summary, using Glew entails simply replacing all OpenGL-related header includes (including vendor-specific extensions) with `glew.h` and `wglew.h`. The function `glewInit()` is then called once in the application to initialize all OpenGL API calls (including *all* extensions). Glew can then be queried at run-time to determine the availability of specific OpenGL features.

D.3.3 Gutz

The Gutz library contains ubiquitous primitives for graphics-related programming such as vectors, matrices, and arrays. This library is a combination of code written by

Joe Kniss, Milan Ikits, and myself. All of the Gutz classes are template-based, lightweight abstractions on top of raw data. The classes are carefully designed to have a run-time representation that consists *only* of the desired data (i.e., no virtual function pointers, etc.). The result is that complex data structures can be created (e.g., multidimensional arrays of vectors) that have a C-like, contiguous underlying memory representation. This representation is critical for both performance and interfacing with low-level API's such as OpenGL that require contiguously allocated data.

The vector classes consist of templated classes for 1D to 4D vectors, while the matrices consist of templated classes for 2×2 to 4×4 matrices. Pre-defined typedefs exist for many of the common instantiations of these objects. The typdefs are named in a similar fashion to OpenGL type specifications. For example, a three-vector of floats is a `vec3f`, a 4×4 matrix of integers is a `mat4i`.

The array classes are templated by element type and separate classes exist for 1D to 5D arrays. There are two types of arrays: `arrayOwn` and `arrayWrap`. These differ by memory ownership policy. Creating an `arrayOwn` object allocates memory for the array. Likewise, deleting an `arrayOwn` frees the memory. In contrast, an `arrayWrap` object does not allocate or free the underlying memory. The purpose of the `arrayWrap` classes is to provide convenient multidimensional accessors around raw data. They also allow the programmer to "cast" array data to different dimensionalities. This design is again motivated by the requirement to communicate blocks of data to/from low-level APIs. Note that the `arrayWrap` class for each dimension of array is a subclass of the corresponding `arrayOwn`. As such, `arrayWrap` objects can be passed as function arguments where the parameter specification is an `arrayOwn`. Also note that an `arrayBase` class exists that is dimension-agnostic and can thus be used to pass arbitrary dimensioned arrays.

D.4 The Second Layer: Glift

The second layer is an object-oriented abstraction around OpenGL called *Glift*. Glift uses OpenGL, Gutz, and Glew to provide a framework for writing modular, re-usable OpenGL code. Unlike other object-oriented OpenGL encapsulations such as GLT[49] and OpenInventor[45], the Glift framework is designed for low-level OpenGL developers rather than high-level graphics programmers. Glift's object structure is designed only to enforce semantically correct OpenGL programming but avoid making assumptions about how OpenGL will be used. Glift also does not encapsulate any windowing-related calls

other than the handling of pbuffers. It is expected that a windowing utility such as Glut [20] will be used.

The Glift framework defines a set of reusable and extensible modules that can be composited into higher-level objects. The multi-level approach is very flexible in that a programmer can choose to work at various levels within the same application. The possible coding levels include raw OpenGL, basic Glift objects, and various levels of composited Glift objects. These composite Glift objects may be as simple as a multi-texture object or as complex as an entire render pass. Another goal of the library is to isolate all GPU-vendor-specific OpenGL code into pluggable modules to facilitate writing applications that support multiple GPU architectures. A class tree of Glift is shown in figure D.2.

The Glift design supports two types of OpenGL calls: those that set/unset GPU pipeline state (the `StateGLI` tree) and those that initiate processing of data through the pipeline (the `DrawableGLI` and `RenderableGLI` trees). A third type of call, pipeline status queries, are not currently supported but could be added later. All OpenGL calls that set/unset state are encapsulated by the class tree based on the `StateGLI` interface. This interface specifies a `bind()` and `release()` public virtual method. OpenGL calls that move data through the pipeline are encapsulated by the class tree based on the `DrawableGLI` interface. `DrawableGLI` simply specifies a public `draw()` method. A third class tree based on the `RenderableGLI` interface combines all the `StateGLI` and `DrawableGLI` objects that specify an entire render pass.

