Title
A Streaming Narrow-Band Algorithm: Interactive Deformation and Visualization of Level Sets

Journal
IEEE Transactions on Visualization and Computer Graphics, 10

Authors
Lefohn, Aaron
Kniss, Joe M.
Hansen, Charles D.

Publication Date
2004

Peer reviewed

A Streaming Narrow-Band Algorithm: Interactive Computation and Visualization of Level Sets

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

(Invited Paper)

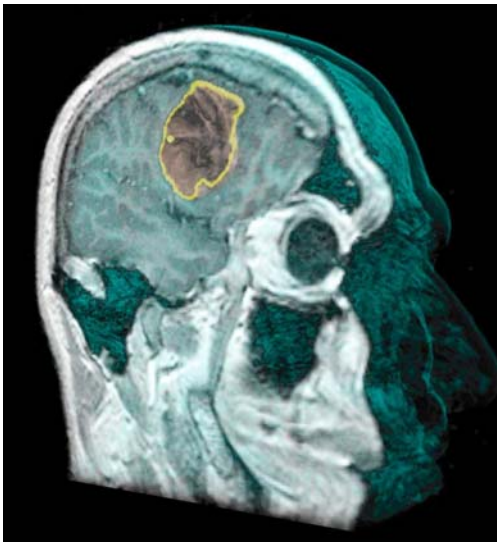

Fig. 1. Interactive level-set segmentation of a brain tumor from a $256 \times 256 \times 198$ MRI with volume rendering to give context to the segmented surface. A clipping plane shows the user the source data, the volume rendering, and the segmentation simultaneously. The segmentation and volume rendering parameters are set by the user probing data values on the clipping plane.

***Abstract*— Deformable isosurfaces, implemented with level-set methods, have demonstrated a great potential in visualization and computer graphics for applications such as segmentation, surface processing, and physically-based modeling. Their usefulness has been limited, however, by their high computational cost 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. The proposed solution is based on a new, streaming implementation of the narrow-band algorithm. The new algorithm packs the level-set isosurface data into 2D texture memory via a multi-dimensional virtual memory system. As the level-set moves, this texture-based representation is dynamically updated via a novel GPU-to-CPU message passing scheme. By integrating the level-set solver with a real-time volume renderer, a user can visualize and intuitively steer the level-set surface as it evolves. We demonstrate the capabilities of this technology for interactive volume segmentation and visualization.**

***Index Terms*— Deformable Models, Image Segmentation, Volume Visualization, GPU, Level Sets, Streaming Computation, Virtual Memory**

All authors are associated with the Scientific Computing and Imaging Institute at the University of Utah.
e-mail:{lefohn|jmk|hansen|whitaker}@sci.utah.edu

I. 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], surface processing [5], and surface reconstruction [6].

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. Setting these free parameters can be difficult 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 provides visual context for the evolving level set due to the global nature of the transfer function's opacity and color assignment. Also, the results of a level-set segmentation can specify a region-of-interest for the volume renderer [7].

The main contributions of this paper are:

- An integrated system demonstrating that 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 that is approximately 15 times faster than previous optimized solutions
- A multi-dimensional virtual memory scheme for GPU texture memory that supports computation on time-dependent, sparse data
- Real-time volume rendering directly from a packed, 2D texture format. The technique also enables volume rendering from a data set represented as a *single* set of 2D slices.
- A message passing scheme between the GPU and CPU that uses automatic mipmap generation to create compact, encoded messages
- Efficient computation of a volumetric distance transform on the GPU

II. Background and Related Work

A. 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 merely review the notation and describe the particular formulation that is relevant to this paper.

An implicit model represents a surface as the set of points $S = \{\bar{x} | \phi(\bar{x}) = 0\}$, where $\phi : \mathbb{R}^3 \mapsto \mathbb{R}$. 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 $\bar{v}$ describes the motion of the surface. Note that $\bar{v}$ can vary in both space and time. Within this framework one can implement a wide range of deformations by defining an appropriate $\bar{v}$. This velocity term is often a combination of several other terms, including data-dependent terms, geometric terms (e.g. curvature), and others. In many applications, these velocities introduce free parameters, and the proper tuning of those parameters is critical to making the level-set model behave in a desirable manner. Equation (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.

The proposed solver addresses the issues surrounding the solutions of (1). For this paper, however, we restrict the discussion on the particular form of this equation that is suitable for the segmentation application described in Sect. VI-A. This special case of (1) occurs when $\bar{v} = G(\bar{x}, \bar{t})\bar{n}$, where $\bar{n}$ is the surface normal and G is a scalar field, which we refer to as the *speed* of the level set. In this case (1) becomes

$$\partial\phi/\partial t = -|\nabla\phi|G. \tag{2}$$

Equation (2) describes a surface motion in the direction of the surface normal, and thus the volume enclosed by the surface expands or contracts, depending on the sign and magnitude of G.

Another important special case occurs when G, in (2), is the mean curvature of the level-set surface. The mean curvature of the level sets of ϕ are expressed as

$$H = \frac{1}{2}\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}. \tag{3}$$

In volume segmentation and surface reconstruction this mean curvature term is typically combined with an application-specific data term in order to obtain a smooth result that reflects interesting properties in the data.

There is a special case of (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* [3] and variations thereof [8]. However, this case covers only a very small subset of interesting speed functions. In general, we are concerned with solutions that allow the model to expand and contract as well as include a curvature term.

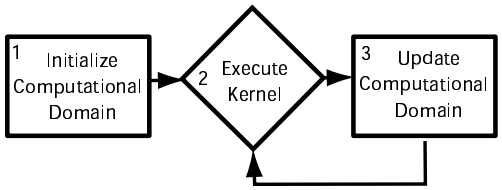

Fig. 2. The three fundamental steps in a sparse-grid solver. Step 1 initializes the sparse computational domain. Step 2 executes the computational kernel on each element in the domain. Step 3 updates the domain if necessary. Steps 2 and 3 are repeated for each solver iteration.

Efficient algorithms for solving the more general equation rely on the observation that at any one time step the only parts of the solution that are important are those adjacent to the moving surface (near points where $\phi = 0$). This observation places level-set solvers as part of a larger class of solvers that efficiently operate on time-dependent, sparse computational domains—i.e. a subset of the original problem domain (Figure 2).

Two of the most common CPU-based level-set solver techniques are the *narrow-band* [9] and *sparse-field* [6], [10] methods. Both approaches limit the computation to a narrow region near the isosurface yet store the complete computational domain in memory. The narrow-band approach implements the initialization and update steps in Figure 2 (Steps 1 and 3) by updating the embedding, ϕ, on a band of 10-20 pixels around the model, using a signed distance transform implemented with the fast marching method [3]. The band is reinitialized whenever the model (defined as a particular level set) approaches the edge. In contrast, the sparse-field method only traverses the complete domain during the initialization step of the algorithm in Figure 2. The sparse-field approach keeps a linked list of active data elements. The list is incrementally updated via a distance transform after each iteration. 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. Although the new solver borrows ideas from both the narrow-band and sparse-field algorithms, it implements a new solution that conforms to the architectural restrictions of GPUs.

B. 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 [11] 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 3 depicts the computation pipeline of a typical GPU. Vertices and texture coordinates are first processed by the vertex processor. The rasterizer then interpolates across the primitives defined by the vertices and generates *fragments* (i.e. pixels). The fragment processor applies textures and/or performs computations that determine the final pixel value. A *render pass* is a set of data passing completely through this pipeline. It can also be thought of as the complete processing of a stream by a given kernel (i.e. a *ForEach* call).

