Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware

Aaron Lefohn   Joe Kniss   Charles Hansen   Ross Whitaker

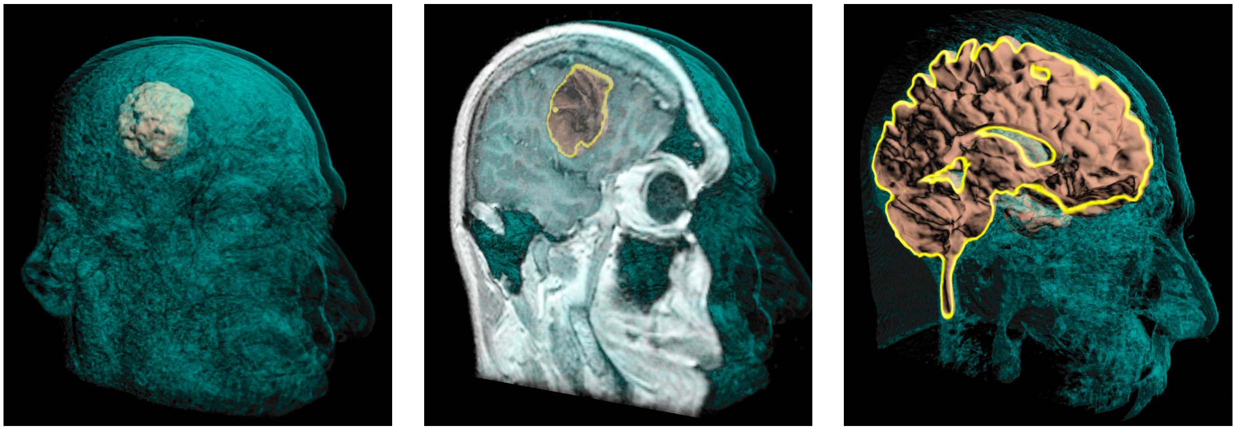
University of Utah, School of Computing
Technical Report UUCS-03-005

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

April 16, 2003

Abstract

Deformable isosurfaces, implemented with level-set methods, have demonstrated a great potential in visualization for applications such as segmentation, surface processing, and surface reconstruction. Their usefulness has been limited, however, by two problems. First, 3D level sets are relatively slow to compute. Second, their formulation usually entails several free parameters that can be difficult to tune correctly for specific applications. The second problem is compounded by the first. This paper presents a solution to these challenges by describing graphics processor (GPU) based algorithms for solving and visualizing level-set solutions at interactive rates. Our efficient GPU-based solution relies on packing the level-set isosurface data into a dynamic, sparse texture format. As the level set moves, this sparse data structure is updated via a novel GPU to CPU message passing scheme. When the level-set solver is integrated with a real-time volume renderer operating on the same packed format, a user can visualize and steer the deformable level-set surface as it evolves. In addition, the resulting isosurface can serve as a region-of-interest specifier for the volume renderer. This paper demonstrates the capabilities of this technology for interactive volume visualization and segmentation.



(a) (b) (c)

Figure 1: (a) 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. (b) A clipping plane shows the user the source data, the volume rendering, and the segmentation simultaneously, while probing data values on the plane. (c) The cerebral cortex segmented from the same data. The yellow band indicates the outline of the level-set model on the clipping plane.

1 Introduction

Level-set methods [1] rely on partial differential equations (PDEs) to model deforming isosurfaces. These methods have applications in a wide range of fields such as visualization, scientific computing, computer graphics, and computer vision [2, 3]. Applications in visualization include volume segmentation [4, 5, 6], surface processing [7, 8], and surface reconstruction [9, 10].

The use of level sets in visualization can be problematic. Level sets are relatively slow to compute and they typically introduce several free parameters that control the surface deformation and the quality of the results. 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. Although efforts have been made to take advantage of the sparse nature of the computation, the most highly optimized solvers are still far from interactive. This paper proposes a solution to the above problems by mapping the level-set PDE solver to a commodity graphics processor (GPU).

While the proposed technology has a wide range of uses within visualization and elsewhere, this paper focuses on a particular application: the visualization and analysis of volume data. By accelerating the PDE solver to interactive rates and coupling it to a real-time volume renderer, it is possible to visualize and steer the computation of a level-set surface as it moves toward interesting regions within a volume. The volume renderer, with its global visualization capabilities, provides context for the evolving level set. Also, the results of a level-set segmentation can specify a region of interest for the volume renderer [11].