In addition, all Glift objects support a `compile()` method that attempts to compile the OpenGL commands encapsulated by the object into a display list. Note that this feature provides a way to "compile away" the abstraction penalty that might otherwise be caused by the extensive use of virtual functions. In practice, however, the GPU consumes most of the execution time in many Glift applications and so the abstraction layers do not affect the execution speed. The Glift `compile()` feature is currently only partially implemented. It is supported throughout the framework, but only works correctly when *all* of the reachable OpenGL calls can legally be compiled into display lists. Future work will add the correct handling of OpenGL calls that cannot be compiled (e.g., wgl calls, vertex array pointer calls, etc.).

Glift is designed to provide a minimal amount of preencapsulated OpenGL state and have obvious extension points for adding more functionality as desired. As Glift matures,

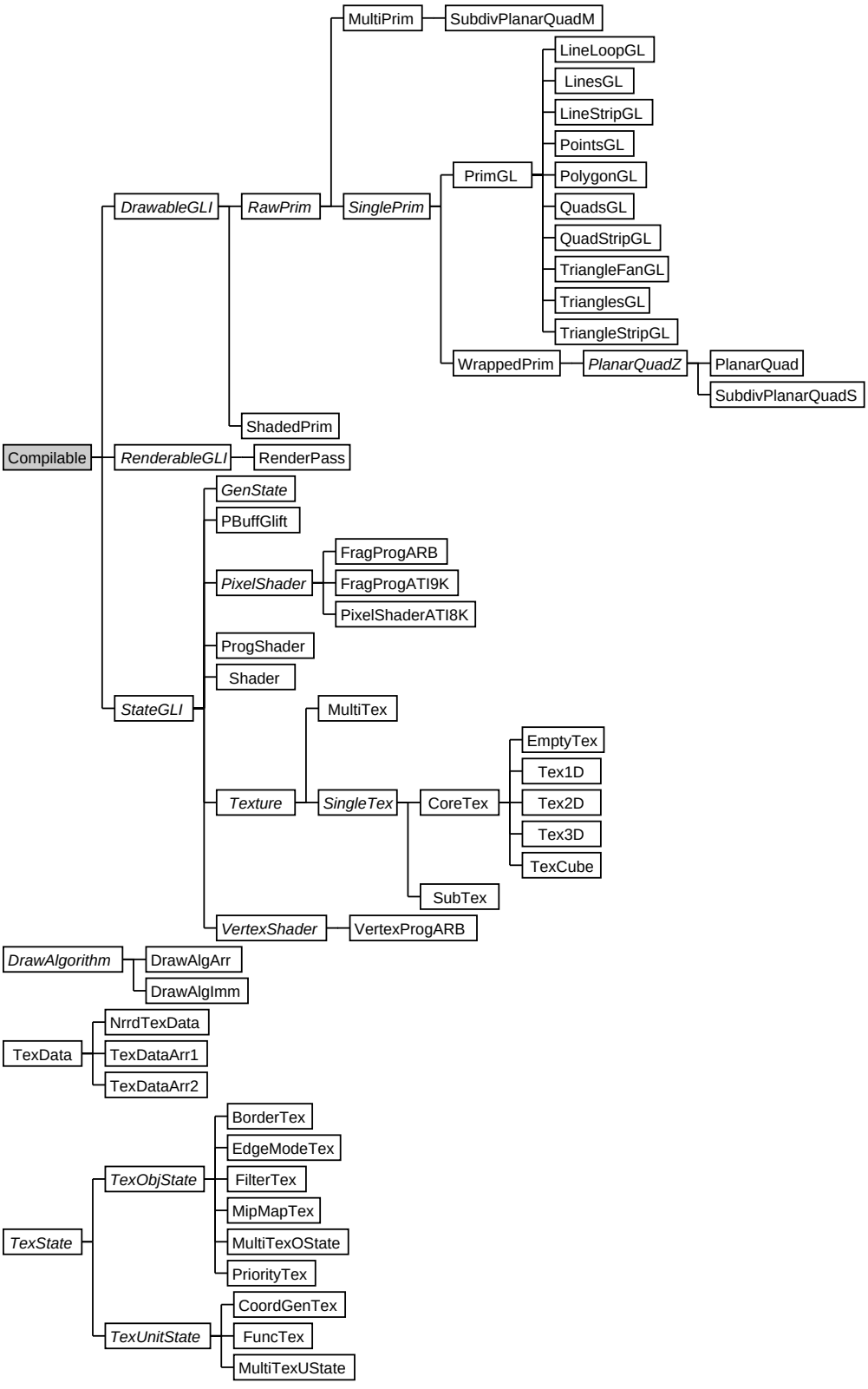


Figure D.2. Class tree for the Glift, object-oriented OpenGL framework.

more functionality will come predefined by the library. The following is a list of current extension points:

Class Name	Purpose
`GenState`	Defining any bind/release state that is not already defined.
`PixelShader`	Defining interfaces to hardware-specific fragment shaders
`VertexShader`	Defining interfaces to hardware-specific vertex shaders
`WrappedPrim`	Defining high-level drawables containing a single PrimGL object
`MultiPrim`	Defining high-level drawables containing multiple PrimGL objects
`RenderPass`	Defining a render pass with functionality different than has been provided
`DrawAlgorithm`	Defining a drawing algorithm other than the standard (glBegin(...)/glEnd(...) or vertex array method
`TexCoordGen`	Defining texture coordinate generation algorithms

To begin compositing a render pass, the `StateGLI` objects are first composited into a `Shader` object. The `Shader` thus contains the specification of the textures and any other OpenGL pipeline state required by the pass. The `DrawableGLI` objects are then defined and put into a `MultiPrim` object. The `Shader` object and the `Multiprim` (or any other `RawPrim`) are combined into a `ShadedPrim` object. This `ShadedPrim` object (or any `Drawable`) is combined optionally with a texture and/or pbuffer destination into a `RenderPass`.