Grid-based computations are solved by first transferring the initial data into texture memory. The GPU performs the computation by rendering graphics primitives that access this texture. In the simplest case, a computation is performed on all elements of a 2D texture by drawing a quadrilateral that covers the same number of grid points (pixels) as the texture. Memory addresses that identify each fragment's data value as well as the location of its neighbors are given as texture coordinates. A fragment program (the kernel) then uses these addresses to read data from texture memory, perform the computation, and write the result back to texture memory. A 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, multi-grid solvers, and others [12]–[14]. All of these examples demonstrate a homogeneous sequence of operations over a densely populated grid structure.

Strzodka et al. [15] 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 [16]. 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 *time-dependent, sparse grid* problems. These problems are difficult to solve efficiently with GPUs for two reasons. The first is that in order to take advantage of the GPU's parallelism, the streams being processed must be large, contiguous blocks of data, and thus grid points near the level-set surface model must be *packed* into a small number of textures. The second difficulty is that the level set moves with each time step, and thus the packed representation must readily adapt to the changing position of the model. This requirement is in contrast to the recent sparse matrix solvers [17], [18] and previous work on rendering with compressed data [19], [20]. Recent work by Sherbondy et al. [21] describes an alternative time-dependent, sparse GPU computation model which is discussed in Section VI-C.

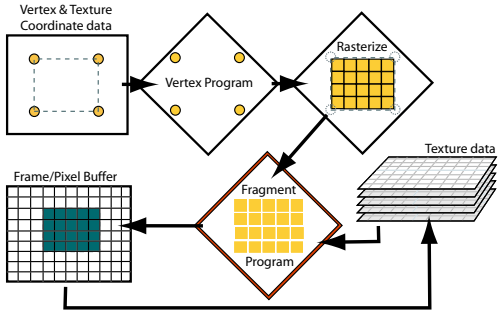

Fig. 3. The modern graphics processor pipeline.

C. Hardware-Accelerated Volume Rendering

Volume rendering is a flexible and efficient technique for creating images from 3D data [22]–[24]. 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 [25], [26]. In recent years, graphics hardware has become more programmable, permitting rendering features with an image quality that rival sophisticated software techniques [27], [28]. In this paper, we describe a novel volume rendering system that leverages programmable graphics hardware to render the packed level-set solution data.

III. A Virtual Memory Address Scheme for Sparse Computation

The limited computational capabilities of modern GPUs, their data-parallel streaming architecture, and our goal of interactive performance impose some important design restrictions on the proposed solver. For instance, the data-parallel computation model requires *homogeneous operations* on the entire computational domain, and memory constraints require us to process *and store* only the active domain on the computational processor (i.e. the GPU). Furthermore, GPUs do not support *scatter* write operations, and the communication bandwidth between the GPU and CPU is insufficient to allow transmission of any significant portion of the computational domain. Our new streaming, narrow-band level-set solver works efficiently within these restrictions and leverages GPU capabilities by packing the active computational domain into 2D texture memory. The GPU solves the 3D, level-set PDE directly on this packed format and quickly updates the packed representation after each solver iteration.

Re-mapping the computational domain (a subset of a volume) to take advantage of the GPU's capabilities has the unfortunate effect of making the computational kernels extremely complicated—that is difficult to design, debug, and modify. The kernel programmer must take the physical memory layout into consideration each time the kernel addresses memory.

Other researchers have successfully re-mapped computational domains to efficiently leverage the GPU's capabilities [12], [17], [18], [29], but they invariably describe these complex kernels in terms of the physical memory layout. This section presents a solution to this problem for level-set computation that allows kernels to access memory as if it were stored in the original, 3D domain—irrespective of the 2D physical layout used on the GPU. Our solution is an extension to the virtual memory systems used in modern operating systems.

A. Traditional Virtual Memory Overview

Nearly all modern operating systems contain a virtual memory system [30]. The purpose of virtual memory is to give the programmer the illusion that the application has access to a contiguous memory address space, while allowing the operating system to allocate memory for each process on demand, in manageable increments, from whatever physical resources happen to be available. Note that there are two meanings of virtual memory. The first is the mapping from a logical address space to a physical address space. The second is the mechanism for mapping logical memory onto a physical memory hierarchy (e.g. main memory, disk, etc). For this discussion, virtual memory only refers to the former definition.

Virtual memory works by adding a level of indirection between physical memory and the memory accessed by an application. Most conventional virtual memory systems divide physical and virtual memory into equally sized *pages*. The data addressed by an application's contiguous virtual address space will often be stored in many, disconnected physical memory pages. A *page table* tracks the mapping from virtual to physical memory pages. When an application requests memory, the system allocates physical memory pages and updates the page table. Note that the virtual and physical pages are identically sized.

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

$$\begin{aligned} \mathrm{VPN} &\leftarrow \tfrac{\mathrm{VA}}{S[\mathrm{P}]} \\ \mathrm{PPA} &\leftarrow \mathrm{PageTable}(\mathrm{VPN}) \\ \mathrm{OFF} &\leftarrow \mathrm{mod}(\mathrm{VA}, \mathrm{S[P]}) \\ \mathrm{PA} &\leftarrow \mathrm{PPA} + \mathrm{OFF}, \end{aligned} \tag{4}$$

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

B. Multi-Dimensional Virtual Memory for GPUs

The virtual memory system used in our solver is a multi-dimensional extension of the traditional virtual memory system described in Section III-A.

Traditional virtual memory systems use one-dimensional virtual and physical address spaces. Our system uses a 3D virtual and a 2D physical memory address space. We use a 3D virtual memory space because the level-set computation is inherently volumetric. The 2D physical memory address space is motivated by the fact that GPUs are optimized to process 2D memory regions. By using a 2D physical address space, we are able to process the *entire* active volumetric domain simultaneously. This maximizes the benefit of the parallel, SIMD architecture of the GPU. We also make the simplifying assumption that virtual and physical pages are identical in dimension and size. Thus, the virtual space is not partitioned equally in all axes: 2D pages must be stacked in 3D to populate the problem domain as seen in Figure 4. Our system uses pages of size $S[\mathrm{P}] = (16, 16)$. This size represents a good compromise between a tight fit to the narrow computational domain and the overhead of managing and computing pages. Empirical results validate this choice.

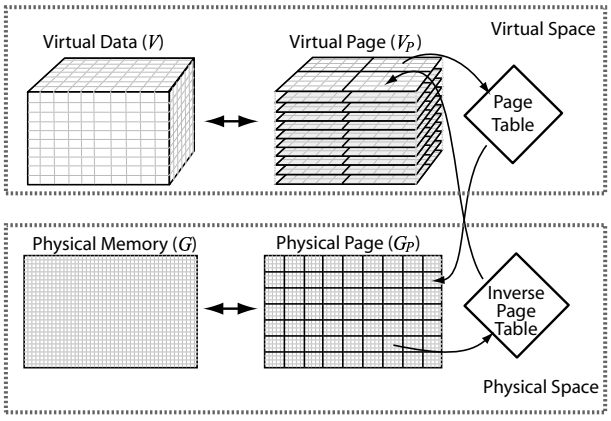

Fig. 4. The multi-dimensional virtual and physical memory spaces used in our virtual memory system. The original problem space is V, the virtual address space. The virtual page space, $\mathrm{V_P}$, is a subdivided version of V. Virtual memory pages are mapped to the physical page space, $\mathrm{G_P}$, by the *page table*. The *inverse page table* maps physical pages in $\mathrm{G_P}$ to virtual pages in $\mathrm{V_P}$. The collection of all elements in $\mathrm{G_P}$ constitute G, the physical memory of the hardware.

