A GPU-Based, Three-Dimensional Level Set Solver with Curvature Flow

Aaron Lefohn
School of Computing
Univ. of Utah
lefohnae@cs.utah.edu

Ross Whitaker
School of Computing
Univ. of Utah
whitaker@cs.utah.edu

University of Utah, School of Computing
Technical Report UUCS-02-017

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

December 11, 2002

Abstract

Level set methods are a powerful tool for implicitly representing deformable surfaces. Since their inception, these techniques have been used to solve problems in fields as varied as computer vision, scientific visualization, computer graphics and computational physics. With the power and flexibility of this approach; however, comes a large computational burden. In the level set approach, surface motion is computed via a partial differential equation (PDE) framework. One possibility for accelerating level-set based applications is to map the solver kernel onto a commodity graphics processing unit (GPU). GPUs are parallel, vector computers whose power is currently increasing at a faster rate than that of CPUs. in this work, we demonstrate a GPU-based, three-dimensional level set solver that is capable of computing curvature flow as well as other speed terms. Results are shown for this solver segmenting the brain surface from an MRI data set.

1 Introduction

The level set approach to representing deformable surfaces[1] has recently spurred advances in image processing[1, 2], surface processing[3, 4], image and volume segmentation[5, 6], surface reconstruction[7], and computer animation[8, 9, 10]. The level set technique represents surfaces implicitly as interfaces, and uses the framework of partial differential equations (PDEs) to compute surface motion[11]. In general, a set of speed functions are created to act on the surface and define a level set PDE, which is solved to find the surface position through time. The details of the speed functions are defined by the application.

In contrast to other modeling techniques, surfaces represented by a level set can easily change size, split into multiple entities, merge multiple surfaces into one and change topological genus. Much of this power and flexibility is due to the fact that level set methods embed a surface in a space that is one-dimensionally higher. This increase in dimensionality clearly leads to more computational complexity. The narrow-band[12] and sparse-field[13] approaches to solving level set equations have reduced the complexity to scale with the hyper-surface area[2] of the surface rather than the size of the space in which it is embedded. Despite these advances, most level set applications do not run at interactive rates. The surface editing operations in Museth et al.[4] took tens of seconds, and the surface reconstructions in Elangovan et al.[6] took up to five hours to compute.

Recent advances in commodity graphics processing units (GPUs) have made them an attractive alternative computing platform for certain applications. GPUs are parallel vector processors that are highly optimized for combining streaming two- and three-dimensional data to make a single, two-dimensional image. In the last two years, GPUs' fixed-function graphics pipeline has begun to be replaced with one that is highly programmable [14, 15, 16]. This programmability, combined with the vector nature of the processors, has opened up the possibility of using GPUs for computations other than scan-conversion.

Rumpf et al.[17] were the first to show that the level set equations could be solved using a graphics accelerator. Their solver implemented the two-dimensional level set method using a time-invariant speed function for flood-fill-like image segmentation. The same group and others [18, 19] have published work stating how various scientific

[1] *Surface* is used throughout this paper to mean a curve, surface, or hyper-surface.

[2] The "Hyper-surface area" is the measure of the $n-1$ space defined by a $\mathbb{R}^n$ surface. For a two-dimensional curve, the hyper-surface area is the arc length. For a three-dimensional surface it is the surface area.

computing primitive operations can be mapped to graphics hardware[20]. It has been shown that grid-based computations may benefit greatly from the high memory bandwidth and parallelism available on graphics boards. While these efforts have helped lay the groundwork for non-graphics use of GPUs, there are some key advances that must first be made before an easily-customizable, three-dimensional level set solver will be made to run interactively on a GPU.

From a practical standpoint, writing scientific computing software for current GPUs is difficult, error-prone, and tedious. The first reason for this is that all GPU instructions are issued via a graphics API such as OpenGL[21] or Direct3D[22]. This fact makes for a non-obvious mapping of scientific computing algorithms to GPUs. This work addresses this problem. The second difficulty in using GPUs for scientific computing is that current GPUs are only capable of low-precision, fixed-point arithmetic. As a result, scales and biases must be carefully applied throughout the calculation. Future GPU releases will be capable of floating-point computations, but memory limitations may continue to make fixed-point computations an attractive option. The other obstacle to computing with GPUs is that much of the software must be written in a hardware-specific assembly language. This of course makes the code error-prone as well as hard to write, read, and reuse. Proudfoot et al.'s *Stanford Shading Language*[23], the OpenGL 2.0 shading language[24], Nvidia's *CG* language[25], and Microsoft's *High Level Shading Language*[26] all attempt to provide a high-level language interface with which to program GPUs. The GPU and operating system specificity of the language, however, may still remain a challenge to cross-platform developers until a standard is established.

