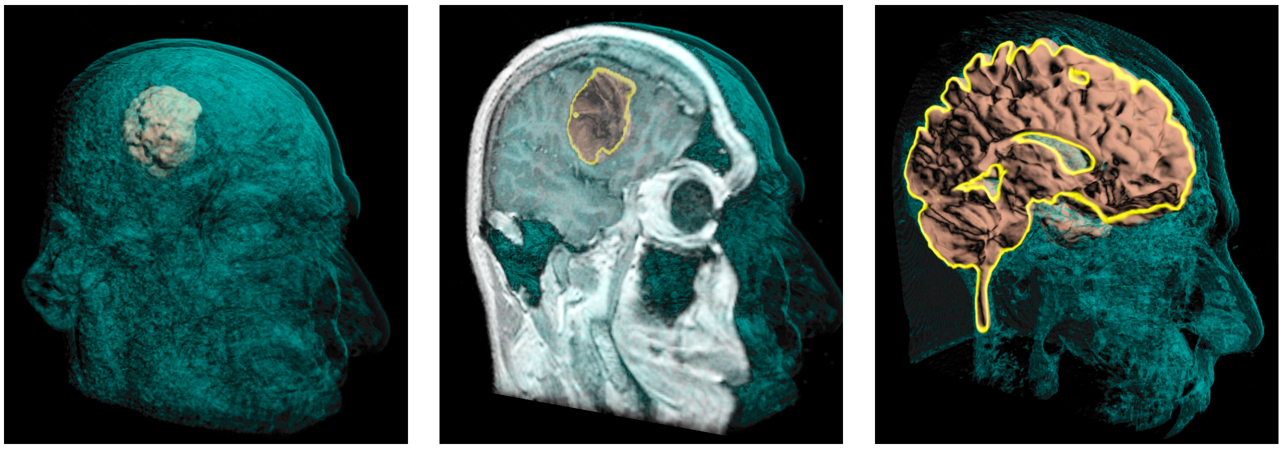
Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware

Aaron E. Lefohn   Joe M. Kniss   Charles D. Hansen   Ross T. Whitaker

Scientific Computing and Imaging Institute, University of Utah*



(a) (b) (c)

Figure 1: (a) Interactive level set segmentation of a brain tumor from a 256 × 256 × 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 intersection of the level-set model with the clipping plane.

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 their high computational cost and and reliance on significant parameter tuning. 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.

*e-mail:{lefohn|jmk|hansen|whitaker}@sci.utah.edu

CR Categories: I.3.3 [Computer Graphics]—Computational Geometry and Object Modeling,Methodology and Techniques

Keywords: Deformable Models, Image Segmentation, Volume Visualization, GPU, Level Sets, Streaming Computation

1 Introduction

Level-set methods [Osher and Sethian 1988] 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 [Fedkiw and Osher 2002; Sethian 1999]. Applications in visualization include volume segmentation [Malladi et al. 1995; Whitaker 1994], surface processing [Tasdizen et al. 2002], and surface reconstruction [Whitaker 1998].

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.

While the proposed technology has a wide range of uses within visualization and elsewhere, this paper focuses on a particular application: the analysis and visualization 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 [Yoo et al. 1992].

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.
- 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.
- Real-time volume rendering directly from this packed texture format.
- A novel message passing scheme between the GPU and CPU that uses automatic mipmap generation to create compact, encoded messages.

The following section discusses previous work and gives some technical 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. It gives the specific form of the level-set equations and desrcribes the results of a performance analysis. 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* [Osher and Sethian 1988]. 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 [Fedkiw and Osher 2002; Sethian 1999]. Here we merely review the notation and describe the particular formulation that is relevant to this paper.

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

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

where $\bar{v}$, which can vary in 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. Equation 1 is the general form of the level-set equation, which can be tuned for wide variety of problems and which motivates the architecture of our solver. We describe the specific form used for volume segmentation in Sect. 4.1.

Solving level-set PDEs on a volume requires proper numerical schemes [Osher and Sethian 1988] 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* [Sethian 1999] and variations thereof [Droske et al. 2001]. 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.