We now introduce notation for the various address spaces in our system. We notate the space of K-length vectors of integers as $\mathbb{Z}^K$. The set of all voxels in the 3D virtual address space (i.e. the problem domain) is defined as $\mathrm{V} \subset \mathbb{Z}^3$. Each of the virtual memory pages is a set of contiguous voxels in V; the space of all virtual pages is $\mathrm{V_P}$ (Figure 4). Similarly, the physical address space, $\mathrm{G} \subset \mathbb{Z}^2$, is subdivided into pages to form the physical page space, $\mathrm{G_P}$. The elements within a virtual or physical page are addressed identically using elements of $\mathrm{P} \subset \mathbb{Z}^2$. We also define a size operator for the 2D and 3D spaces described above. For X in $\{\mathrm{V}, \mathrm{V_P}, \mathrm{G}, \mathrm{G_P}, \mathrm{P}\}$, we define $S[X]$ to be a 2-vector or 3-vector (according to the dimension of X) giving the number of elements along each axis of the space X. Note that $S[\mathrm{V_P}] = \mathrm{S[V]}/\mathrm{S[P]}$ and $S[\mathrm{G_P}] = \mathrm{S[G]}/\mathrm{S[P]}$ (using component-wise division).

Virtual-to-physical address translation in a multi-dimensional virtual memory system works analogously to the 1D algorithm. Virtual addresses are now 3D position vectors in V and physical addresses are 2D vectors in G. The page table is a 3D table that returns 2D physical page

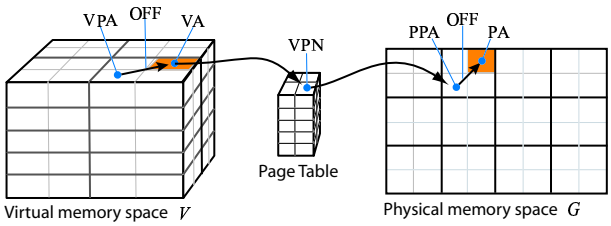

Fig. 5. The virtual-to-physical address translation scheme in our multi-dimensional virtual memory system. A 3D virtual address, VA, is first translated to a virtual page number, VPN. A page table translates the VPN to a physical page address, PPA. The PPA specifies the origin of the physical page containing the physical address, PA. The offset is then computed based on the virtual address and used to obtain the final 2D physical address, PA.

addresses. With these multi-dimensional definitions in mind, Eq (4) still applies to the vector-valued quantities. Figure 5 shows an example multi-dimensional address translation.

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

C. Virtual-to-Physical Address Translation

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

The simplest and most general way to implement the virtual-to-physical address translation for a GPU-based virtual memory system is to directly implement the computation in (4) and store the page table on the GPU as a 3D texture. A significant benefit of this approach is that it is completely general. Unfortunately, without dedicated memory-management hardware to accelerate the translation, this scheme suffers from several efficiency problems. First, the page table lookup means that a *dependent* texture read is required for each memory access. A dependent texture is defined as using the result of one texture lookup to index into another. This may cause a significant loss in performance on current GPUs. Second, storing the page table on the GPU consumes limited texture memory. The third problem is that a divide, modulus, and addition operation are required for each memory access. This consumes costly and limited fragment program instructions. Note that Section III-D discusses other problems with storing the page table on the GPU related to the limited capabilities of current GPU architectures.

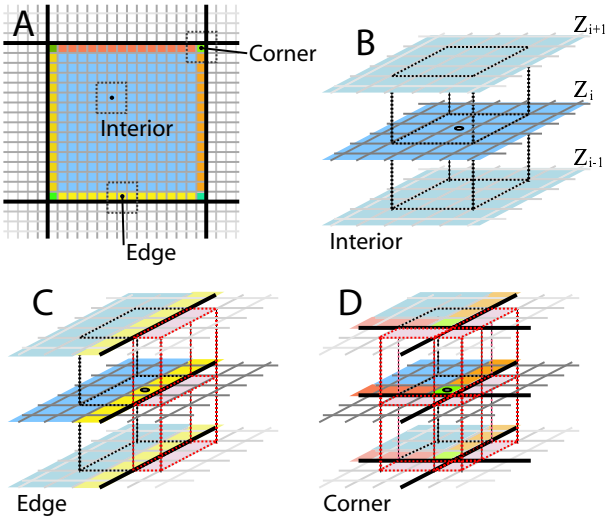

Fig. 6. The *substream* boundary cases used to statically resolve the conditionals arising from $3 \times 3 \times 3$ neighbor accesses across memory page boundaries. The nine *substream* cases are: interior, left edge, right edge, top edge, bottom edge, lower-left corner, lower-right corner, upper-right corner, and upper-left corner (a). The interior case accesses its neighbors from only three memory pages (b). The edge cases require six pages (c), and the corner cases require twelve memory pages (d). Note that for reasonably large page sizes, the more cache-friendly interior case has by far the highest number of data elements.

We can avoid the memory and computational inefficiencies that arise from storing the page table on the GPU by examining the pattern of virtual addresses required by the application's fragment program. In the case of our level-set solver, the fragment programs only use virtual addresses within a $3 \times 3 \times 3$ neighborhood of each active data element. This means that each active memory page will only access adjacent virtual memory pages (Figure 6). Moreover, we show that this simplified translation case makes it possible to lift the entire address translation from the fragment processor to the vertex processor and rasterizer.

Once we resolve the virtual addresses used by a fragment program, we can determine which virtual pages each active page will access. With this *relative page* information, the GPU can perform the virtual-to-physical address translation without a page table in texture memory. The CPU makes this possible by sending the PPAs for all required pages to the GPU as texture coordinates. The GPU can then use the relative neighbor offset vectors to decide which adjacent page contains the requested value (see Figure 6(a)).

The GPU's task of deciding which adjacent page contains a specific neighbor value unfortunately requires a significant amount of conditional logic. This logic must classify each data element into one of nine boundary cases: one of the four corners, one of the four edges, or an interior element (see Figure 6). Unfortunately current fragment processors do not support conditional execution. This logic could alternatively be encoded into a texture; however, this would again force the use of an expensive dependent texture read. Just as statically resolving virtual addresses allowed us to optimize the GPU computation, all active data elements can be pre-classified into the nine boundary cases. The result is that all

memory addresses used in each case will lie on the same pages relative to each active page (see Figure 6). In other words, the memory-page-locating logic has been statically resolved by pre-classifying data elements into their respective boundary cases. The data elements for these *substream* cases are generated by drawing unique geometry for each case. The corner substream cases are represented as points, the edges as lines, and the interior regions as quadrilaterals.

Kapasi et al. [31] describe an efficient solution to conditional execution in streaming architectures. Their solution is to route stream elements to different processing elements based on the code branch. Substreams are merely a static implementation of this data routing solution to conditional execution. The advantage is that the computation kernel run on each substream contains no conditional logic and is optimized specifically for that case. Our solution additionally gains from optimized cache behavior for the most common, interior, case (77% of the data points in a 16×16 page). The interior data elements require only three memory pages to access all neighbors (Figure 6(b)). In comparison, reading all neighbors for an edge element requires loading six pages (Figure 6(c)). The corner cases require twelve pages from disparate regions of physical memory(Figure 6(d)). The corner cases account for less than 2% of the active data elements.

With the use of substreams, the GPU can additionally optimize the address computation by computing physical addresses with the vertex processor rather than the fragment processor. Because all data elements (i.e. fragments) use exactly the same relative memory addresses, the offset and physical address computation steps of (4) can be generated by interpolating between substream vertex locations. The vertex processor and rasterizer can thus perform the entire address translation. This optimization distributes computational load to under-utilized processing units and reduces the number of limited and expensive fragment instructions.