The main contributions of this paper are:

- An integrated system that demonstrates level-set computations can be intuitively controlled by coupling a real-time volume renderer with an interactive solver.
- The design of a GPU-based 3D level-set solver, which is approximately 15 times faster than previous optimized solutions.
- A dynamic, packed texture format that enables the efficient processing of time-dependent, sparse GPU computations.
- A novel message passing scheme between the GPU and CPU that uses automatic mipmap generation to encode messages and update textures.
- Real-time volume rendering directly from this packed texture format.

The following section discusses previous work and background for level sets, GPUs and hardware accelerated volume rendering. Section 3 discusses the algorithmic and graphics hardware details of our level-set solver and volume renderer. Section 4 describes our segmentation application and compares our results to previous implementations. In section 5, we give conclusions, describe future research directions, and make suggestions for future GPU improvements.

2 Background and Related Work

2.1 Level Sets

This paper describes a new solver for an implicit representation of deformable surface models called the method of *level sets* [1]. The use of level sets has been widely documented in the visualization literature, and several works give comprehensive reviews of the method and the associated numerical techniques[2, 3]. Here we simply review the notation and describe the particular formulation that is relevant to this paper.

In an implicit model the surface consists of all points $\mathcal{S} = \{\bar{x} | \phi(\bar{x}) = 0\}$, where $\phi : \Re^3 \mapsto \Re$. Level-set methods relate the motion of that surface to a PDE on the volume, i.e.

$$\partial\phi/\partial t = -\nabla\phi \cdot \bar{v}, \tag{1}$$

where $\bar{v}$, which can vary of space and time, describes the motion of the surface. Within this framework one can implement a wide range of deformations by defining an appropriate $\bar{v}$. This velocity (or speed) 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.

Solving level-set PDEs on a volume requires proper numerical schemes [1] and entails a significant computational burden. Stability requires that the surface can progress at most a distance of one voxel at each iteration, and thus a large number of iterations are required to compute significant deformations. The purpose of this paper is to offer a solution that is relevant to a wide variety of level-set applications; that is, the ability to solve such equations efficiently on commodity graphics hardware.

There is a special case of Eq. 1 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* [2] and variations thereof [12]. However, this case covers only a very small subset of interesting speed functions. In general we are concerned with problems that require a curvature term and simultaneously require the model to expand and contract, such as those discussed in [8, 6, 10].

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 $\phi = 0$). In light of this observation several authors have proposed numerical schemes that compute solutions for only those voxels that lie in a small number of layers adjacent to the surface. Adalsteinson and Sethian [13] have proposed the *narrow band method*, which updates the embedding, ϕ, on a band of 10-20 pixels around the model, and reinitializes that band whenever the model approaches the edge. Whitaker [14] proposed the *sparse-field* method, which introduces a scheme in which updates are calculated only on the wavefront, and several layers around that wavefront are updated via a distance transform at each iteration. A similar strategy is described in [15]. 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 our GPU-based solver.

2.2 Scientific Computation on Graphics Processors

Graphics processing units (GPUs) 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 non-gaming applications.

For many data-parallel computations, graphics processors out-perform central processing units (CPUs) by more than an order of magnitude because of their *streaming* architecture [16] 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. In contrast to vector architectures, the computation kernel in a streaming architecture may consist of many (possibly thousands) of instructions and use temporary registers to hold intermediate values. The GPU takes advantage of the data-level parallelism inherent in the streaming model by having many identical processing units execute the computation in parallel.

Currently GPUs must be programmed via graphics APIs such as OpenGL [17] or DirectX [18]. Therefore all computations must be cast in terms of computer graphics primitives such as vertices, textures, texture coordinates, etc. Figure 2 depicts the computation pipeline of a typical GPU. 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, such as the level-set partial differential equations, are solved by first transferring the initial data into texture memory. The GPU performs the computation by rendering graphics primitives that address this texture. In the simplest case, a two-dimensional array of data undergoes some computation by drawing a quadrilateral that has 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 3D grid is processed as a sequence of 2D 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, Navier-Stokes solvers, and others [19, 20, 21]. All of these examples demonstrate a homogeneous sequence of operations over a densely populated grid structure.

Rumpf *et. al.* [22] 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. Lefohn and Whitaker demonstrate a full three dimensional level-set solver, with curvature, running on a graphics processor [23]. Neither of these approaches, however, take advantage of the sparse nature of level-set PDEs and therefore they perform only marginally better (e.g. twice as fast) than sparse or narrow band CPU implementations.