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 [1995] 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 [1998] 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 Peng et al. [1999]. 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 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 [Owens 2002] 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 or DirectX. 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 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 computation is

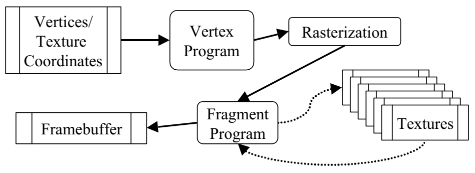


Figure 2: The modern graphics processor pipeline.

performed on all elements of a 2D texture 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, multigrid solvers, and others [Goodnight et al. 2003; Larsen and McAllister 2001; Strzodka and Rumpf 2001]. All of these examples demonstrate a homogeneous sequence of operations over a densely populated grid structure.

Strzodka et al. [2001] 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 [2002]. 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 recent sparse matrix solvers [Bolz et al. 2003; Krüger and Westermann 2003] and previous work on rendering with compressed data [Beers et al. 1996; Kraus and Ertl 2002]. Recent work by Sherbondy et al. [2003] describes a dynamic, sparse GPU computation model and is discussed in Section 4. Section 3 gives a detailed description of our solution to the sparse, dynamic computation problem.

2.3 Hardware-Accelerated Volume Rendering

Volume rendering is a flexible and efficient technique for creating images from 3D data [Drebin et al. 1988; Levoy 1988; Sabella 1988]. 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 [Cabral et al. 1994; Wilson et al. 1994]. In recent years, graphics hardware has become more programmable, permitting rendering features with an image quality that rival sophisticated software techniques [Engel et al. 2001; Kniss et al. 2002]. In this paper, we describe a novel volume rendering system that leverages programmable graphics hardware to simultaneously render the packed level-set solution and source data.

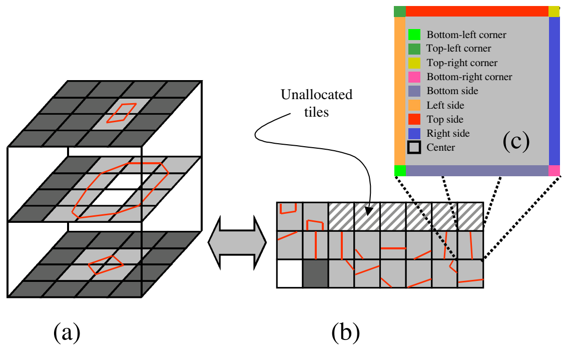


Figure 3: The spatial decomposition scheme for packing active regions of the volume into texture memory. The *unpacked* tile space is shown in (a) and the *packed* tile space is shown in (b). CPU-based data structures exist for both of these spaces. The only data stored on the GPU is that represented by (b). Three dimensional neighborhoods are efficiently reconstructed on the packed format by processing boundary pixels in nine separate special cases. These nine *substreams* are shown in (c).

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. Note that this section focuses on our new solution to the sparse/narrow-band computation problem. We therefore refer the reader to Lefohn et al. [2002] for a detailed description of the level-set equations.

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 [Sethian 1999] and sparse field [Whitaker 1998] methods achieve this by operating on sequences of heterogeneous operations. For instance, the sparse-field method keeps a linked list of *active* voxels on which the computation is performed.

Like the narrow band and sparse field CPU-based solvers, our sparse GPU level-set solver computes only those voxels near the isosurface. To run efficiently on GPUs, however, our solution must also have the following characteristics: texture-based data structures that can be efficiently updated, no *scatter* write operations, minimal memory requirements, and be highly data-parallel. We achieve these goals by decomposing the volume into a set of small 2D tiles (e.g. 16×16 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. The 3D level-set PDE is computed directly on this packed format. The CPU is used only to help manage the packing of the active data. Figure 4 shows a flow diagram of our level-set solver.