D. Bootstrapping the Virtual Memory System

This section describes the steps required to initialize the GPU virtual memory system. To begin, the application specifies the page size, S[P], the virtual page space size, $\mathrm{S}[\mathrm{V_P}]$, and the fundamental data type to use (i.e. 32-bit floating point, 16-bit fixed point, etc.). The virtual memory system then allocates an initial physical memory buffer on the GPU. It also creates a page table, an inverse page table, a geometry engine, and a stack of free pages on the CPU. The decision to place the aforementioned data structures on the CPU is based on the efficiency concerns described in Section III-C as well as GPU architectural restrictions. These restrictions include: the GPU's lack of random write access to memory, lack of writable 3D textures, lack of dynamically sized output buffers, and limited GPU memory.

The page table is defined to store a `MemoryPage` object that contains the vertices and texture coordinates required by the GPU to access the physical memory page. The inverse page table is designed to store a VPN vector for each active physical page. Figure 5 shows these mappings. Note that the page table and inverse page table were referred to as the *unpacked map* and *packed map* respectively in Lefohn et al. [32].

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

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

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

IV. Sparse GPU Level-Set Solver

This section now explains our GPU level-set solver implementation using the virtual memory system and level-set equations presented in Section III and Section II-A. Note that the details of the level-set discretization are found in Lefohn et al. [33].

A. Initialization of Computational Domain

The solver begins by initializing the sparse computational domain (Step 1 in Figure 2). An initial level-set volume is passed to the level-set solver by the host application. The sparse domain initialization involves identifying active memory pages in the input volume, allocating GPU memory for each active page, then sending the initial data to the GPU.

The solver identifies active virtual pages by checking each data element for a non-zero derivative value in any of the six cardinal directions. If any element in a page contains non-zero derivatives, the entire page is activated. The initialization code then requests a GPU memory page from the virtual memory system for each active page. The level-set data is then *drawn* into GPU memory using the vertex locations in each `MemoryPage` object.

This scheme is effective only because the input level-set volume is assumed to be a clamped distance transform– meaning that regions on or near the isosurface have non-zero gradients while regions outside or inside the surface have gradients of zero. The outside voxels have a value of zero (black) and the inside ones have a value of one (white). Section IV-B explains how the distance transform embedding is maintained throughout the level-set computation.

The inactive virtual pages do not need to be represented in physical memory. If an active data element queries an inactive value, however, an appropriate value needs to be returned. Because all inactive regions are either uniformly black or

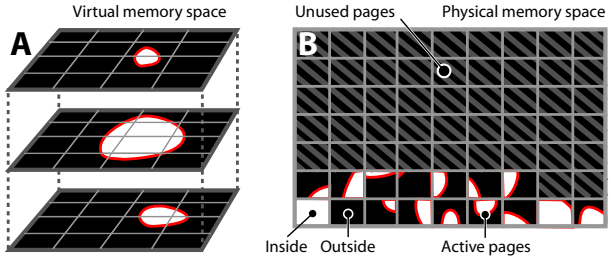

Fig. 7. The level-set solver's use of the paged virtual memory system. All *active* pages (i.e. those that contain non-zero derivatives) in the virtual page space (a) are mapped to unique pages of physical memory (b). The inactive virtual pages are mapped to the static *inside* or *outside* physical page. Note that the only data stored on the GPU is that represented by (b).

white, we solve this boundary condition problem by defining a special, inactive page state. A virtual page in this state is mapped to one of two *static* physical pages. One of these static pages is black, representing regions outside of the level-set surface. The other static page is white and represents regions inside the level-set surface. The page table contains these many-to-one mappings, but the inverse page table does not store a valid entry for the static pages. Note that we could have alternatively solved this boundary problem using single pixels instead of entire pages. We also could have solved the problem by creating substreams for the active elements on the boundary of the active set.

B. Distance Transform on the GPU

In order to take advantage of the sparse nature of level-set solutions, algorithms must maintain a somewhat consistent *level-set density*, which is defined as the number of level sets per unit volume. If the level-set density becomes too low (spread out) it can become difficult to efficiently isolate the computation to the desired interface. Alternatively, a level-set density that becomes too high (close together) can cause aliasing and numerical problems. The most common way of maintaining a desired level-set density is to keep the embedding, ϕ, resembling a distance transform [6], [9], [34].

The new streaming level-set solver maintains the distance transform by introducing an additional speed term, G_r, to the level-set PDE (1) that controls the surface motion. This speed term *pushes* the level sets of ϕ, either closer together or farther apart, so that they resemble a clamped distance transform (CDT). The CDT has a constant level-set density within a predefined band and ensures that voxels near the isosurface have finite derivatives while those farther away have gradient magnitudes of zero. As described in Sections IV-A and IV-D, the identification of zero-derivative regions is critical for an efficient solver implementation. This *rescaling* speed term, G_r, is computed as

$$G_r = \phi g_\phi - \phi|\nabla\phi|, \tag{5}$$

where g_ϕ is the target gradient magnitude within the computational domain, and $|\nabla\phi|$ is the gradient magnitude in the direction of the level-set model isosurface. The target parameter, g_ϕ, can be set based on the numerical precision of the level-set data. By setting g_ϕ sufficiently high, numerical errors caused by underflow can easily be avoided. It is important to note that G_r is strictly a numerical construct; it does not affect the movement of the zero level set, i.e. the surface model. Also note that the solver can be used to compute *only* the distance transform (i.e. no surface movement) by setting g_ϕ to one and making G_r the only speed term.

C. Level-Set Computation

The GPU next performs the level-set computation (Step 2 of the sparse algorithm in Figure 2). The details of the level-set discretization used by our solver are given in Lefohn et al. [33]. This section gives a high-level overview of the computation. The level-set update proceeds in the following steps:

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

The derivative computation in Step A above uses the substream-based, virtual-to-physical address scheme described in Section III-C. The derivatives are computed in nine substream render passes, each of which outputs to the same four, 4-tuple buffers. The speed function computations in Step B are application-dependent. Example speed terms include the curvature computation described in (3), the rescaling term described in (5), and the thresholding term described in (7). There will be zero or more render passes for each speed function. The level-set update (Step C) is the up-wind scheme described in Lefohn et al. [33]. This is computed in a single pass. Note that additional GPU memory must be allocated to store the intermediate results accumulated in Steps A and B before they are consumed in Step C. Our solver performs register allocation of temporary buffers to minimize GPU memory usage.

D. Update of Computational Domain

After each level-set update, the solver determines which virtual pages need to be added-to or removed-from the active domain. The solver accomplishes this by aggregating gradient information from all elements in each active page. In our solver, the GPU must compute this information because the level-set solution exists only in physical memory. The active set must be updated by the CPU, however, because the page table and geometry engine exist in CPU main memory. In addition, the amount of information passed from the GPU to the CPU must be kept to a minimum because of the limited bandwidth between the two processors. This section gives an overview of an algorithm that works within these constraints. Lefohn et al. [33] explains the full details of the algorithm.

The GPU creates a memory allocation/deallocation request by producing a small image (of size $S[\mathrm{G_P}]$) with a single-byte pixel per physical page. The value of each pixel is a bit code that encapsulates the activation or deactivation state of each page and its six adjacent neighbors (in $\mathrm{V_P}$). The CPU reads this small ($< 64\mathrm{kB}$) message, decodes it, and submits the allocation/deallocation requests to the virtual memory system (Figure 8).