This paper presents a GPU computational model that supports *sparse and dynamic* 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 sparse matrix solver presented in [24] and previous work on rendering with compressed data [25, 26]. Section 3 describes how our design addresses these challenges.

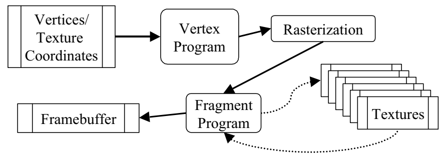


Figure 2: The modern graphics processor pipeline.

2.3 Hardware-Accelerated Volume Rendering

Volume rendering is a flexible and efficient technique for creating images from 3D data [27, 28, 29]. With the advent of dedicated hardware for rasterization and texturing, interactive volume rendering has become one of the most widely used techniques for visualizing moderately sized 3D rectilinear data [30, 31]. In recent years, graphics hardware has become more programmable, permitting rendering features with an image quality that rival sophisticated software techniques [32, 33, 34]. In this paper, we describe a novel volume rendering system that leverages programmable graphics hardware to simultaneously render the level-set solution and source data.

3 Implementation

This section gives a technical description of our implementation. We begin with a high-level description of the algorithms used for both the sparse-grid, streaming, level-set solver and the real-time volume renderer. We then cover some of the implementation details that are specific to the architecture of current graphics processors.

3.1 Algorithmic Details

3.1.1 GPU Level-Set Solver

The efficient solution of the level-set PDEs relies on updating only those voxels that are on or near the isosurface. The narrow band and sparse field methods achieve this by operating on sequences of heterogeneous operations. For instance, the sparse-field method [14] keeps a linked list of *active* voxels on which the computation is performed. Such algorithms are not well suited for streaming architectures and thus the mapping of the sparse-field algorithm to GPUs requires a very different approach.

The sparse GPU level-set solver decomposes the volume into a set of small 2D tiles (e.g. 16 x 16 pixels each). Only those tiles with non-zero derivatives are stored on the GPU (see Fig. 3). These *active* tiles are packed, in an arbitrary order, into a large 2D texture on the GPU. The 3D level-set PDE is computed directly on this packed format.

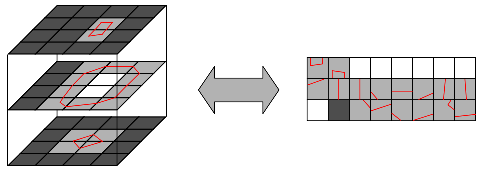


Figure 3: The spatial decomposition scheme for packing active regions of the volume into texture memory. The CPU-based tracks the location of each tile in texture memory.

Three-dimensional neighborhood information is reconstructed through texture coordinates from all neighboring tiles. Two data structures, a *packed map* and *unpacked map*, are kept on the CPU to track each tile's packed and unpacked position. The packed map stores the volumetric location of each tile in the sparse, GPU texture. The unpacked map stores a pointer to an abstract *tile* object that contains the vertices and texture coordinates for the actual texture data. There are two special tiles set aside for *white* and *black* regions. Tiles that are not active (i.e. homogeneous in value) are either inside or outside of the level set, and are mapped to either the white or black tile in texture memory. Also note that the vertices are replicated for each tile because each tile needs its own set of texture coordinates in order to find its neighboring tiles. A diagram of these mapping is shown in Fig. 3.

Neighbor lookups across tile boundaries represent eight special cases (four corners and four edges) of texture lookups. We render geometry to draw only those pixels in each special case and send texture coordinates that identify all 3D neighbors for those cases. This method allows for all data points in each case (e.g. all left-edge pixels from all tiles) to be processed in the same render pass, and thus take maximum advantage of the parallelism in the GPU.

There are several important details that make this strategy effective. First, because active tiles are identified by non-zero gradients, it is crucial that the volume in which the level-set surface is embedded, ϕ, resemble a clamped distance transform. In this way regions on or near the model will have finite derivatives, while tiles outside this narrow band will be flat (white or black), with derivative values of zero (and thereby undergo no change for that iteration). This is accomplished by adding an additional speed term to the velocity term $\bar{v}(t)$ in Eq. 1. This *rescaling* term, G_r is of the form,

$$G_r = \phi g_\phi - \phi|\nabla\phi|, \tag{2}$$

where ϕ is the value of the embedding at a voxel and $|\nabla\phi|$ is the gradient in the direction of the isosurface. The target gradient, g_ϕ, is set based the numerical precision of the level-set data. This speed term is strictly a numerical construct; it does not affect the movement of the zero level set, i.e. the surface model.