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

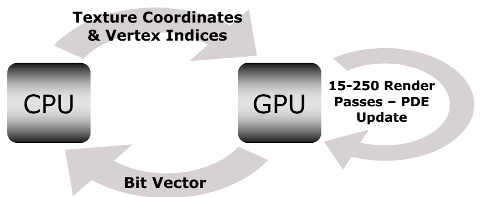

Texture Coordinates & Vertex Indices
CPU
GPU
15-250 Render Passes – PDE Update
Bit Vector

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

stores a *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 either inside or outside of the level set) are mapped to 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 locate its neighboring tiles. A diagram of these mappings are shown in Fig. 3.

The overview of the GPU portion of the computation is given below. The six steps shown are those required for a single iteration of the level-set PDE. See Lefohn et al. [2002] for an explanation of the twenty-one first and second derivatives and the discretization of the level-set equations.

1. Compute 21, 1st and 2nd partial derivatives. 9 *substream* passes—each to the same 4 buffers.
2. Compute N level-set speed terms. At least N passes.
3. Update level-set PDE. 1 pass.
4. Create eight-bytes of neighborhood info. 9 *substream* passes.
5. Down sample neighborhood information.
6. Create bit vector message. 1 pass.
7. Send bit vector to CPU.

The remaining portion of this section describes the details of the algorithm above. Step 1 is the only point in the computation when neighboring data values are read. The location of all necessary neighbor values is reconstructed on-the-fly by using texture coordinates to locate adjacent tiles in the 3D *unpacked* space. The position of data elements in relation to tile boundaries divides these *gather* operations into nine different cases: interior, corners, and edges (Fig. 3(c)). Rather than use a single fragment program to compute all nine cases, we instead create a specialized fragment program for each boundary case. Each specialized program is associated with geometry that rasterizes only the pixels needed for that case. We call this method of statically resolving conditionals using specialized fragment programs and geometry, *substreams*. The concept is a static implementation of the data-routing idea described Kapasi et al. [2000].

Our use of substreams is motivated by two characteristics of graphics hardware. The first is that GPUs do not support conditional execution in the fragment stage (all paths are executed and a single result is conditionally assigned). The second motivation is that the majority of the pixels are in the *interior* case, which has highly local neighbor lookups. In contrast, the neighbors for the eight boundary cases are almost never local, making texture caches almost useless. If we had instead combined all cases into one fragment program with an indirection texture to locate the address of each neighbor, neighbor lookups would be significantly slower for the common (interior) case.

Step 2 of the algorithm computes the speed terms described in Sect. 4. We add an additional term, however, to keep the volume in which the level-set is embedded, ϕ, resembling a clamped distance transform (CDT). This is necessary because active tiles are identified by non-zero gradients. The CDT ensures that voxels near the isosurface have finite derivatives while those farther away have gradient magnitudes of zero. Our new speed term is added 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 on 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. More detailed discussions of embedding-rescaling computations such as Eq. 2 can be found in the literature [Lefohn et al. 2003; Fedkiw et al. 1999].

After the solver updates the level-set data in step 3, it creates a compressed, bit-vector message. This message enables the CPU (in step 7) to determine which tiles are active in the next pass. This compressed message provides the CPU with aggregated information about each tile at each iteration, so that it can send vertices and texture coordinates for the new active set of tiles that the GPU will need in the next iteration. All of this communication between the CPU and GPU must be at the level of tiles to avoid a communication bottleneck. The aggregated tile description is generated on the GPU from a logical combination of the status of each pixel within each tile. This aggregation is performed efficiently by using the built-in mipmap generation functionality of the GPU.

The GPU creates the bit vector message in three stages—steps 4, 5, and 6. The first stage (step 4) creates information buffers that determine the active status of each voxel and its neighbors. The information buffers created in step 4 consist of eight bytes per active voxel. Each byte is set to either its maximum value (true) or zero (false). The first byte is set to true if any of the six, one-sided cardinal derivatives are non-zero. This determines if the voxel needs to be active on the next iteration. Each of the next six tests represent the active status of adjacent tiles in the unpacked 3D neighborhood. Each test is true only if a tile boundary is crossed in the corresponding direction and a non-zero derivative exists across that boundary. Note that the substream technique is used to process only those voxels that lie on tile boundaries. The eighth value is simply the level-set embedding value of the voxel.