The next section discusses the technical background behind level set methods and graphics accelerators. Section 3 describes the mapping of the level set equations to a GPU and the software framework that has been built to support it. Section 4 discusses our results and possibilities for future work.

2 Technical Background

2.1 Level Set Explanation

In the level set approach, a n-dimensional surface is embedded in a $\mathbb{R}^{n+1}$ space. A scalar function, $\phi(\mathbf{x}, t)$ defines an embedding of a surface, where $\mathbf{x} \in \mathbb{R}^{n+1}$ and t is time. The set of points on the surface, $\mathbf{S}$, are mapped by ϕ such that

$$\mathbf{S} = \{\mathbf{x} | \phi(\mathbf{x}) = k\}, \tag{1}$$

where k is an arbitrary scalar value (often zero). It can also be said that $\mathbf{S}$ is the k level set of ϕ. A closed form of ϕ is not known, but an initial estimation of it can be obtained by building a discrete sampling of it on a $\mathbb{R}^{n+1}$ grid. This is done by setting each point in $\mathbf{S}$ to k, all points inside the interface to $\phi > k$ and positions outside to $\phi < k$. If the initial sampling of ϕ sets all values in the range $[0, 1]$, then a curve or surface can easily be represented by a grayscale image or volume.

In order to propagate ϕ (and therefore the surface) in time, we define the first-order, partial differential equation

$$\frac{\partial \phi}{\partial t} = F|\nabla \phi|, \tag{2}$$

where F is a signed, scalar speed function that defines the speed in the direction normal to ϕ at any point $\mathbf{x}$. F may be of the form $F(\mathbf{x})$, $F(\mathbf{x}, t)$, or $F(\phi, D_\phi, \ldots)$. The choice of speed function is defined by the application and combinations of them may be used to capture a desired behavior.

The initial estimation of ϕ is propagated forward in time via the up-wind scheme[27]. To guarantee a stable solution, the upwind scheme approximates $\nabla \phi$ using one-sided derivatives that are always in the "up-wind" direction of the propagating surface. The largest allowable time step, $\triangle t$, is inversely proportional to the maximum speed at a given time, t. The solution will become unstable if a larger value for $\triangle t$ is used. Given that $\frac{\partial \phi}{\partial t}$ is defined by equation 2 and the general update equation is

$$\phi(\mathbf{x}, t + \triangle t) = \phi(\mathbf{x}, t) + \triangle t \frac{\partial \phi}{\partial t}, \tag{3}$$

the level set update equation is

$$\phi(\mathbf{x}, t + \triangle t) = \phi(\mathbf{x}, t) + \triangle t F |\nabla \phi|. \tag{4}$$

The mean curvature of ϕ, hereafter referred simply as curvature, is commonly used as a speed function to propagate ϕ. A surface under curvature flow will become "smoother" and in fact, under pure curvature flow, a convex surface will converge to the n-sphere and finally a single point[28]. Curvature flow is often combined with other speed functions to smooth out an otherwise rough surface solution. The mean curvature of ϕ is defined as

$$H = c_n \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|}, \tag{5}$$

where, if n is the dimensionality of the surface, $c_n = 1/(n-1)$. The details of estimating $\nabla \phi$ and H are presented in section 3.

2.2 Details of a Graphics Processing Unit

Graphics processing units are designed to render vertex positions, material properties, lighting information, and texture maps into a single, two-dimensional image. In addition to the (x, y, z) position, each vertex may have a set of texture coordinates, a normal vector, and/or a color associated with it. The processing unit can be thought of as a pipeline (figure 1). This pipeline first transforms vertices into a common model space, then applies lighting calculations to each vertex. The vertex information is then rasterized into *fragments*. A fragment is the generalization of a pixel. In addition to the (x, y, z) position and (r, g, b, α) values associated with a pixel, a fragment may also contain a set of interpolated texture coordinates. There can be multiple fragments that map to the same (x, y) pixel location. After being rasterized a fragment is textured and then passed through a series of tests before becoming a pixel. These tests include the scissor, alpha, stencil and depth test. If a fragment survives all of these and several other operations, it becomes a pixel. The final output is the (r, g, b, α) 4-tuple as well as the pixel's depth (z) value.

GPU programming consists of two basic types of operations: those that set or unset pipeline state and those that move data through the pipeline. In a fixed-function graphics pipeline, render state is set/unset by making graphics API calls. A programmable pipeline provides an assembly-level or language-level interface for specifying portions of the render state. There are two points of programmability in most modern GPUs: the vertex stage and the fragment stage. Vertex programs (or shaders as they are commonly referred to) control the vertex transformation and texture coordinate pipeline stages. Fragment shaders allow a programmer to specify the final color of a fragment by combining multiple texels, texture coordinates, and colors with a limited instruction set. A *texel* is the texture map equivalence of a pixel.