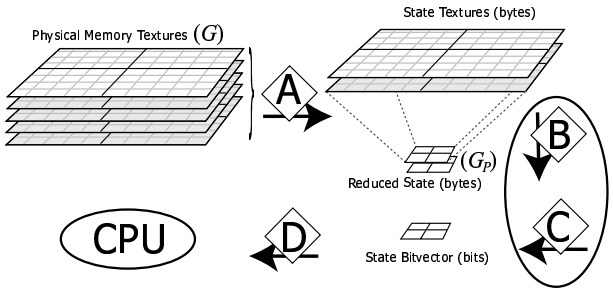

Fig. 8. The GPU's creation of a memory allocation/deallocation request. Step A uses solver-specific data to create two buffers containing the active state of each data element and its adjacent neighbors. Step B uses automatic mipmapping to reduce the buffers from size $S[\mathrm{G}]$ to the physical page space size, $S[\mathrm{G_P}]$. Step C combines the information from the two down-sampled state buffers into an eight-bit code for each pixel. This code encapsulates whether or not each active virtual memory page and its adjacent neighbors should be enabled. In step D, the CPU reads the bit-code buffer, decodes it, and allocates/deallocates pages as requested.

The GPU creates the bit-code image by first computing two, four-component neighbor information buffers of size $S[\mathrm{G}]$ (Step A of Figure 8). This computation uses the previously-computed, one-sided derivatives of ϕ to identify the required active pages. A page must be activated if it contains elements with non-zero gradient magnitudes. The automatic mipmapping GPU feature is then used to down-sample the resulting buffers (i.e. aggregate data samples) to the page-space image (Step B in Figure 8). The final GPU operation combines the active page information into the bit code (Step C in Figure 8). A fragment program performs this step by emulating a bit-wise OR operation via conditional addition of powers of two. Finally, in step D of Figure 8, the CPU reads this message from the GPU.

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

E. GPU Implementation Details

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

There are several details related to render pass output buffers that are critical to the performance of the level-set solver. First is the ability to output multiple, high-precision 4-tuple results from a fragment program. Writing sixteen scalar outputs from a single render pass enables us to perform the expensive 3D neighborhood reconstruction only once and use the gathered data to compute the derivatives in a single pass. Second, we avoid the expensive change between render targets [35] (i.e. pixel buffers) by allocating a single pixel buffer with many *render surfaces* (front, back, aux0, etc.) and using each surface as a separate output buffer.

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

V. Volume Rendering of Packed Data

The direct visualization of the level-set evolution is important for a variety of level-set applications. For instance, in the context of segmentation, direct visualization allows a user to immediately assess the quality and accuracy of the pending segmentation and steer the evolution toward the desired result. Volume rendering is a natural choice for visualizing the level-set surface model, because it does not require an intermediate geometric extraction, which would severely limit interactivity. If one were to use marching cubes, for instance, a distinct triangle mesh would need to be created (and rendered) for each iteration of the level-set solver. The proposed solver, therefore, includes a volume renderer, which produces a full 3D (transfer-function based) volume rendering of the evolving level set on the GPU [28].

For rendering the evolving level-set model, we use a variant of traditional 2D texture based volume rendering [25]. We modify the conventional approach to render the level-set solution directly from the packed physical memory layout, which is physically stored in a single 2D texture. Because the level-set data and physical page configuration are dynamic, it would be inefficient to pre-compute and store 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. This new technique is thus both applicable to rendering compressed data as well as traditional texture-based volume rendering from a single set of 2D slices.

The volume rendering algorithm utilizes a two pass approach for reconstruction and rendering. Figure 9 illustrates the steps involved. An additional off-screen buffer caches two reconstructed neighboring slices containing the level-set solution and its gradient (Figure 9 A). During the rendering phase arbitrary slices along the preferred slice direction are interpolated from these neighboring slices (Figure 9 B). Once all interpolated slices between slice i and $i-1$ are rendered and composited, the next slice $(i+1)$ is reconstructed. This newly reconstructed slice replaces the cached slice, $i-1$. The GPU then renders and composites the next set of interpolated slices (i.e. those between slice $i+1$ and i). This pattern continues until all slices have been reconstructed and rendered.

When the preferred slice axis, based on the viewing angle, is orthogonal to the virtual memory page layout, we reconstruct 2D slices of the level-set solution and its gradient using a

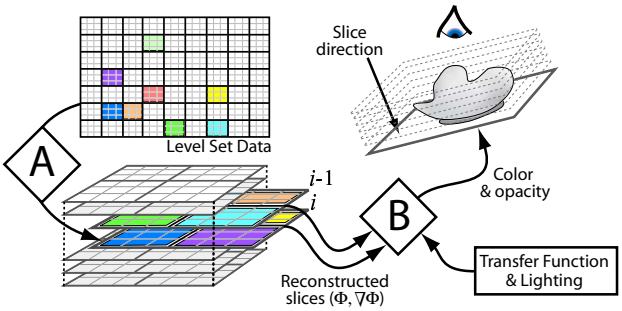

Fig. 9. Two pass rendering of packed volume data. In step A, a 2D slice (i) is reconstructed from the physical page (packed) layout, G_P. In step B, one or more intermediate slices between i and $i-1$ are interpolated, transformed into optical properties (via the transfer function), lit, and rendered for the current view. The next iteration begins by reconstructing slice $i+1$, replacing $i-1$, and so on.

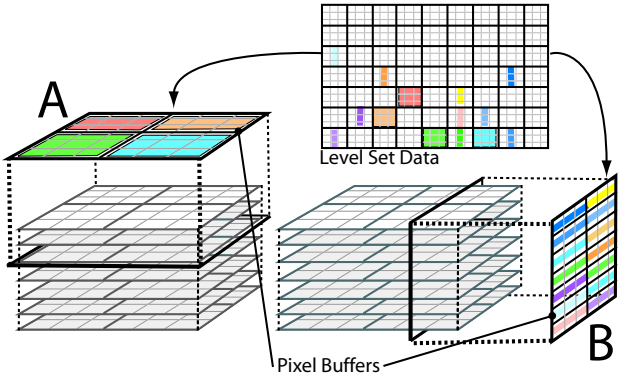

Fig. 10. Reconstruction of a slice for volume rendering the packed level-set model: (a) When the preferred slicing direction is orthogonal to the virtual memory page layout, the pages (shown in alternating colors) are draw into a pixel buffer as quadrilaterals. (b) For slicing directions parallel to the virtual page layout, the pages are drawn onto a pixel buffer as either vertical or horizontal lines.

textured quadrilateral for each page, as shown in Fig. 10 A. On the other hand, if the preferred slice direction is parallel to the virtual page layout, we render a row or column from each page using textured line primitives, as in Fig. 10 B. In both cases, slices are reconstructed into a pixel buffer which is bound as a texture in the rendering pass. These slices are reconstructed at the same resolution as level-set solution.

In the rendering phase, we leverage the hardware's bilinear filtering for in-plane interpolation of the reconstructed level-set slice. Trilinear interpolation of an arbitrary slice between two adjacent reconstructed slices is accomplished by combining them, *i.e.* performing linear interpolation along the preferred slice direction, in the fragment program. This same fragment program also evaluates the transfer function and lighting for the interpolated data. For efficiency, we also reuse data wherever possible. For instance, lighting for the level-set surface, evaluated in the rendering phase, uses gradient vectors computed during the level-set update stage.

VI. Application and Results

This section describes an application for interactive volume segmentation and visualization, which uses the level-set solver and volume renderer described previously. 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 [36].