After the GPU updates the level-set data, it creates a compressed, encoded message. The CPU reads this message to determine the status of all tiles for the next pass. The GPU computes this message in several steps. First it produces, for each pixel in an active tiles, an eight byte code (2 four-channel images) which indicates if a pixel has any nonzero derivatives,

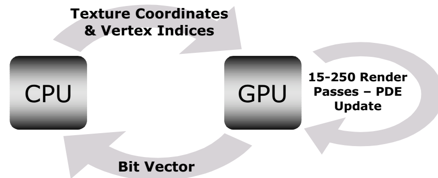


Figure 4: Flow diagram of the GPU-based level-set solver.

if it is a boundary pixel and has a nonzero derivative perpendicular to the boundary (for each of the six cardinal directions), and if the pixel is white. The GPU down-samples these images using the automatic mipmap generation feature combined with a fragment program that reduces each channel to a single bit. The result is small bit-vector image, one pixel per tile, that encodes the overall color of the tile and derivative information within the tile and across each boundary. This image (< 64KB) is read back by the CPU and decoded. Using these eight bits the CPU can determine how to configure the tile for the next iteration. It activates new tiles (white or black as appropriate), frees tiles that are no longer active, and updates the packed and unpacked maps described above.

Figure 4 shows a flow diagram of the computation. The pseudocode for the GPU portion of the computation is given in Fig. 5. Because of the packed representation and arbitrary positioning of the tiles, the neighbor lookups are relatively expensive. Therefore our design ensures that pixel neighborhood lookups occur only once, which is during the computation of volume derivatives. To ensure modularity, we have encapsulated each render pass in a function call, where the input textures are arguments and the output textures are the return value(s). For increased generality the application can create level-set speed functions as modules and pass them to the solver as function arrays. In the pseudocode below, d is an array of four textures that contain 1st and 2nd partial derivatives, p is the packed level-set texture, s is an array of evaluated speed function textures, and t is an array of 2 textures containing active tile information.

3.2 Volume Rendering of Packed Data

Our volume renderer performs a full 3D (transfer-function based) volume rendering of the original data simultaneously with the evolving level set. For rendering the original volume, the input data and its gradient vectors are kept on the GPU as 3D textures. The volume data is rendered on the GPU with multidimensional transfer functions as described in [34].

For rendering the evolving level-set model, we use a modification of the conventional 2D sliced approach to texture-based volume rendering [30]. We modify the conventional approach to render the level-set solution directly from the packed tiles, which are stored in a single 2D texture. The level-set data and tile configuration are dynamic, and therefore we do not precompute and store the three separate versions of the data, sliced along cardinal views, as is typically done with 2D texture approaches. Instead we reconstruct these views as needed.

```
for each level-set iteration, n
  // Compute 19, 1st and 2nd Partial Derivs
  for each 9 neighbor lookup cases, i
    d = computeDerivs[i]( p )

  // Compute speed terms
  for each speed function, i
    s[i] = speedFunc[i]( d, p, srcData )

  // Update level-set PDE
  p  = updateLS( d, speed, p )

  // Send tile information message to CPU
  t[0]   = interiorTileInfo( d, p )

  for each 8 tile boundary cases, i
    t[1] = bndryTileInfo[i]( d, p )

  message = makeBitVector( t )
  updateTiles( message )
```

Figure 5: Pseudocode for the GPU-based level-set solver.

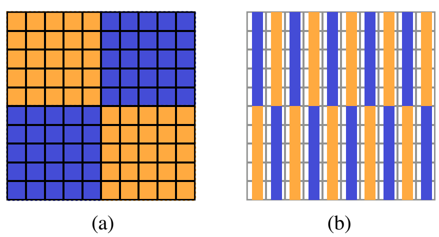


Figure 6: For volume rendering the packed level-set model: (a) When the preferred slicing direction is orthogonal to the packed texture, the tiles (shown in alternating colors) are rendered into slices as quadrilaterals. (b) For slicing directions parallel to the packed texture, the tiles are drawn onto slices as either vertical or horizontal lines.