The use of the texture objects (the `Texture` class tree) require some additional explanation. To create a texture object, the user first creates an instance of `MultiTexOState` to specify the texture object state. If texture data is to be downloaded to the texture object, the user also creates an appropriate `TexData` object. The constructor of the desired texture object then takes a pointer to the `MultiTexOState` object and optionally the `TexData` object. In addition, the texture constructors also accept an optional pointer to a pbuffer object (`PbuffGlift`).

Glift is by no means a completed project. The first issue is to complete the `compile()` implementation to support non-compilable API calls. The second future project is to add a `PrimWrap` object to the `DrawableGLI` tree that does not own its vertex and attribute data. It instead should only hold pointers to the application-owned data. This is important for adoption of Glift into existing applications. It may be possible to implement this by adding C++ template *policies* [2] to the current objects. The third future-work issue is adding smart-pointers (i.e., reference counting pointers) throughout

Glift for automatic memory management. The last, and most ambitious, future direction for Glift is to rework the core objects such that Glift can interchangeably use Microsoft's DirectX or OpenGL as an underlying API. An important goal for Glift design is that the framework have a clearly defined layer that remains *below* that of a scene graph (i.e., a scene graph could be built using Glift objects). It may be advantageous to remove the Glift objects that are above this abstraction level in order to facilitate its adoption.

In addition to its use in the level-set solver described in this thesis, Glift is now being used by Joe Kniss in his Simian volume renderer [22]. An early version of Glift was also used to build the front-end of the real-time ray tracing demo, Star-Ray, shown at the SGI Siggraph 2002 exhibition booth [46].

D.5 The Third Layer: CompGPU

The third software layer, *CompGPU*, uses Glift objects to abstract a GPU render pass as a function-object (functor). The function-call abstracted by CompGPU is essentially a `forEach` loop over data stored in texture memory. The specification of which data elements to include in the computation is specified by the rasterization of two-dimensional geometry. The computation performed on each element is specified by the vertex and fragment programs, and the results of the `forEach` call are written to the specified output buffer(s). While the level-set solver was successfully built using CompGPU objects, the design has proven to be cumbersome and problematic. This software layer should be re-designed before future projects adopt it. As such, this section describes both the successes and failures of the design.