A *render pass* is defined as one set of data moving entirely through the GPU pipeline. The destination of a render pass is normally the color buffer, which can then be displayed to a video monitor. The destination can alternatively be a non-displayable buffer called a pixel buffer (pbuffer). Pbuffers can be sized differently than the display window and can also be associated with a texture (render-to-texture). When associated with a texture, the pbuffer memory can be both rendered into and bound as a texture map. This will prove to be an important feature when using GPUs as computational platforms. We have found that changing between destination buffers is a very slow process, taking approximately 0.25 milliseconds. We have thus gone to great lengths to avoid changing pbuffer targets.

A given GPU will support some number of *texture units*, N_t, which determines how

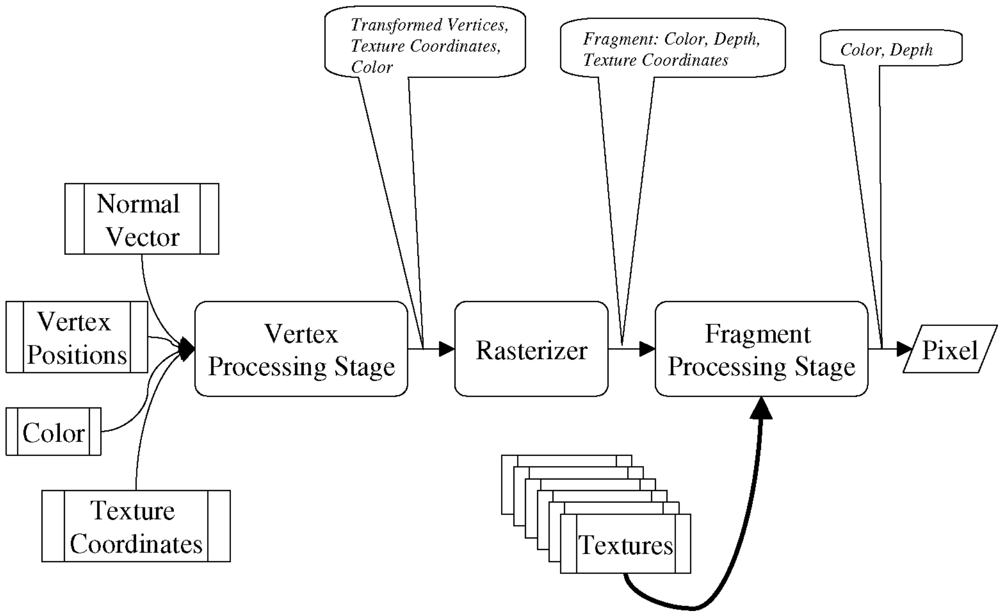
Transformed Vertices, Texture Coordinates, Color
Fragment: Color, Depth, Texture Coordinates
Color, Depth
Normal Vector
Vertex Positions
Color
Texture Coordinates
Vertex Processing Stage
Rasterizer
Fragment Processing Stage
Pixel
Textures

Figure 1: *The graphics processing unit (GPU) pipeline.*

many textures may be simultaneously applied in the same render pass. Each textured vertex may be given up to N_t separate texture coordinates (location in a texture map). Note that each of these coordinates may be for a one, two, or three dimensional texture. Also note that texture coordinates will generally not fall exactly on a texel location. The texture sampling can therefore be specified to use the nearest texel or compute an interpolated value at the texture coordinate position. The interpolation is linear, bi-linear, or tri-linear depending on the dimensionality of the texture map. N_t is between two and six in current hardware and is expected to increase substantially in future cards.

The remaining sections will give specific hardware examples in terms of the ATI Radeon 8500 GPU because it is the GPU on which we have implemented my level set solver. While future Nvidia GPUs will be supported, the 8500's fragment shader features were the most feature-rich at the time this work began.

Vertex shaders are powerful tools for graphics applications, but for grid-based computations, such as a level set solver, nearly all of the calculations are performed with the fragment shader. The discussion will thus be focused on the details of the latter. The input to a fragment shader program consists of up to N_t texture coordinates, up to N_t textures, two (r, g, b, α) colors, and up to N_c fixed-point constants. Each texture coordinate may contain up to four, 32-bit floating-point values. All other inputs are 8-bit, fixed point numbers. The output from a fragment shader is a single, 8-bit, fixed-point (r, g, b, α) 4-tuple. The internal precision of the temporary registers is generally higher precision than the input and output. As with the other limitations, it is expected that the available precision will increase on future GPUs. In the short term, however, the work proposed herein must address the challenges presented by computing with fixed-point operations.