The 2D slice-based rendering requires interpolation between two adjacent slices in the back-to-front ordering along the appropriate cardinal direction. When reconstructing these two slices on the fly from the packed level-set data, two cases must be considered. The first case is when the preferred slice axis, based on the viewing angle, is orthogonal to the packed texture. In this case the slices can be reconstructed using quadrilaterals, one for each tile in the level-set model. If the preferred slice direction is parallel to the packed texture, we must reconstruct those slices by rendering a row or column from each tile using textured line primitives. Figure 6 illustrates the two cases for 2D slice-based rendering of the level-set model.

For efficiency the renderer reuses data wherever possible. For instance, lighting for the level-set solution uses gradient vectors computed in the level-set update stage. The rendering of the source data relies on precomputed gradient data—the gradient magnitude is used by the transfer function and the gradient direction is used in the lighting model.

3.3 Graphics Hardware Implementation Details

This subsection describes implementation details that are specific to the current generation of graphics hardware. Suggestions for future graphics hardware features are given in Sec. 5.

The level-set solver and volume renderer are implemented in programmable graphics hardware using vertex and fragment programs on the ATI Radeon 9700 GPU. The programs are written in the OpenGL ARB_vertex_program and ARB_fragment_program assembly languages. The bulk of the computations are performed in fragment programs, but vertex programs are used to efficiently compute texture coordinates for neighbor lookups; therefore minimizing both AGP bandwidth and valuable fragment instructions.

Critical to the performance of the system are two capabilities pertaining to render pass destination buffers. The first capability, relatively recent on GPUs, is the ability to output multiple, high-precision 4-tuple results from a fragment program. Multiple outputs enable us to perform the expensive 3D neighborhood reconstruction only once and use the gathered data to compute all derivatives in the same pass. The second feature crucial to the performance is the ability to quickly change render pass destination buffers. As discussed in [24], current display drivers require the OpenGL render context to change in order to change render targets. This operation is unnecessary and expensive—it can take up to 0.3 milliseconds. To avoid this overhead we allocate a single buffer with many *render surfaces* (front, back, aux0, etc.) and switch between them. When the complexity of the computation requires more intermediate buffers, we use sub-regions of larger buffers to augment this multisurface approach.

There is a subtle speed-versus-memory tradeoff that must be carefully considered. The packed level-set texture can be as large as 2048^2 (the largest 2D texture currently allowed on GPUs). In order to minimize the memory costs of the intermediate buffers (derivatives, speed values, etc.), the level-set data is updated in sub-regions. Minimizing the number of these sub-regions is important because adding a large number of render passes introduces a significant amount of overhead and reduces computational efficiency. We currently use 512^2 sub-regions when the level-set texture is 2048^2 and use a single region when it is smaller.

4 Application and Results

This section describes an application for interactive volume segmentation and visualization, which uses the level-set solver described previously. The system combines interactive level-set models with real-time volume rendering on the GPU. We show pictures from the system (also see the associated video) and present timing results relative to our current benchmark for level-set deformations, which is a highly optimized CPU solution.

4.1 Volume Visualization and Analysis

For segmenting volume data with level sets, the velocity usually consists of a combination of two terms [4, 5]

$$\frac{\partial \phi}{\partial t} = |\nabla \phi| \left[\alpha D(\bar{x}) + (1-\alpha) \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} \right], \tag{3}$$

where D is a data term that forces the model toward desirable features in the input data, the term $\nabla \cdot (\nabla\phi/|\nabla\phi|)$ is the mean curvature 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. There are several variations on this approach in the literature, e.g. [35].

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 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. Thus, 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 paper we have chosen a simple speed function to demonstrate the effectiveness of *interactivity* and *real-time visualization* in level-set solvers. The speed function we use in this work depends solely on the input data I at the point $\bar{x}$. Thus it is a grey scale transformation of the input intensity:

$$D(I) = \epsilon - |I - T|, \tag{4}$$

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. When the model lies on a voxel with a greyscale level between $T - \epsilon$ and $T + \epsilon$, the model expands and otherwise it contracts. The speed term is gradual, as shown in Fig. 7, 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. To control the model a user specifies three free parameters, T, ϵ, and α, *as well as* an initialization. The user generally places the 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.

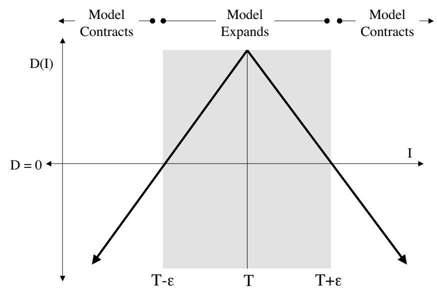


Figure 7: A speed function based on image intensity causes the model to expand over regions with greyscale values wit erwise.