The CompGPU layer consists of only a single class, `ComputeSlab`. All clients of CompGPU subclass `ComputeSlab` to create a specific render pass. The computation (i.e., render pass) is initiated by calling the `compute()` virtual function with the appropriate parameters.

The design decision to use `ComputeSlab` as a base class is a severe problem with CompGPU. Although the design does maximize code reuse, the fact that each computation must be a separate class definition leads to an explosion in code size. A policy-based CompGPU layer appears to be a much better solution. Mark Harris's SlabOp class [15] is a much better starting point than CompGPU. SlabOp, however, is missing a function-call-like syntax. This last features is difficult to support in a general fashion, yet is an important abstraction for writing general-purpose GPU computation applications.

One of the largest challenges is the specification of arguments to the `compute()` function. In practice, many of the input arguments (textures, fragment programs, geometry, etc.) remain constant each time a computation is performed. There are instances, however, when some of these arguments do change. An argument caching mechanism is thus needed so that the programmer can dynamically select which parameters need to be updated.

The current mechanism for handling arguments is to pass static arguments via the subclass constructor, and pass dynamic arguments to the `compute()` function. This is an effective solution, but leads to a large number of `compute()` versions in the base class. A policy-based implementation may be the solution to this—where the interface of the `compute()` call is defined by a policy. Note that Mark Harris's SlabOp handles this problem by requiring the programmer to set the state of the SlabOp object before calling the analogue of `compute()` with no arguments. The problem with this approach is that it leads to difficult-to-read code that does not have the appearance of function (`forEach`) calls. Implementing a C++ function call on top of each SlabOp call may be a reasonable solution.

D.6 The Fourth Layer

The fourth software layer contains the level-set solver and visualization modules. Both of these libraries make extensive use of Gutz, Glift, and CompGPU objects. The solver and visualization objects communicate with each other via several predefined interfaces.

D.6.1 Level-Set Solver

The GPU-based level-set solver uses CompGPU, Glift, and Gutz objects to build a flexible solver framework. This framework includes the full specification of various level-set solvers and the speed function modules used by the solvers. The current design also includes two-dimensional visualization tools, but these should be removed—just as the volume rendering module is entirely separate from the solver. Although the design includes level-set-specific functionality, the framework lays the groundwork for a more general solver infrastructure in the future. Much of the infrastructure described herein could and should be handled by a compiler. The design of this framework, however, does outline a set of required features for future streaming languages/APIs for general purpose GPU computation.

The core solver class is `SolverRLS`. The specific solvers are subclasses of `SolverRLS`. The various solvers include versions for different GPU architectures, separate two-dimensional and three-dimensional versions, and sparse and dense computation versions. In addition, a parallel class tree, `SpeedRLS`, specifies the speed function modules. The solvers take `SpeedRLS` objects as constructor inputs.

The `SolverRLS` class tree is an example of the Strategy object-oriented design pattern. The base class provides the functionality that is common to all solvers and specifies abstract interfaces for functionality that is required, but specific to the specific solvers (implemented as subclasses). `SolverRLS` provides memory management services, user interface (UI) hooks, as well as speed function management.

The solver subclasses own their specific computation. They create the CompGPU objects for each pass, specify the order of the passes, and integrate the speed function modules into the computation. Each subclass reports the number of live temporary buffers at each program point (where each program point specifies a render pass) to the base class. The base class then uses this information to allocate an appropriate number of temporary pbuffers/textures and perform register allocation to resolve conflicts between the buffers. The subclass then receives a set of pbuffer/texture pointers to use for each program point that minimizes memory usage and guarantees that no data conflicts will occur.