A. Volume Segmentation With Level-Sets

For segmenting volume data with level sets, the speed functions usually consists of a combination of two terms [4], [37]

$$\frac{\partial \phi}{\partial t} = |\nabla \phi| \left[\alpha D(\bar{x}) + (1-\alpha) \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} \right], \qquad (6)$$

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

This combination of a data-fitting speed function with the curvature term is critical to the application of level sets to volume segmentation. Most level-set data terms D from the segmentation literature are equivalent to well-known algorithms such as isosurfaces, flood fill, or edge detection when used without the smoothing term (i.e. $\alpha = 1$). The smoothing term alleviates the effects of noise and small imperfections in the data, and can prevent the model from leaking into unwanted areas. 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 greyscale value input data I at the point $\bar{x}$:

$$D(I) = \epsilon - |I - T|, \qquad (7)$$

where T controls the brightness of the region to be segmented and ϵ controls the range of greyscale values around T that could be considered inside the object. In this way a model situated on voxels with greyscale values in the interval $T \pm \epsilon$ will expand to enclose that voxel, whereas a model situated on greyscale values outside that interval will contract to exclude that voxel. The speed term is gradual, as shown in Fig. 11, and thus the effects of D diminish as the model approaches the boundaries of regions with greyscale levels within the $T \pm \epsilon$ range. This makes the effects of the curvature term relatively larger. This choice of D corresponds to a simple, one-dimensional statistical classifier on the volume intensity [38].

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 pre-processed (thresholded, flood filled, etc.) version of the source data.

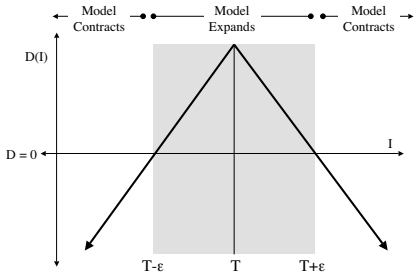

Fig. 11. A speed function based on image intensity causes the model to expand over regions with greyscale values within the specified (positive) range and contract otherwise.

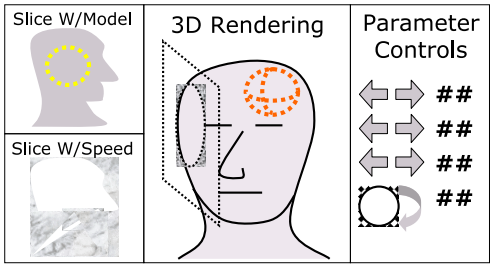

Fig. 12. A depiction of the user interface for the volume analysis application. Users interact via slice views, a 3D rendering, and a control panel.

B. 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. 12). 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 tasks 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 add-to or subtract-from the model by drawing white or black spheres, respectively. This feature gives the user "3D paint" and "3D eraser" tools with which to interactively edit the level-set solution.

The volume renderer displays a 3D reconstruction of the current level set isosurface (see Section V) as well as the input data. In addition, an arbitrary clipping plane, with texture-mapped source data, can be enabled via the GUI (Figure 1). 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 [39].

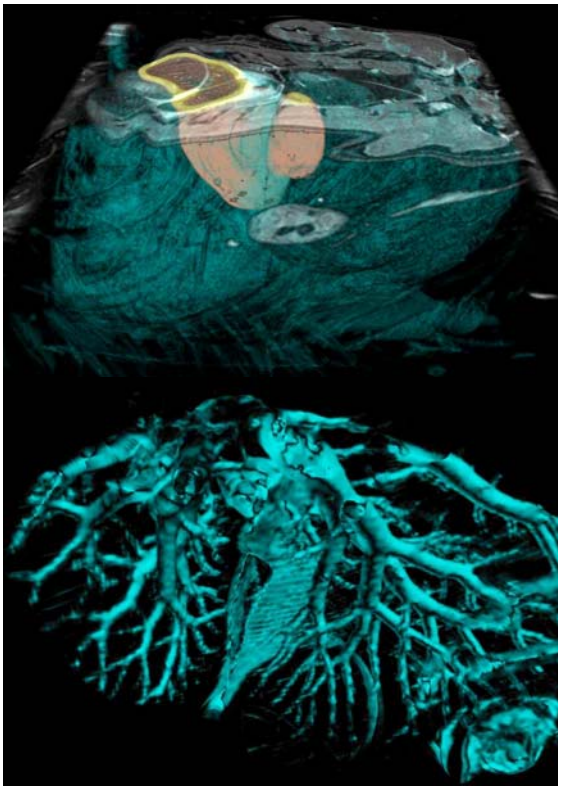

Fig. 13. (top) Volume rendering of a 256^3 MRI scan of a mouse thorax. Note the level-set surface which is deformed to segment the liver. (bottom) Volume rendering of the vasculature inside the liver using the same transfer function as in (top) with the level-set surface is being used as a region-of-interest specifier.

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. 13) and transmission electron tomography data of a gap junction (Fig. 14). In all of these examples a user interactively controls the level-set surface evolution and volume rendering via the multi-view interface. The initializations for the tumor and mouse were drawn via the user interface. The initialization for Figure 14 was seeded with a thresholded version of the source data.

C. 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 [36]. All benchmarks were run on an Intel Xeon 1.7 GHz processor with 1 GB of RAM and an

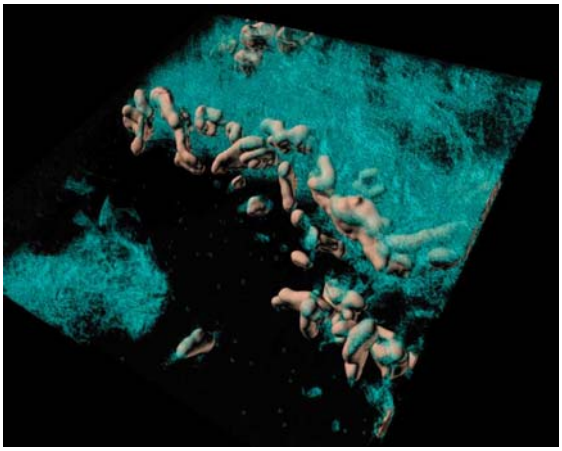

Fig. 14. Segmentation and volume rendering of $512 \times 512 \times 61$ 3D transmission electron tomography data. The picture shows cytoskeletal membrane extensions and connexins (pink surfaces extracted with the level-set models) near the gap junction between two cells (volume rendered in cyan).

ATI Radeon 9800 Pro GPU. All timings include the complete computation, i.e. both the virtual memory system update and the level-set computation are included. For a $256 \times 256 \times 175$ volume, the level-set solver runs at rates varying from 70 steps per second for the tumor segmentation (Fig. 1) to 3.5 steps per second for the final stages of the cortex segmentation from the same data set. 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.

The amount of texture memory required for the level-set computation is proportional to the surface area of the level-set surface—i.e. the number of active pages. Our tests have shown that for many applications, only 10%-30% of the volume is active. To take full advantage of this savings, the total size of physical memory, $S[\mathrm{G}]$, must increase when the number of allocated pages grows beyond the physical memory's capacity. Our current implementation performs only static allocation of the maximum physical memory space, but future versions could easily realize the above memory savings. Section VII 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. [21], 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 significant portion of the execution time. In addition, the 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 packed tiles, and more experimentation is necessary to understand the circumstances under which each approach is advantageous. Furthermore, they are not mutually exclusive, and Sect. VII discusses the possibility of using depth culling in combination with our packed representation.