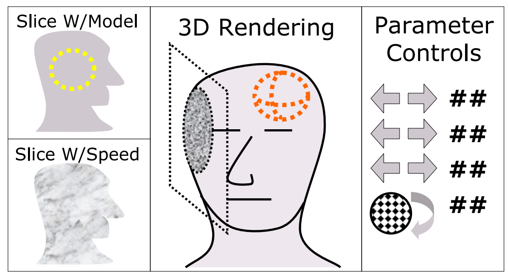


Figure 8: The GUI for the volume analysis application. Users interact via slice views, a 3D rendering, and a control panel.

4.2 Interface and Usage

The application in this paper consists of a graphical user interface that presents the user with two slice viewing windows, a volume renderer, and a control panel (Fig. 8). 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 2D window displays the current segmentation as a yellow line overlaid on top of the source data. The second 2D window displays a visualization of the level-set speed function that clearly delineates the positive and negative 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 3D spherical initializations for the level-set solver. The first two 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 3D reconstruction of the current level set isosurface as well as the input data. In addition, an arbitrary clipping plane, with texture-mapped source data, can be enabled via the GUI (Fig. 1b). 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.

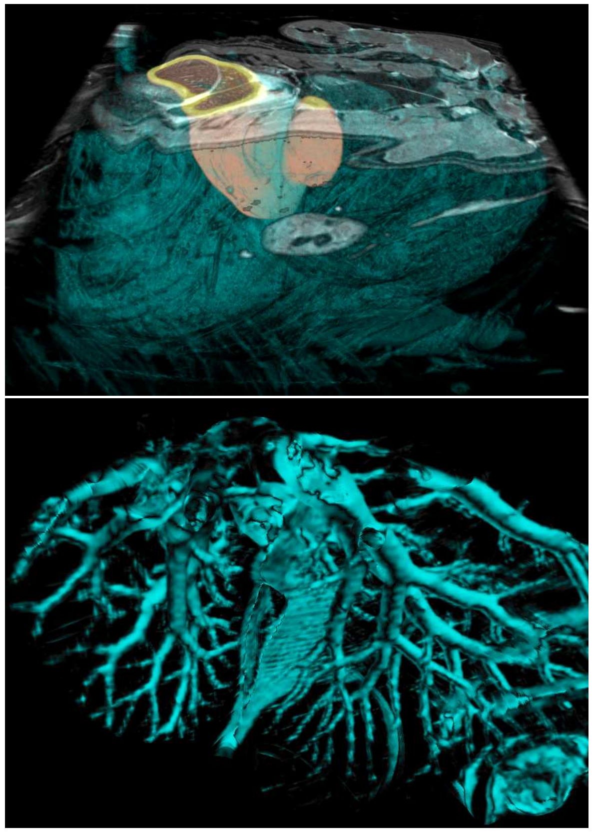


Figure 9: (top) Volume rendering of a 256^3 MRI scan of a mouse thorax. Note the level set surface which is deformed to segment the liver. (bottom) Volume rendering of the vasculature inside the liver using the same transfer function as in (a) with the level-set surface is being used as a region of interest specifier.

The volume renderer uses a 2D transfer function to render the level set surface and a 3D 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 [36].

We demonstrate our interactive level-set solver and volume rendering system with the following three data sets: a brain tumor MRI (Fig. 1), an MRI scan of a mouse (Fig. 9), and transmission electron tomography data of a gap junction (Fig. 10). 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.

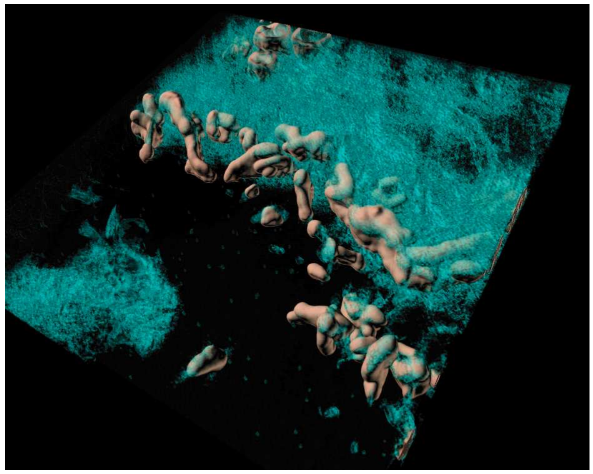


Figure 10: Segmentation and volume rendering of $512 \times 512 \times 61$ 3D transmission electron tomography. The picture shows cytoskeletal membrane extensions and connexins (pink surfaces extracted with the level-set models) near the gap junction between two cells (volume rendered in cyan).