The solver infrastructure is designed to allow for fully modular speed functions that can be arbitrarily added to appropriate solvers without having to change the solver. The solvers interact with the speed functions by informing the base class of `DataPack`s that are available and for which program points these `DataPack`s are valid. A `DataPack` is a set of level-set-specific temporary values that are currently held in texture memory. The elements of a `DataPack` are called `SolverSID`s (solver service IDs). These `DataPack`s are used by the base class to schedule the execution of speed function modules into the computation.

The speed function modules encapsulate an entire level-set speed function computation. They are implemented as subclasses of `SpeedRLS`. The computation of the speed function may include zero to many render passes. Just like the solver modules, the speed function module owns its own CompGPU objects and specifies the order of the computation. The speed functions receive input data from the solver by subscribing to `DataPack`s provided by the solver. Speed function modules also report the number of

temporary buffers required for the speed computation. The `SolverRLS` class analyzes the `DataPack` and temporary buffer requests to schedule the execution of the speed module and allocate pbuffers and textures appropriately. As mentioned above, much of this functionality would be much more concisely expressed in language form—either specifically for level-sets or for general streaming GPU computation. The Brook [5] streaming language is a start in this direction.

The naming scheme for the solvers requires some additional explanation. The names of the subclasses all begin with `SolverRLS_`. The next three characters describe attributes of the solver. The first character is either D or S and denotes if the solver users a dense (i.e., full) or sparse memory representation, respectively. The second character is also either D or S and denotes if the solver uses dense or sparse computation, irrespective of the memory representation. The last character represents the dimensionality of the solver and is thus either 2 or 3. The last part of the solver name denotes the GPU architecture for which the solver is designed. Currently the two options are A8 and A9, which stand for ATI Radeon 8500 and ATI Radeon 9x00 GPUs, respectively. Note that the A9 classification is for GPUs with a model number of 9600 or higher. As such, the streaming narrow-band solver described throughout the thesis is named `SolverRLS_SS3_A9`.

D.6.2 Visualization Modules

The three-dimensional volume rendering module described in this thesis is an entirely separate library. The majority of this code was written by Joe Kniss and leverages his *Simian* [22] volume rendering library. The level-set-solver-specific code is in a module called `lsetRen`. This is not implemented as a class, but should be. It specifies an initialization call (essentially a constructor), a function to set the level-set input data texture, and a function used to update the virtual-to-physical page table mapping. This update function is called by the `SolverRLS_SS3_A9` module when a visualization update is requested. The module also specifies hooks for the three-dimensional user-interface features (three-dimensional manipulations, clipping plane operations, etc.).

The two-dimensional visualization modules are currently owned by the solver modules. They should, however, be entirely separate and communicate with the solvers similar to the communication scheme used by the volume renderer.

D.7 The Fifth Layer: Level-Set Segmentation Application

The fifth and final layer is the interactive level-set segmentation application. This layer creates an instance of the level-set solver and configures it for volume segmentation by creating appropriate speed functions. The segmentation application also instantiates the volume visualization module. This software level has received the least amount of development time and effort of the entire application and should be viewed as a minimal implementation with much room for improvement.

The application specifies the graphical user interface (GUI) using *Glut* [20] and *Glui* [39]. All file i/o and raster-data manipulation is performed using Gordon Kindlmann's *nrrd* library [21]. The `main` routine for the application is contained in the file, "base.cpp."

REFERENCES

[1] Adalsteinson, D., and Sethian, J. A. A fast level set method for propagating interfaces. *Journal of Computational Physics* (1995), 269–277.

[2] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[3] Beers, A. C., Agrawala, M., and Chaddha, N. Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM Press, pp. 373–378.

[4] Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *ACM Transactions on Graphics* (July 2003), vol. 22, pp. 917–924. (Proceedings of ACM Siggraph 2003).

[5] Buck, I., Foley, T., Horn, D., Sugerman, J., Hanrahan, P., Houston, M., and Fatahalian, K. BrookGPU. `http://graphics.stanford.edu/projects/brookgpu/`, 2004.

[6] Cabral, B., Cam, N., and Foran, J. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *ACM Symposium On Volume Visualization* (1994).

[7] Cates, J., Lefohn, A. E., and Whitaker, R. GIST: An interactive, GPU-based level-set segmentation tool for 3D medical images. Under review at Medical Image Analysis.