All fragment shader instructions are vector-type instructions that operate on all fragments in a render pass. The fragments are processed in parallel by multiple functional units and an entire (r, g, b, α) 4-tuple is processed by each instruction. The number of instructions allowed in a fragment program, N_f, is limited by the GPU hardware (the ATI Radeon 8500 allows only sixteen). It is expected that future hardware releases will substantially increase the number of allowed instructions. Fragment shaders have a limited number of temporary registers, N_r, available to hold intermediate computations. This is often the same as the number of texture units. Just as with N_t, N_r in current hardware is between two and six.

In addition to sampling the input textures, the ATI Radeon 8500's fragment shader supports the following operations: add, subtract, multiply, move, conditional choice of two values and dot product. Division is supported only by divisors of two, four, and eight. Instruction modifiers allow for the instruction destination to mask any of the four channels and for an instruction source to repeat any one channel on all four channels. Branching and subroutine calls do not exist.

An especially powerful feature in programmable GPUs is dependent texture reads. This allows for the computation of texture coordinates within a fragment shader and the use of those coordinates to perform a lookup into another texture. Dependent texture reads allows the use of textures as lookup tables (LUTs). The ATI Radeon's hardware allows for two sample stages per fragment shader. In the first stage, up to N_t textures are sampled and mathematical operations are performed on the input. The second stage begins when the same textures are sampled again, but now with the possibility of using a computed result as the texture coordinates for the lookup.

Grid-based computations on GPUs use texture maps to hold all input, intermediate and output data. Each render pass renders to a texture map via a pbuffer, and that texture is bound as input to a later pass. As mentioned early, the data in each texture are 8-bit, fixed-point values. Computations are performed by mapping multiple texture maps to a single, planar quadrilateral (a *compute slab*) and using a fragment shader to combine the inputs into a single output. For three-dimensional calculations, a stack of compute slabs is used, and the data is stored in multiple two-dimensional textures. Each compute slab uses data from the slabs above and below it in addition to its own.

3 Design

Our system consists of an OpenGL-based, 2D and 3D level set solver running on an ATI Radeon 8500 graphics processing unit. An image and volume segmentation program has been created based on this solver. The speed function for this application is created by combining a pre-computed, data-driven speed with the mean curvature.

3.1 Computation Overview

The 2D solver requires five render passes to compute the curvature speed term, a pass to combine the speed terms, three more passes to compute the up-wind approximation to $\nabla\phi$, and a single pass to combine this information and update $\phi(t)$ to $\phi(t + \triangle t)$. If only time-invariant speed functions are used, ϕ can be advanced in six instead of ten passes. Pseudocode for the 2D solver is shown below. Each numbered line represents a render pass, with the input textures as function arguments. Destination pbuffers/textures are considered variables, and temporaries are allocated using register allocation strategies.

```
for(int t=0; t < numSteps; t++) {
  // Compute first set of derivatives
  Tex2D d1    = deriv1( phi[z] );                        // 1

  // Compute Curvature
  Tex2D d2     = deriv3( phi[z] );                       // 2
  Tex2D d3     = deriv4( phi[z] );                       // 3
  Tex2D cx     = curvX( d1, d2, d3, normalizeLUT );      // 4
  Tex2D curv   = curvY( d1, d2, d3, cx, normlalizeLUT ); // 5

  // Sum the speed functions
  Tex2D speed  = sumSpeed( curv, G );                    // 6

  // Upwind Computation
  Tex2D minG   = minGrad( d1 );                          // 7
  Tex2D maxG   = maxGrad( d1 );                          // 8
  Tex2D gMag   = gradMag1( minG, maxG, speed,            // 9
                           phi[z], l2NormLUT );

  // Do PDE timestep update
  Tex2D phi[z] = phiUp( gMag, multScaleLUT );            // 10
}
```

The partitioning of the computation into render passes is dictated by N_t, N_r and N_f. The current implementation attempts to exploit the fact that entire 4-tuples are processed in a single instruction, but parallelism is limited by lookup table accesses, the single output limitation, and of course computational dependencies.