4.3 Performance Analysis

Our GPU-based level-set solver achieves a speedup of ten to fifteen times over a highly-optimized, sparse-field, CPU-based solver. All benchmarks were run on an Intel Xeon 1.7 GHz processor with 1 GB of RAM and an ATI Radeon 9700 Pro GPU. 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 (Fig. 1). 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.

A profile of the the level-set solver reveals the following distribution of execution time: 70% on GPU arithmetic instructions, 15% on texture memory reads, 10% on CPU performing bit vector readback and updating the active tiles, and 5% on transferring data across the AGP bus. These estimates were made based on the profiling techniques described in [37]. Creating the bit vector message consumes approximately 15% of the GPU arithmetic and texture instructions. The entire sparse algorithm adds a 15%-20% computation overhead, but for most applications the speedup over a dense GPU-based implementation far eclipses this additional overhead.

5 Conclusions

This papers 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. set solver interactive By relying on graphics hardware the level-set solver operates at interactive rates (approximately 15 times faster than previous solutions).

This mapping relies on a novel dynamic, packed texture and a GPU-to-CPU message passing scheme. While the GPU updates the level set, it renders the surface model directly from this packed texture format. Future extensions and applications of the level-set solver include the processing of multivariate data as well as surface reconstruction and surface processing. Most of these only involve changing only the speed functions.

Another promising area of future work is to adapt these volume processing algorithms to leverage the evolving capabilities of GPUs. For instance, a current limitation with the proposed method is volume size. The efficiency of our memory usage is hampered by inflexibilities in the GPU memory model and instruction set. We have identified several new features that would alleviate this shortcoming. First, in order to spread the packed representation across multiple textures, we would need an efficient mechanism for rendering to subregions of a 3D buffer. Alternatively, a mechanism for dynamically specifying the source texture of a read operation would provide a similar capability—i.e. more indirection in texture reads. Another promising strategy for reducing memory usage is the development of better compression schemes. Better compression schemes could be facilitated by the addition of integer data types and bitwise operations into the fragment processor.

Current GPU capabilities also limit the computational efficiency of the proposed algorithms. We could achieve better computational efficiency within each tile if we could avoid processing pixels that are not sufficiently close to the surface, i.e. we could achieve an even narrower band of computation. This would require a more flexible depth and/or stencil culling mechanism in which multiple data buffers could access a single depth/stencil buffer. We could save additional fragment instructions by computing all texture addresses in the vertex stage. This would require more per-vertex interpolants. For instance, the sampling of a $3 \times 3 \times 3$ kernel requires *at least* 21, 4-tuple interpolants.

Acknowledgments

The authors would like to thank Gordon Kindlmann for his *nrrd* library (used for dataset manipulation and I/O), part of the *teem* toolkit available at `http://www.cs.utah.edu/~gk/teem`. Milan Ikits' *GLEW* sofware was also used extensively for OpenGL extension management. We also would like to thank Steve Lamont at the National Center for Microscopy and Imaging Research at the University of California San Diego for the transmission electron tomography data. In addition, Simon Warfield, Michael Kaus, Ron Kikinis, Peter Black and Ferenc Jolesz provided the tumor database. The mouse data was supplied by the Center for In Vivo Microscopy at Duke University. This work was supported by grants #ACI0089915 and #CCR0092065 from the National Science Foundation.

References

[1] S. Osher and J. Sethian, "Fronts propogating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations," *Journal of Computational Physics*, vol. 79, pp. 12–49, 1988.

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

[3] R. Fedkiw and S. Osher, *Level Set Methods and Dynamic Implicit Surfaces.* Springer, 2002.

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

[5] R. Malladi, J. A. Sethian, and B. C. Vemuri, "Shape modeling with front propogation: A level set approach," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 17, no. 2, pp. 158–175, 1995.