In step 5, the solver uses the automatic mipmapping feature on the GPU to down sample these eight bytes of information until each tile is reduced to a single pixel. Any non-zero value in the original information buffers will result in a non-zero down sampled value for the entire tile.

Finally in step 6, the GPU creates the bit vector image/message by combining the eight bytes per pixel of down sampled data into a single eight-bit code for each pixel. The bit code is created with a fragment program that emulates a bitwise OR operation by conditionally adding power-of-two values. For each of the eight bytes that are non-zero, a unique power-of-two value is added to the final, single-byte result.

The CPU then reads back and decodes this small ($<$ 64 kB) bit-vector image in step 7. The bit code denotes whether a tile or any of the six adjacent tiles need to be active for the next iteration. The code also encapsulates whether a newly inactive tile is inside or outside the level-set surface. The CPU uses this information to activate new tiles (white or black as appropriate), frees tiles that are no longer active, and updates the packed and unpacked maps described above.

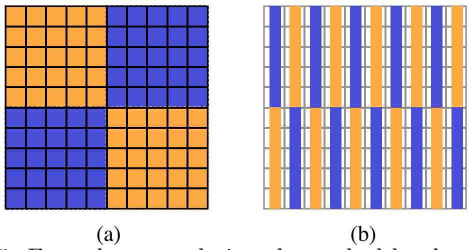


Figure 5: For volume rendering the packed level-set model: (a) When the preferred slicing direction is orthogonal to the packed texture, the tiles (shown in alternating colors) are rendered into slices as quadrilaterals. (b) For slicing directions parallel to the packed texture, the tiles are drawn onto slices as either vertical or horizontal lines.

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 Kniss et al. [2002].

For rendering the evolving level-set model, we use a modification of the conventional 2D sliced approach to texture-based volume rendering [Cabral et al. 1994]. 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 each time the volume is rendered.

The 2D slice-based rendering requires interpolation between two adjacent slices in the back-to-front ordering along the appropriate cardinal direction. We reconstruct each slice in *unpacked* space by texture mapping either quadrilateral or line primitives with data from the packed level-set texture. When the preferred slice axis, based on the viewing angle, is orthogonal to the unpacked slices, we reconstruct using textured quadrilateral for each tile. If the preferred slice direction is parallel to the unpacked slicing, we instead render a row or column from each tile using textured line primitives. Figure 5 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 surface uses gradient vectors computed during 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 9800 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. Vertex programs are used, however, to efficiently compute texture coordinates for neighbor lookups—thereby 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 Bolz et al. [2003] discuss, changing pbuffers is very expensive due to the unnecessary context switch. We avoid this overhead by allocating a single buffer with many *render surfaces* (front, back, aux0, etc.) and switching 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. We maximize the size of these sub-regions to keep computational efficiency as high as possible. 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 and present timing results relative to our current benchmark for level-set deformations, which is a highly optimized CPU solution [The Insight Toolkit 2003].

4.1 Volume Visualization and Analysis

For segmenting volume data with level sets, the velocity usually consists of a combination of two terms [Malladi et al. 1995; Whitaker 1994]

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

where D is a data term that forces the model to expand or contract toward desirable features in the input data, the term $\nabla \cdot (\nabla \phi / |\nabla \phi|)$ is the mean curvature $\mathcal{H}$ 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 corresponds to a surface velocity (from Eq. 1), $\bar{v} = \bar{n}(D+H)$, where $\bar{n}$ is the surface normal.

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. 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,

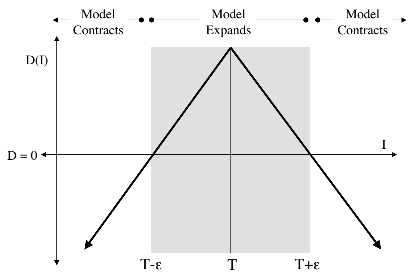


Figure 6: A speed function based on image intensity causes the model to expand over regions with greyscale values within the specified range and contract otherwise.