The 3D solver requires seven render passes per slab to compute the mean curvature, and a total of sixteen render passes per slab to compute an entire time step update. For a $256x256x175$ data set, this means that 2800 render passes are required to update the entire volume a single PDE time step. Pseudocode for the 3D solver is shown below.

```
for(int t=0; t < numSteps; t++) {
  for(int z=0; z < numSlabs; z++) {
    // Compute two sets of 4-vec derivatives
    Tex2D d1    = deriv1( phi[z] );                          // 1
    Tex2D d2    = deriv2( phi[mz], phi[pz] );                // 2

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

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

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

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

For the remainder of the discussion, only the three-dimensional solver will be discussed in detail, but the two-dimensional case can easily be derived from the discussion.

3.2 Initialization

The initialization stage of the program loads two grayscale volumes into texture memory on the GPU: the source image to be segmented, $I(x, y, z)$, and the initial level set image, $\phi(x, y, z, 0)$. The initial ϕ solution is a grayscale volume with pixel values of 255 inside a boundary (often a sphere), 127 on the boundary and 0 outside (figure 2).

The scalar ϕ data is packed into the RGB channels of RGBA, 2D textures. The slabs are processed bottom to top and the old version of each slab is stored in the α channel of each RGB pbuffer in order for the next slab to use to compute the correct derivatives in the z-direction. Because these special render-to-texture buffers must be in RGB or RGBA format, this packing both reduces memory usage and avoids the costly changing of destination buffers. Note that this packing relies on the ability to use a buffer as both a texture input and a render destination in the the same pass. While this behavior is explicitly disallowed by the render-to-texture specification, current display drivers permit this operation. Because the render pipeline computes pixels in parallel with no guarantees of ordering, the sampling of neighbors in this situation can be undefined. If, however, one is careful to only sample neighborhoods of data which is not currently being written, then the usage has a clear meaning. We would like to emphasize the importance of this feature and encourage GPU manufactures to allow for this usage model.

The data-driven speed function, $\mathbf{G}(x, y, z)$ is computed based on two thresholding constants, I_{lo} and I_{hi}, input from the user interface. This time-invariant, spatial speed function is computed as

$$\begin{aligned} I_{ave} &= \frac{I_{hi} - I_{lo}}{2} \\ \mathbf{G}(x, y) &= \begin{cases} I(x, y, z) - I_{lo} & \text{if } I \leq I_{ave} \\ I_{hi} - I(x, y, z) & \text{otherwise} \end{cases} . \end{aligned} \tag{6}$$

This data is stored into texture memory in 2D, luminance textures. This speed function attracts the level set surface towards grayscale values in the source image that are $\in [I_{lo}, I_{hi}]$. A graph of equation 6 is shown in figure 3.