[8] Cates, J., Whitaker, R., and Jones, G. Case study: An evaluation of user-assisted hierarchical watershed segmentation. Under review at Medical Image Analysis, 2004.

[9] Cebenoyan, C., and Wloka, M. Optimizing the graphics pipeline. Game Developer's Conference 2003, `http://developer.nvidia.com/`, 2003.

[10] Drebin, R. A., Carpenter, L., and Hanrahan, P. Volume rendering. In *ACM Computer Graphics (SIGGRAPH '88 Proceedings)* (August 1988), pp. 65–74.

[11] Droske, M., Meyer, B., Rumpf, M., and Schaller, C. An adaptive level set method for medical image segmentation. In *Proc. of the Annual Symposium on Information Processing in Medical Imaging* (2001), R. Leahy and M. Insana, Eds., Springer, Lecture Notes Computer Science.

[12] Fedkiw, R., Aslam, T., Merriman, B., and Osher, S. A non-oscillatory Eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *Journal of Computational Physics 152* (1999), 457–492.

[13] Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G. A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003* (July 2003), pp. 102–111.

[14] Grayson, M. A short note on the evolution of surfaces via mean curvatures. *Journal of Differential Geometry 58* (1989), 555.

[15] Harris, M., and Lefohn, A. E. Slabop research noteblog. `http://www.cs.unc.edu/~harrism/noteblog/archive/2003_01_26_archive.html%`, 2003.

[16] Hillis, W. D. *The Connection Machine*. MIT Press, 1985.

[17] Ikits, M. Glew, the OpenGL Extension Wrangler. `http://glew.sourceforge.net`, 2002.

[18] Kapasi, U., Dally, W. J., Rixner, S., P. R. Mattson, J. D. O., and Khailany, B. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture* (2000), pp. 159–170.

[19] Kaus, M., Warfield, S. K., Nabavi, A., Black, P. M., Jolesz, F. A., and Kikinis, R. Automated segmentation of MRI of brain tumors. *Radiology 218* (2001), 586–591.

[20] Kilgard, M. Glut, the OpenGL utility toolkit. `http://www.opengl.org/developers/documentation/glut/index.html`, 1997.

[21] Kindlmann, G. Teem. `http://teem.sourceforge.net`, 2003.

[22] Kniss, J., Kindlmann, G., and Hansen, C. Multi-dimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics 8*, 3 (July-September 2002), 270–285.

[23] Kniss, J., Premoze, S., Hansen, C., Shirley, P., and McPherson, A. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics 9* (April-June 2003), 150–162.

[24] Kraus, M., and Ertl, T. Adaptive texture maps. In *Graphics Hardware 2002* (Sept. 2002), pp. 7–16.

[25] Krüger, J., and Westermann, R. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Transactions on Graphics* (July 2003), vol. 22, pp. 908–916. (Proceedings of ACM SIGGRAPH 2003).

[26] Larsen, E. S., and McAllister, D. Fast matrix multiplies using graphics hardware. In *Super Computing 2001* (Nov. 2001), ACM SIGARCH/IEEE.

[27] Lefohn, A. E., Cates, J., and Whitaker, R. Interactive, GPU-based level sets for 3D segmentation. In *Medical Image Computing and Computer Assisted Intervention* (2003), pp. 564–572.

[28] Lefohn, A. E., Kniss, J., Hansen, C., and Whitaker, R. Interactive deformation and visualization of level set surfaces using graphics hardware. In *IEEE Visualization* (October 2003), pp. 497–504.

[29] Lefohn, A. E., Kniss, J., Hansen, C., and Whitaker, R. A streaming narrow-band algorithm: Interactive deformation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics* (2004), To Appear.

[30] Lefohn, A. E., and Whitaker, R. A GPU-based, three-dimensional level set solver with curvature flow. University of Utah technical report UUCS-02-017, December 2002.

[31] Levoy, M. Display of surfaces from volume data. *IEEE Computer Graphics & Applications 8*, 5 (1988), 29–37.