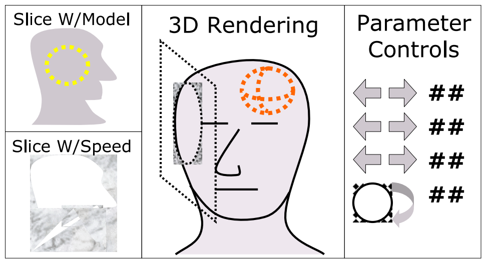


Figure 7: A depiction of the user interface for the volume analysis application. Users interact via slice views, a 3D rendering, and a control panel.

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 greyscale value input data I at the point $\bar{x}$:

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

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. 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 inverval will contract to exclude that voxel. The speed term is gradual, as shown in Fig. 6, 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 choice of D corresponds to a simple, one-dimensional statistical classifier on the volume intensity [Lefohn et al. 2003].

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 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. 7). 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.

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 [Kniss et al. 2003].

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. 8), and transmission electron tomography data of a gap junction (Fig. 9). 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.

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 implementation [The Insight Toolkit 2003]. All benchmarks were run on an Intel Xeon 1.7 GHz processor with 1 GB of RAM and an ATI Radeon 9800 Pro GPU. 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 (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.

The speed of our solver is bound almost entirely by the fragment stage of the GPU. In addition, the speed of our 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.

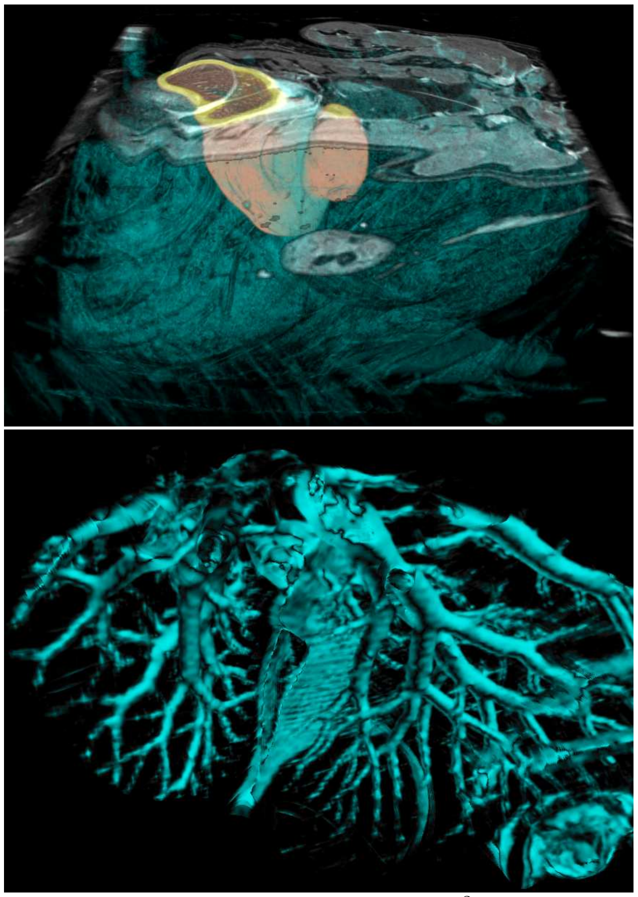


Figure 8: (top) Volume rendering of a 256^3 MRI scan of a mouse thorax. Note the level set surface which is deformed to segment the liver. (bottom) Volume rendering of the vasculature inside the liver using the same transfer function as in (a) with the level-set surface is being used as a region-of-interest specifier.

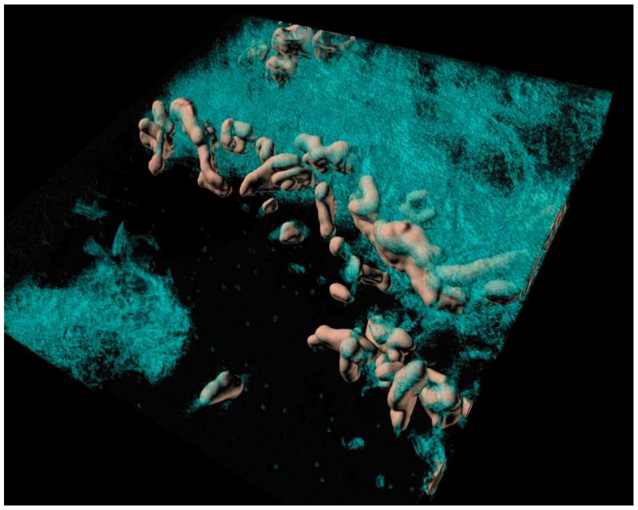


Figure 9: Segmentation and volume rendering of $512 \times 512 \times 61$ 3D transmission electron tomography data. The picture shows cytoskeletal membrane extensions and connexins (pink surfaces extracted with the level-set models) near the gap junction between two cells (volume rendered in cyan).