As with any sparse algorithm, it will be advantageous to simply compute the entire (original) domain if the active domain becomes sufficiently large. Our experience with segmentation thus far, however, has shown that the the computation remains sufficiently sparse even for large structures such as a cerebral cortex segmentation. The sparseness is due to the fact that only the surface needs to represented, and the interior regions need not be represented or computed.

VII. Conclusions and Future Work

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 an efficient multi-dimensional virtual memory system to implement a time-dependent, sparse computation scheme. The memory mappings are updated via a novel GPU-to-CPU message passing algorithm. The GPU renders the level-set surface model directly from a sparse, compressed 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.

There are a couple ways in which the memory and computational efficiency of our solver can be improved. First, it may be worth achieving an even narrower band of computation around the level-set model. This is possible by using depth culling to avoid computation on inactive elements within each active page [21]. Implementing this depth culling requires a memory model in which an arbitrary number of data buffers can access a single depth buffer. The second optimization is to allow the total amount of physical memory to change at run time and grow to the limits of GPU memory. This requires spreading physical memory across multiple 2D textures (i.e. creating a 3D physical memory space). The proposed *super buffer* [40] OpenGL extension supports both of these proposed optimizations.

The GPU virtual memory abstraction also indicate promising future research. We are currently beginning work on a more general virtual memory implementation that fully abstracts N-dimensional GPU memory. The goal is to provide an API that allows a GPU application programmer to specify an optimal physical and virtual memory layout for their problem, then write the computational kernels irrespective of the physical layout. The kernels will specify memory accesses via abstract memory access interfaces, and an operating-system-like layer will replace these memory access calls with the appropriate address translation code.

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 data set manipulation (`http://teem.sourceforge.net`). Milan Ikits' *GLEW* library was used for OpenGL extension management (`http://glew.sourceforge.net`). 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

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

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

[3] 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.

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

[6] R. Whitaker, "A level-set approach to 3D reconstruction from range data," *International Journal of Computer Vision*, vol. October, pp. 203–231, 1998.

[7] 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, pp. 63–71, 1992.

[8] 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.

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

[10] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang, "A PDE based fast local level set method," *Journal of Computational Physics*, vol. 155, pp. 410–438, 1999.

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

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

[13] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Super Computing 2001*, ACM SIGARCH/IEEE, Nov. 2001.

[14] R. Strzodka and M. Rumpf, "Using graphics cards for quantized FEM computations," in *Proceedings VIIP Conference on Visualization and Image Processing*, 2001.

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

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

[17] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," in *ACM Transactions on Graphics*, vol. 22, pp. 917–924, July 2003.

[18] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *ACM Transactions on Graphics*, vol. 22, pp. 908–916, July 2003.

[19] A. C. Beers, M. Agrawala, and N. Chaddha, "Rendering from compressed textures," in *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pp. 373–378, Aug. 1996.

[20] M. Kraus and T. Ertl, "Adaptive texture maps," in *Graphics Hardware 2002*, pp. 7–16, Sept. 2002.

[21] A. Sherbondy, M. Houston, and S. Nepal, "Fast volume segmentation with simultaneous visualization using programmable graphics hardware," in *IEEE Visualization*, pp. 171–176, October 2003.

[22] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *Computer Graphics (Proceedings of SIGGRAPH 88)*, vol. 22, pp. 65–74, Aug. 1988.

[23] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics & Applications*, vol. 8, pp. 29–37, 1988.

[24] P. Sabella, "A rendering algorithm for visualizing 3D scalar fields," in *Computer Graphics (Proceedings of SIGGRAPH 88)*, vol. 22, pp. 51–58, Aug. 1988.

[25] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *ACM Symposium On Volume Visualization*, pp. 91–98, Oct. 1994.

[26] 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.

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

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

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

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

[31] U. Kapasi, W. Dally, S. Rixner, P. Mattson, J. Owens, and B. Khailany, "Efficient conditional operations for data-parallel architectures," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pp. 159–170, 2000.

[32] A. E. Lefohn, J. Kniss, C. Hansen, and R. Whitaker, "Interactive deformation and visualization of level set surfaces using graphics hardware," in *IEEE Visualization*, pp. 75–82, October 2003.

[33] A. E. Lefohn, J. Kniss, C. Hansen, and R. Whitaker, "A streaming narrow-band algorithm: Supplemental information." IEEE Digital Library.

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

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

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

[37] R. Malladi, J. A. Sethian, and B. C. Vemuri, "Shape modeling with front propagation: A level set approach," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 17, pp. 158–175, 1995.

[38] A. E. Lefohn, J. Cates, and R. Whitaker, "Interactive, GPU-based level sets for 3D brain tumor segmentation," in *Medical Image Computing and Computer Assisted Intervention*, pp. 564–572, 2003.

[39] J. Kniss, S. Premoze, M. Ikits, A. E. Lefohn, and C. Hansen, "Gaussian transfer functions for multi-field volume visualization," in *IEEE Visualization*, pp. 497–504, October 2003.

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

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

APPENDIX A: DISCRETIZATION OF LEVEL-SET EQUATIONS

This appendix describes the discretization of equation 1 and the curvature computation 3. We discretize equation 1 using the *up-wind* scheme [1] and compute the curvature of the level-set surface using the *difference of normals* method [41].

We begin by describing the finite difference derivatives required for the level-set update and curvature computation. The neighborhood, u, from which these derivatives are computed is specified with the numbering scheme

6	7	8
3	4	5
0	1	2

. (8)

Note that u_4 denotes the center pixel, and $u_i^{\pm z}$ represents the i^{th} sample on the slice above or below the current one. The derivatives of the level-set embedding, ϕ, are then defined as

$$\begin{array}{lcllcl} D_x & = & (u_5 - u_3)/2 & D_x^{+z} & = & (u_5^{+z} - u_3^{+z})/2 \\ D_y & = & (u_7 - u_1)/2 & D_x^{-z} & = & (u_5^{-z} - u_3^{-z})/2 \\ D_z & = & (u_4^{+z} - u_4^{-z})/2 & D_y^{+x} & = & (u_8 - u_2)/2 \\ D_x^+ & = & u_5 - u_4 & D_y^{-x} & = & (u_6 - u_0)/2 \\ D_y^+ & = & u_7 - u_4 & D_y^{+z} & = & (u_7^{+z} - u_1^{+z})/2 \\ D_z^+ & = & u_4^{+z} - u_4 & D_y^{-z} & = & (u_7^{-z} - u_1^{-z})/2 \\ D_x^- & = & u_4 - u_3 & D_z^{+x} & = & (u_5^{+z} - u_5^{-z})/2 \\ D_y^- & = & u_4 - u_1 & D_z^{-x} & = & (u_3^{+z} - u_3^{-z})/2 \\ D_z^- & = & u_4 - u_4^{-z} & D_z^{+y} & = & (u_7^{+z} - u_7^{-z})/2 \\ D_x^{+y} & = & (u_8 - u_6)/2 & D_z^{-y} & = & (u_1^{+z} - u_1^{-z})/2. \\ D_x^{-y} & = & (u_2 - u_0)/2 & & & \end{array} \quad (9)$$