3.3 Computation Details

For the sake of clarity, all quantities except for the constant scalars, I_{lo} and I_{hi}, are assumed to be three dimensional samplings and the operations thus apply to all sample points (e.g. $\phi \Rightarrow \phi(x, y, z)$). This notational convention closely matches the way in which the vector instructions are specified to the GPU. It is also assumed

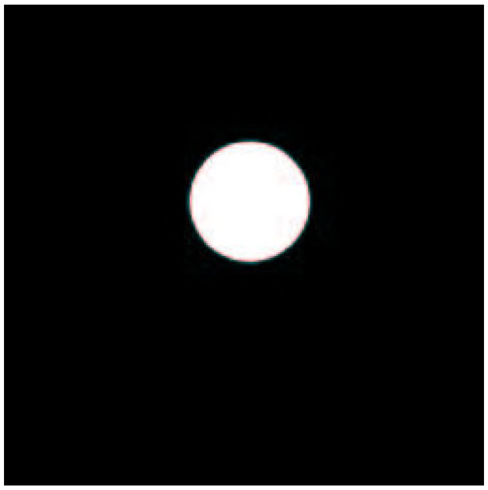

Figure 2: *2D slice of the initial level set solution.*

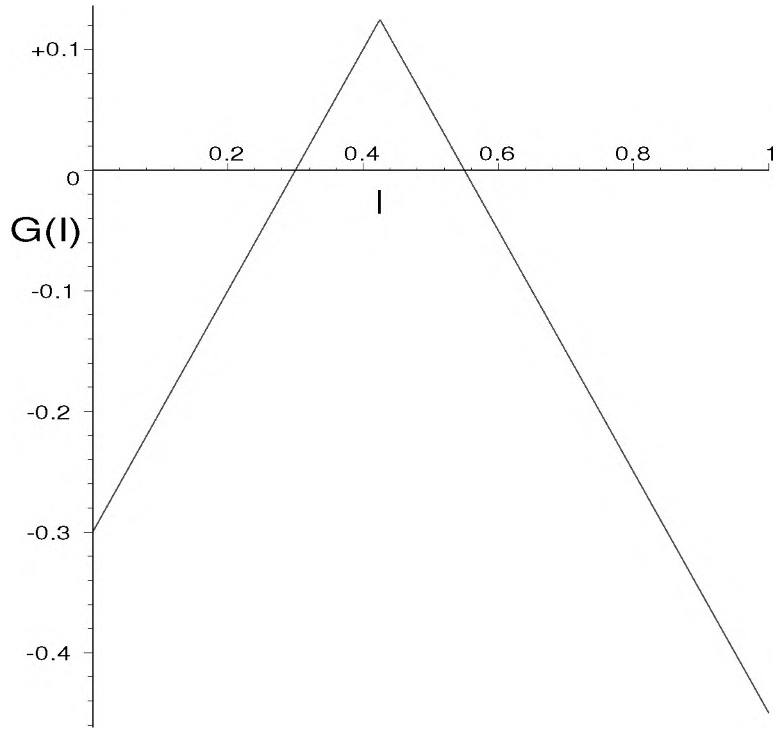

Figure 3: *The data-driven speed function, G, with $I \in [0, 1]$.*

that all quantities are sampled at the same time step, t, unless a time argument is explicitly given as in equation 19.

Six render passes are required to compute the twenty-one different derivatives required for the curvature and upwind computation. The neighborhood, n, from which these derivatives are computed is specified with the numbering scheme

$$\begin{array}{|c|c|c|}\hline 6 & 7 & 8 \\ \hline 3 & 4 & 5 \\ \hline 0 & 1 & 2 \\ \hline \end{array}\,. \tag{7}$$

Note that 4 denotes the center pixel, and $n_i^{\pm z}$ represents the ith sample on the slab above or below the current one. Neighborhoods are sampled by shifting the texture coordinates by the desired number of texels in each direction. The derivatives of ϕ are defined as

$$\begin{array}{rcl} D_x &=& (n_5 - n_3)/2 \\ D_y &=& (n_7 - n_1)/2 \\ D_z &=& (n_4^{+z} - n_4^{-z})/2 \end{array} \quad \begin{array}{rcl} D_x^+ &=& n_5 - n_4 \\ D_y^+ &=& n_7 - n_4 \\ D_z^+ &=& n_4^{+z} - n_4 \\ D_x^- &=& n_4 - n_3 \\ D_y^- &=& n_4 - n_1 \\ D_z^- &=& n_4 - n_4^{-z} \end{array} \quad \begin{array}{rcl} D_x^{+y} &=& (n_8 - n_6)/2 \\ D_x^{-y} &=& (n_2 - n_0)/2 \\ D_x^{+z} &=& (n_5^{+z} - n_3^{+z})/2 \\ D_x^{-z} &=& (n_5^{-z} - n_3^{-z})/2 \\ D_y^{+x} &=& (n_8 - n_2)/2 \\ D_y^{-x} &=& (n_6 - n_0)/2 \\ D_y^{+z} &=& (n_7^{+z} - n_1^{+z})/2 \\ D_y^{-z} &=& (n_7^{-z} - n_1^{-z})/2 \\ D_z^{+x} &=& (n_5^{+x} - n_5^{-z})/2 \\ D_z^{-x} &=& (n_3^{+x} - n_3^{-z})/2 \\ D_z^{+y} &=& (n_7^{+x} - n_7^{-z})/2 \\ D_z^{-y} &=& (n_1^{+x} - n_1^{-z})/2 \end{array}\,. \tag{8}$$

Note that in order to retain as much precision as possible, ϕ is stored $\in [0, 255]$ but in order for the derivatives to stay within the same range, they are computed with $\phi/2$. The range $[-127, +127]$ is represented by $[0, 255]$. Results of derivatives are shifted by -127 when used and $\nabla\phi$ is scaled by 2 in the ϕ update step (equation 19).

Curvature is computed using the above derivatives and the "difference of normals" method introduced by Whitaker and Xue[1]. The two normals, $\mathbf{n}^+$ and $\mathbf{n}^-$, are computed by

$$\mathbf{n}^+ = \begin{bmatrix} \dfrac{D_x^+}{4\sqrt{(D_x^+)^2+\left(\frac{D_y^{+x}+D_y}{2}\right)^2+\left(\frac{D_z^{+x}+D_z}{2}\right)^2}} \\ \\ \dfrac{D_y^+}{4\sqrt{(D_y^+)^2+\left(\frac{D_x^{+y}+D_x}{2}\right)^2+\left(\frac{D_z^{+y}+D_z}{2}\right)^2}} \\ \\ \dfrac{D_z^+}{4\sqrt{(D_z^+)^2+\left(\frac{D_x^{+z}+D_x}{2}\right)^2+\left(\frac{D_y^{+z}+D_y}{2}\right)^2}} \end{bmatrix} \tag{9}$$

and