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 tiles. Our tests have shown that for many applications, only 10%-30% of the volume is active. To take full advantage of this savings, texture memory must be dynamically allocated as the surface expands. Our current implementation performs only static allocation, but future versions could easily realize the above memory savings. Section 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. [2003], our packing scheme 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 signficant portion of the execution time. In addition, our packing strategy allows us to 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. Our algorithm should also be able to process larger volumes, due to the memory savings discussed above. Our algorithm, however, does incur overhead associated with maintaining the tiles, and more experimentation is necessary to understand the circumstances under which each approach is advantageous. Furthermore, they are not mutually exclusive, and Sect. 5 discusses the possibility of using depth culling in combination with our packed representation.

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. 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, the efficiency of our memory usage is hampered by inflexibilities in the GPU memory model and instruction set. The first way in which we could use memory more efficiently is by spreading the packed representation across multiple textures. We could then dynamically allocate texture memory as needed and would not be limited to the maximum size of 2D textures. This approach requires either an efficient mechanism for rendering to a slice of a 3D buffer or the ability to dynamically select which texture is sampled (i.e. more indirection in texture reads). The former solution is now possible with the *uber buffer* [Percy and Mace 2003] OpenGL extension. A second strategy for reducing memory usage is the development of better compression schemes. Implementing these more aggressive compression algorithms will almost certainly require the ability to use integer data types and bitwise operations in 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 [Percy and Mace 2003]. In addition, 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 from a 3D texture requires *at least* 21, 4-tuple interpolants.

Future implementations of our algorithm could also take advantage of recently proposed higher-level shading language features. The Java-like *interfaces* proposed in Mark et al. [2003] could be used to separate memory access operations from arithmetic computation code. This would maximize code reuse for the nine specialized substream fragment programs because these programs differ only in the definition of their gather operation.

Acknowledgments

Thanks to Evan Hart, Mark Segal, Jeff Royal and Jason Mitchell at ATI for donating technical advice and hardware to this project. Gordon Kindlmann's *nrrd* toolkit was used for dataset manipulation. Milan Ikits' *GLEW* library was used for OpenGL extension management. Erik Jorgensen helped with production of the video. 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. This work was supported by grants from NSF, ACI0089915 and CCR0092065, and ONR N000140110033. We also thank John Owens and the anonymous reviewers for their input on the manuscript.

References

ADALSTEINSON, D., AND SETHIAN, J. A. 1995. A fast level set method for propogating interfaces. *Journal of Computational Physics*, 269–277.

BEERS, A. C., AGRAWALA, M., AND CHADDHA, N. 1996. Rendering from compressed textures. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 373–378.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *ACM Transactions on Graphics*, vol. 22, 917–924.

CABRAL, B., CAM, N., AND FORAN, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *ACM Symposium On Volume Visualization*, 91–98.

DREBIN, R. A., CARPENTER, L., AND HANRAHAN, P. 1988. Volume rendering. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, vol. 22, 65–74.

DROSKE, M., MEYER, B., RUMPF, M., AND SCHALLER, C. 2001. An adaptive level set method for medical image segmentation. In *Proc. of the Annual Symposium on Information Processing in Medical Imaging*, Springer, Lecture Notes Computer Science, R. Leahy and M. Insana, Eds.

ENGEL, K., KRAUS, M., AND ERTL, T. 2001. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Graphics Hardware 2001*.

FEDKIW, R., AND OSHER, S. 2002. *Level Set Methods and Dynamic Implicit Surfaces*. Springer.