[32] Malladi, R., Sethian, J. A., and Vemuri, B. C. Shape modeling with front propagation: A level set approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence 17*, 2 (1995), 158–175.

[33] Microsoft Corporation. Direct3D. `http://www.microsoft.com/directx`, 2002.

[34] Osher, S., and Sethian, J. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics 79* (1988), 12–49.

[35] Owens, J. D. *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, Nov. 2002.

[36] Peng, D., Merriman, B., Osher, S., Zhao, H., and Kang, M. A PDE-based fast local level set method. *J. Comput. Phys. 155* (1999), 410–438.

[37] Percy, J., and Mace, R. OpenGL extensions: Siggraph 2003. `http://mirror.ati.com/developer/techpapers.html`, 2003.

[38] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[39] Rademacher, P. Glui, the OpenGL user interface library. `http://www.cs.unc.edu/~rademach/glui/`, 1999.

[40] Rumpf, M., and Strzodka, R. Level set segmentation in graphics hardware. In *International Conference on Image Processing* (2001), pp. 1103–1106.

[41] Russell, R. M. The cray-1 processor system. *Communications of the ACM 21*, 1 (1978), 63–72.

[42] Sabella, P. A rendering algorithm for visualizing 3D scalar fields. In *ACM Computer Graphics (SIGGRAPH '88 Proceedings)* (August 1988), pp. 51–58.

[43] Segal, M., and Akeley, K. The OpenGL graphics system: A specification (version 1.4). `http://www.opengl.org`, 2003.

[44] Sethian, J. A. *Level Set Methods and Fast Marching Methods Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999.

[45] SGI. Open inventor. http://oss.sgi.com/projects/inventor/, 2003.

[46] SGI, and Scientific Computing and Imaging Institute at the University of Utah. Star-ray interactive ray tracer demo at Siggraph 2002. http://www.sci.utah.edu/stories/2002/sum_star-ray.html, 2002.

[47] Sherbondy, A., Houston, M., and Nepal, S. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *IEEE Visualization* (October 2003), pp. 171–196.

[48] Silberschatz, A., and Galvin, P. *Operating System Concepts*. Addison-Wesley, 1998.

[49] Stewart, N. Glt OpenGL C++ Toolkit. http://www.nigels.com/glt/, 2002.

[50] Strzodka, R., and Rumpf, M. Using graphics cards for quantized FEM computations. In *Proceedings VIIP Conference on Visualization and Image Processing* (2001).

[51] Taiwan, E. Powerstrip. http://www.entechtaiwan.net/ps.htm.

[52] The Insight Toolkit. http://www.itk.org, 2003.

[53] Udupa, J., LeBlanc, V., Schmidt, H., Imielinska, C., Saha, P., Grevera, G., Zhuge, Y., Currie, L., Molholt, P., and Jin, Y. A methodology for evaluating image segmentation algorithms. In *Proceedings of SPIE Vol. 4684* (2002), SPIE, pp. 266–277.

[54] Warfield, S. K., Kaus, M., Jolesz, F. A., and Kikinis, R. Adaptive, template moderated, spatially varying statistical classification. *Medical Image Analysis 4*, 1 (2000), 43–45.

[55] Warfield, S. K., Zou, K. H., and Wells, W. M. Validation of image segmentation and expert quality with an expectation-maximization algorithm. In *MICCAI 2002: Fifth International Conference on Medical Image Computing and Computer-Assisted Intervention* (Heidelberg, Germany, 2002), Springer-Verlag, pp. 298–306.

[56] Whitaker, R. T. Volumetric deformable models: Active blobs. In *Visualization In Biomedical Computing 1994* (Mayo Clinic, Rochester, Minnesota, 1994), R. A. Robb, Ed., SPIE, pp. 122–134.

[57] Whitaker, R. T. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision 29*, 3 (1998), 203–231.

[58] Xue, X., and Whitaker, R. Variable-conductance, level-set curvature for image denoising. In *IEEE International Conference on Image Processing* (October 2001), pp. 142–145.