$$\mathbf{n}^- = \begin{bmatrix} \dfrac{D_x^-}{4\sqrt{(D_x^-)^2+\left(\frac{D_y^{-x}+D_y}{2}\right)^2+\left(\frac{D_z^{-x}+D_z}{2}\right)^2}} \\ \\ \dfrac{D_y^-}{4\sqrt{(D_y^-)^2+\left(\frac{D_x^{-y}+D_x}{2}\right)^2+\left(\frac{D_z^{-y}+D_z}{2}\right)^2}} \\ \\ \dfrac{D_z^-}{4\sqrt{(D_z^-)^2+\left(\frac{D_x^{-z}+D_x}{2}\right)^2+\left(\frac{D_u^{-z}+D_y}{2}\right)^2}} \end{bmatrix} \tag{10}$$

respectively. The components of the divergence from equation 5 are then computed as

$$\frac{\partial \mathbf{n}_x}{\partial x} = \mathbf{n}_x^+ - \mathbf{n}_x^-, \tag{11}$$

$$\frac{\partial \mathbf{n}_y}{\partial y} = \mathbf{n}_y^+ - \mathbf{n}_y^-, \tag{12}$$

and

$$\frac{\partial \mathbf{n}_z}{\partial z} = \mathbf{n}_z^+ - \mathbf{n}_z^-, \tag{13}$$

Finally, the curvature,

$$H = \frac{\partial \mathbf{n}_x}{\partial x} + \frac{\partial \mathbf{n}_y}{\partial y} + \frac{\partial \mathbf{n}_z}{\partial z} \tag{14}$$

is computed. Note that the normalization of $\mathbf{n}^+$ and $\mathbf{n}^-$ are performed with a cube map lookup table because division and square-root are not instructions available in the Radeon 8500's fragment shader hardware. A lookup table is also used because the intermediate values in the normalization may get quite large, but the end result is within the limited, available range. The constant factor of four in equation 9 and 10 is necessary to keep the resulting value $\in [0, 255]$.

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

$$\nabla\phi_{\max} = \begin{bmatrix} \sqrt{max(D_x^+,0)^2 + max(-D_x^-,0)^2} \\ \\ \sqrt{max(D_y^+,0)^2 + max(-D_y^-,0)^2} \\ \\ \sqrt{max(D_z^+,0)^2 + max(-D_z^-,0)^2} \end{bmatrix} \tag{15}$$

is computed followed by

$$\nabla\phi_{\min} = \begin{bmatrix} \sqrt{min(D_x^+,0)^2 + min(-D_x^-,0)^2} \\ \\ \sqrt{min(D_y^+,0)^2 + min(-D_y^-,0)^2} \\ \\ \sqrt{min(D_z^+,0)^2 + min(-D_z^-,0)^2} \end{bmatrix}. \tag{16}$$

Just as with the normalization of the gradient vectors, the euclidean norms in the above equations are computed with a lookup table. The final choice of $\nabla\phi$ is defined by

$$\nabla\phi = \begin{cases} \|\nabla\phi_{\max}\|_2 & \text{if } F > 0 \\ \|\nabla\phi_{\min}\|_2 & \text{otherwise} \end{cases}, \tag{17}$$

where F is computed as the linear combination of H and G:

$$\mathbf{F} = \; c_0\mathbf{G} + c_1\mathbf{H} \quad \text{where } c_1 = 1 - c_0 \; . \tag{18}$$

The $\phi(t + \triangle t)$ values are then computed by

$$\phi(t + \triangle t) = \phi(t) + \triangle t c F|\nabla\phi|, \tag{19}$$

where c is the scaling by two necessitated by ϕ being stored in $[0, 255]$. Due to roundoff errors and other inconsistencies in the GPU processing, $c = 1.9$ in this implementation. The next section discusses this issue further. Because c must be a floating-point value, it is passed into the fragment shader as an "extra" texture coordinate.

3.4 Object Oriented OpenGL Framework