FEDKIW, R., ASLAM, T., MERRIMAN, B., AND OSHER, S. 1999. A non-oscillatory Eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *Journal of Computational Physics 152*, 457–492.

GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003*, 102–111.

KAPASI, U., DALLY, W., RIXNER, S., MATTSON, P., OWENS, J., AND KHAILANY, B. 2000. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, 159–170.

KNISS, J., KINDLMANN, G., AND HANSEN, C. 2002. Multi-Dimensional Transfer Functions for Interactive Volume Rendering. *Transactions on Visualization and Computer Graphics 8* (July-September), 270–285.

KNISS, J., PREMOZE, S., IKITS, M., LEFOHN, A., AND HANSEN, C. 2003. Gaussian transfer functions for multi-field volume visualization. In *IEEE Visualization*, To Appear.

KRAUS, M., AND ERTL, T. 2002. Adaptive texture maps. In *Graphics Hardware 2002*, 7–16.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Transactions on Graphics*, vol. 22, 908–916.

LARSEN, E. S., AND MCALLISTER, D. 2001. Fast matrix multiplies using graphics hardware. In *Super Computing 2001*, ACM SIGARCH/IEEE.

LEFOHN, A., AND WHITAKER, R. 2002. A GPU-based, three-dimensional level set solver with curvature flow. University of Utah tech report UUCS-02-017, December.

LEFOHN, A., CATES, J., AND WHITAKER, R. 2003. Interactive, GPU-based level sets for 3D brain tumor segmentation. In *Medical Image Computing and Computer Assisted Intervention*, To Appear.

LEVOY, M. 1988. Display of surfaces from volume data. *IEEE Computer Graphics & Applications 8*, 29–37.

MALLADI, R., SETHIAN, J. A., AND VEMURI, B. C. 1995. Shape modeling with front propagation: A level set approach. *IEEE Trans. on Pattern Analysis and Machine Intelligence 17*, 158–175.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. In *ACM Transactions on Graphics*, vol. 22, 896–907.

OSHER, S., AND SETHIAN, J. 1988. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics 79*, 12–49.

OWENS, J. 2002. *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University.

PENG, D., MERRIMAN, B., OSHER, S., ZHAO, H., AND KANG, M. 1999. A PDE based fast local level set method. *Journal of Computational Physics 155*, 410–438.

PERCY, J., AND MACE, R. 2003. OpenGL extensions: Siggraph 2003. `http://mirror.ati.com/developer/techpapers.html`.

RUMPF, M., AND STRZODKA, R. 2001. Level set segmentation in graphics hardware. In *International Conference on Image Processing*, 1103–1106.

SABELLA, P. 1988. A rendering algorithm for visualizing 3D scalar fields. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, vol. 22, 51–58.

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

SHERBONDY, A., HOUSTON, M., AND NEPAL, S. 2003. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *IEEE Visualization*, To Appear.

STRZODKA, R., AND RUMPF, M. 2001. Using graphics cards for quantized FEM computations. In *Proceedings VIIP Conference on Visualization and Image Processing.*

TASDIZEN, T., WHITAKER, R., BURCHARD, P., AND OSHER, S. 2002. Geometric surface smoothing via anisotropic diffusion of normals. In *IEEE Visualization*, 125–132.

THE INSIGHT TOOLKIT. 2003. `http://www.itk.org`.

WHITAKER, R. T. 1994. Volumetric deformable models: Active blobs. In *Visualization In Biomedical Computing 1994*, SPIE, Mayo Clinic, Rochester, Minnesota, R. A. Robb, Ed., 122–134.

WHITAKER, R. 1998. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision October*, 203–231.

WILSON, O., GELDER, A. V., AND WILHELMS, J. 1994. Direct Volume Rendering via 3D Textures. Tech. Rep. UCSC-CRL-94-19, University of California at Santa Cruz, June.

YOO, T., NEUMANN, U., FUCHS, H., PIZER, S., CULLIP, T., RHOADES, J., AND WHITAKER, R. 1992. Direct visualization of volume data. *IEEE Computer Graphics and Applications 12*, 63–71.