[6] R. Whitaker, D. Breen, K. Museth, and N. Soni, "A framework for level set segmentation of volume datasets," in *Proceedings of ACM Intnl. Wkshp. on Volume Graphics*, pp. 159–168, June 2001.

[7] T. Tasdizen, R. Whitaker, P. Burchard, and S. Osher, "Geometric surface smoothing via anisotropic diffusion of normals," in *Proceedings of IEEE Visualization*, pp. 125–132, October 2002.

[8] D. Breen and R. Whitaker, "A level-set approach to 3d shape metamorphosis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, no. 2, pp. 173–192, 2001.

[9] R. Whitaker and V. Elangovan, "A direct approach to estimating surfaces in tomographic data," *Medical Image Analysis*, vol. 6, pp. 235–249, 2002.

[10] K. Museth, D. Breen, L. Zhukov, and R. Whitaker, "Level-set segmentationfrom multiple non-uniform volume datasets," in *IEEE Visualization 2002*, pp. 179–186, October 2002.

[11] T. Yoo, U. Neumann, H. Fuchs, S. Pizer, T. Cullip, J. Rhoades, and R. Whitaker, "Direct visualization of volume data," *IEEE Computer Graphics and Applications*, vol. 12, no. 4, pp. 63–71, 1992.

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

[13] D. Adalsteinson and J. A. Sethian, "A fast level set method for propogating interfaces," *Journal of Computational Physics*, pp. 269–277, 1995.

[14] R. T. Whitaker, "A level-set approach to 3D reconstruction from range data," *International Journal of Computer Vision*, vol. October, no. 3, pp. 203–231, 1998.

[15] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang, "A pde based fast local level set method," *J. Comput. Phys.*, vol. 155, pp. 410–438, 1999.

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

[17] M. Segal and K. Akeley, "The OpenGL graphics system: A specification (version 1.2.1)." http://www.opengl.org, 2003.

[18] Microsoft Corporation, "Direct3D." http://www.microsoft.com/directx, 2002.

[19] M. Rumpf and R. Strzodka, "Using graphics cards for quantized FEM computations," in *IASTED Visualization, Imaging and Image Processing Conference*, 2001.

[20] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, "Physically-based visual simulation on graphics hardware," in *Proc. SIGGRAPH/EG Graphics Hardware Workshop '02*, ACM, 2002.

[21] W. Li, X. Wei, , and A. Kaufman, "Implementing lattice boltzmann computation on graphics hardware," in *The Visual Computer*, (Heidelberg, Germany), Springer-Verlag, to appear 2003.

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

[23] A. Lefohn and R. Whitaker, "A gpu-based, three-dimensional level set solver with curvature flow." University of Utah tech report UUCS-02-017, December 2002.

[24] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "The gpu as numerical simulation engine," in *ACM SIGGRAPH*, p. *To Appear*, 2003.

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

[26] M. Kraus and T. Ertl, "Adaptive Texture Maps," in *Proc. SIGGRAPH/EG Graphics Hardware Workshop '02*, pp. 7–15, 2002.

[27] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics & Applications*, vol. 8, no. 5, pp. 29–37, 1988.

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

[29] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," in *ACM Computer Graphics (SIGGRAPH '88 Proceedings)*, pp. 51–58, August 1988.

[30] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," in *ACM Symposium On Volume Visualization*, 1994.

[31] O. Wilson, A. V. Gelder, and J. Wilhelms, "Direct Volume Rendering via 3D Textures," Tech. Rep. UCSC-CRL-94-19, University of California at Santa Cruz, June 1994.

[32] C.Rezk-Salama, K.Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization," in *Siggraph/Eurographics Workshop on Graphics Hardware 2000*, 2000.

[33] K. Engel, M. Kraus, and T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading," in *Siggraph/Eurographics Workshop on Graphics Hardware 2001*, 2001.

[34] J. Kniss, G. Kindlmann, and C. Hansen, "Multi-Dimensional Transfer Functions for Interactive Volume Rendering," *Trasactions on Visualization and Computer Graphics*, vol. 8, pp. 270–285, July-September 2002.

[35] V. Caselles, R. Kimmel, and G. Sapiro, "Geodesic active contours," in *Fifth International Conference on Computer Vision*, pp. 694–699, IEEE, IEEE Computer Society Press, 1995.

[36] J. Kniss, S. Premoze, M. Ikits, A. Lefohn, and C. Hansen, "Gaussian transfer functions for multi-field volume visualization." Under review: *IEEE Visualization 2003*.

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