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

$$\mathbf{n}^+ = \begin{bmatrix} \frac{D_x^+}{\sqrt{(D_x^+)^2 + \left(\frac{D_y^{+x} + D_y}{2}\right)^2 + \left(\frac{D_z^{+x} + D_z}{2}\right)^2}} \\ \frac{D_y^+}{\sqrt{(D_y^+)^2 + \left(\frac{D_x^{+y} + D_x}{2}\right)^2 + \left(\frac{D_z^{+y} + D_z}{2}\right)^2}} \\ \frac{D_z^+}{\sqrt{(D_z^+)^2 + \left(\frac{D_x^{+z} + D_x}{2}\right)^2 + \left(\frac{D_y^{+z} + D_y}{2}\right)^2}} \end{bmatrix} \quad (10)$$

and

$$\mathbf{n}^- = \begin{bmatrix} \frac{D_x^-}{\sqrt{(D_x^-)^2 + \left(\frac{D_y^{-x} + D_y}{2}\right)^2 + \left(\frac{D_z^{-x} + D_z}{2}\right)^2}} \\ \frac{D_y^-}{\sqrt{(D_y^-)^2 + \left(\frac{D_x^{-y} + D_x}{2}\right)^2 + \left(\frac{D_z^{-y} + D_z}{2}\right)^2}} \\ \frac{D_z^-}{\sqrt{(D_z^-)^2 + \left(\frac{D_x^{-z} + D_x}{2}\right)^2 + \left(\frac{D_y^{-z} + D_y}{2}\right)^2}} \end{bmatrix} \quad (11)$$

respectively. The components of the divergence from equation 3 are then computed as

$$\frac{\partial \mathbf{n}_x}{\partial x} = \mathbf{n}_x^+ - \mathbf{n}_x^-, \quad (12)$$

$$\frac{\partial \mathbf{n}_y}{\partial y} = \mathbf{n}_y^+ - \mathbf{n}_y^-, \quad (13)$$

and

$$\frac{\partial \mathbf{n}_z}{\partial z} = \mathbf{n}_z^+ - \mathbf{n}_z^-, \quad (14)$$

Finally, we estimate H with

$$H = \frac{1}{2}\left(\frac{\partial \mathbf{n}_x}{\partial x} + \frac{\partial \mathbf{n}_y}{\partial y} + \frac{\partial \mathbf{n}_z}{\partial z}\right). \quad (15)$$

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

$$\nabla\phi_{\text{max}} = \begin{bmatrix} \sqrt{max(D_x^+, 0)^2 + max(-D_x^-, 0)^2} \\ \sqrt{max(D_y^+, 0)^2 + max(-D_y^-, 0)^2} \\ \sqrt{max(D_z^+, 0)^2 + max(-D_z^-, 0)^2} \end{bmatrix} \quad (16)$$

is computed followed by

$$\nabla\phi_{\text{min}} = \begin{bmatrix} \sqrt{min(D_x^+, 0)^2 + min(-D_x^-, 0)^2} \\ \sqrt{min(D_y^+, 0)^2 + min(-D_y^-, 0)^2} \\ \sqrt{min(D_z^+, 0)^2 + min(-D_z^-, 0)^2} \end{bmatrix}. \quad (17)$$

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

$$\nabla\phi = \begin{cases} \|\nabla\phi_{\text{max}}\|_2 & \text{if } F > 0 \\ \|\nabla\phi_{\text{min}}\|_2 & \text{otherwise} \end{cases}, \quad (18)$$

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

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

$$\phi(t + \triangle t) = \phi(t) + \triangle t F |\nabla\phi|. \quad (19)$$

APPENDIX B: GPU MEMORY ALLOCATION REQUEST GENERATION

This appendix describes the details of the GPU memory allocation/deallocation request scheme used by the GPU virtual memory system. The algorithm is described first in terms of an abstract client solver. Section IV and the B subsection of this appendix describe the client-specific details.

A. General Allocation Request Algorithm

The allocation request algorithm consists of the following steps (see the corresponding steps in Figure 8):

A. GPU computes VPN of requested active pages.
B,C. GPU compresses active-page request.
D. CPU processes memory request.
  1. Reads compressed request image from GPU.
  2. Decodes memory allocation/deallocation requests.
  3. Releases newly deactivated pages.
  4. Allocates/initializes newly activated pages.

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

In Step A, the GPU uses client-specific data to create two RGBA (i.e. 4-tuple) buffers that hold eight *true* or *false* (e.g. 255 or 0) values for each active data element (Figure 8). The first six values represent whether or not the virtual page in each of the six cardinal directions should be active for the next pass. The seventh value indicates if the active page itself should be active, and the eighth value is free to be used by the client. This eight-dimensional, active-page information vector, J, is thus $\mathrm{J} = (+\mathrm{x}, -\mathrm{x}, +\mathrm{y}, -\mathrm{y}, +\mathrm{z}, -\mathrm{z}, \mathrm{self}, \mathrm{clientSpecific})$, where the first six elements refer to relative neighbor offsets in the virtual page space, $\mathrm{V_P}$.

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

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

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

In Step D.1, the CPU reads the bit-code message from the GPU. Step D.2 begins by the CPU wrapping the message buffer with a bit-vector accessor. The resulting bit vector is a linear representation of the physical page space, $\mathrm{G_P}$, where each byte represents the information for a page. Two auxiliary bit-vectors are allocated–each a bit-addressed, linear representation of the virtual memory page space, $\mathrm{V_P}$. The first is the `newActiveSet` bit vector, and the second is the client-specific `eighthBitSet` bit vector. After the allocation message is decoded, a true bit in the `newActiveSet` bit vector will denote an active virtual page.

In the next stage of Step D.2, the CPU decodes the bit-vector message. For each 8-bit sequence, the current linear index is converted to a physical page number (PPN). The inverse page table then converts the PPN to a VPN. Because each bit in the bit-code message represents an offset direction from the current virtual page, the decoder can easily reconstruct the VPN for each neighbor of each active page. The decoder then reads the seven spatial page bits. It then computes the VPN for the page represented by each true bit and sets the corresponding bit in the `newActiveSet` bit vector to true. If the eighth bit is true, the `eighthBitSet` is set to true for the corresponding virtual page.

The virtual memory system next determines which virtual memory pages to deallocate and which to allocate (Steps D.3 and D.4). The set of newly deactivated pages is constructed by performing a set-subtraction of the `newActiveSet` from the `oldActiveSet`. The set of pages that need to be allocated for the next pass is created by computing the opposite set difference. Each deallocated memory page is pushed onto a stack of free memory pages. The page table are updated based on the client's implementation of `ReleasePage` function. Each newly activated page is mapped to a physical memory location by popping a page from the free page stack. The physical page is mapped in the page tables and the geometry engine is appropriately updated. The new physical memory is then initialized via the client's `InitNewPage` implementation.

B. Level-Set Solver Implementation Details

For Step A of the update algorithm described in Section IV-D and the preceding subsection, the level-set solver defines the functions `IsNeighborActive` and `IsSelfActive`. The `IsNeighborActive` reads the previously computed, one-side derivative that crosses a page boundary onto a specific neighbor. The function returns true if the derivative is non-zero. The `IsSelfActive` function returns true if *any* of the six, cardinal, one-sided derivatives are non-zero. The level-set solver simply writes the value of the level-set embedding to the eighth data value. This is used to determine if a newly deactivated page is inside or outside of the level-set surface. The `IsEigthBitTrue` function used by the fragment program in Step B returns true if the eighth data value is greater than zero. If a page becomes inactive, it is guaranteed to be either all black or all white. The down-sampled level-set embedding for the page will thus be either pure black or pure white.

The `eighthBitSet` used in the bit-code message decoding stage (Step D.2) is used to determine if a newly deactivated memory page is inside or outside the level-set surface. If the bit for the page is true, then the page is inside the surface. Otherwise it is outside. This information is used by the solver's `ReleasePage` function to map deactivated pages to the correct *static* physical page (white or black). As described in Section IV-A, these static mappings ensure that derivatives across boundaries of the active domain are correct.

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