The current level set solver is built with modules from an object-oriented framework that has been built on top of OpenGL. The design attempts to provide a framework in which re-usable OpenGL code can easily be written. Unlike other object-oriented OpenGL encapsulations such as GLT[29] and OpenInventor[30], my framework is designed for low-level OpenGL developers rather than high-level graphics programmers. It also does not attempt to encapsulate any windowing-related calls other than the handling of pbuffers. It is expected that GLUT or some other windowing utility will be used.

The new framework defines a set of reusable and extensible modules that can be composited into higher-level objects. The multi-level approach is very flexible in that a user can choose to work at any level from raw OpenGL code to manipulating entire render passes. Multiple levels can be used within the same application. Another goal of the library is to encapsulate all hardware-specific OpenGL code into pluggable modules so that multiple GPU architectures can be easily supported by an application. In addition all OpenGL extensions from Nvidia and ATI are automatically loaded. This framework is referred to as *Glift*. A class tree of Glift is shown in figure 4.

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

In addition, all Glift objects support a `compile()` method that attempts to compile the OpenGL commands encapsulated by the object into a display list. Note that this feature provides a way to "compile away" the abstraction penalty that might otherwise exist. My experience thus far has shown, however, that the current level set solver is completely GPU-bound and the Glift abstraction layers do not affect the execution speed of the code.

Glift is designed to provide a minimal amount of pre-encapsulated OpenGL state and have obvious extension points for adding more functionality as desired. As Glift matures, more functionality can come pre-defined by the library. The following is a list of current extension points:

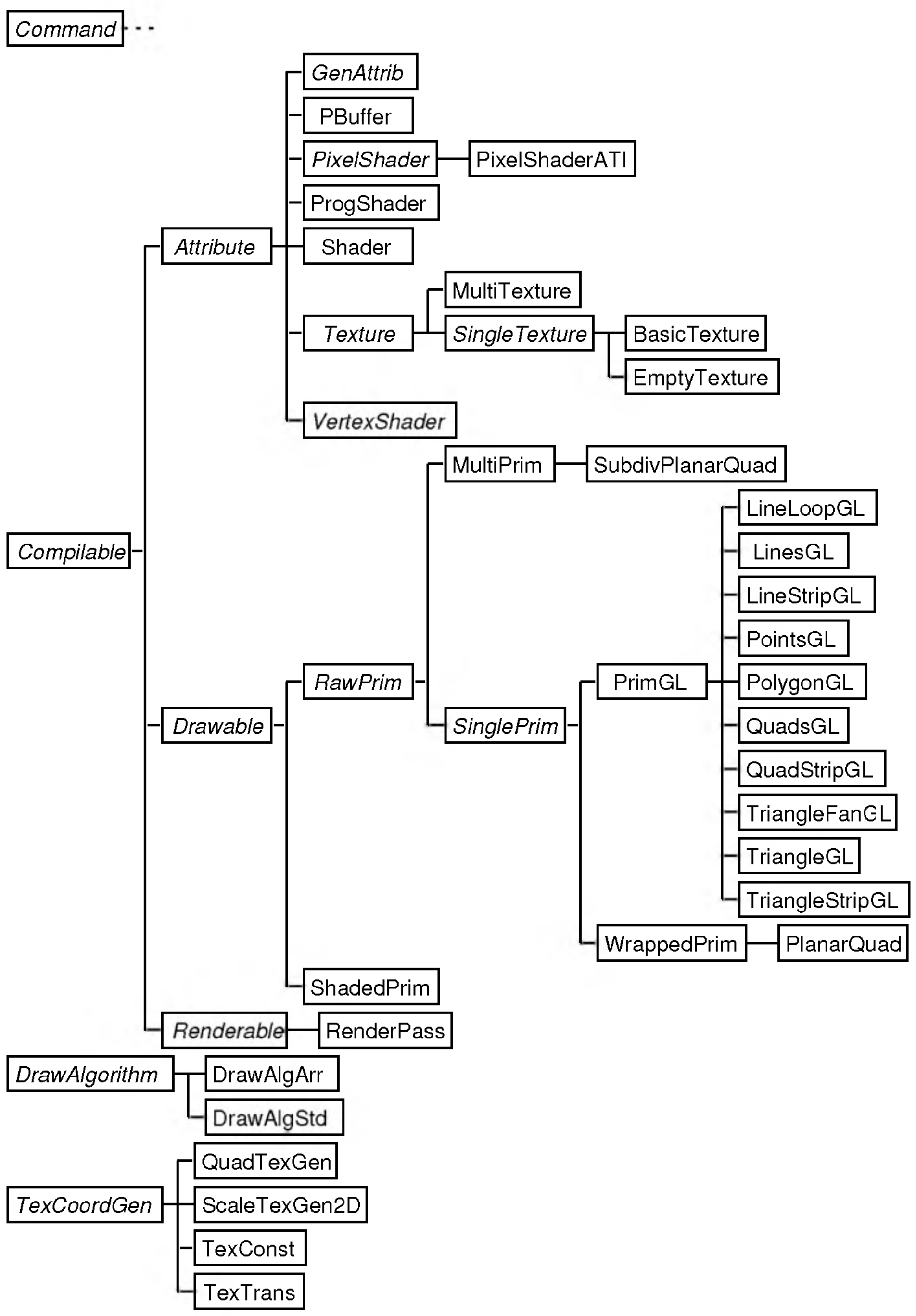

Figure 4: *Class tree for the Glift, object-oriented OpenGL framework.*

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

To build a render pass, the attributes are first combined into a `Shader` object. The `Drawables` are then defined and put into a `MultiPrim`. The `Shader` object and the `Multiprim` (or any other `RawPrim`) are combined into a `ShadedPrim` object. At this point, any texture coordinate perturbations (texture coordinate offsets and texture coordinate constants) that are defined by the shader are applied to the texture coordinates for the primitives. In addition, if multiple textures are specified in the shader, multiple sets of texture coordinates are generated. This `ShadedPrim` object (or any `Drawable`) is combined optionally with a texture and/or pbuffer destination into a `RenderPass`.

Glift has been used to build a `ComputeSlab` class that serves as a base class to all the render passes in the level set solver. The subclasses of `ComputeSlab` define only their input textures (`SingleTexture` pointers), the shading language being used, the destination pbuffer, texture coordinate perturbations, and the fragment shader (potentially in multiple languages). A `ComputeSlab` can be thought of as a function call where the input textures are the arguments, the fragment program is the subroutine code, and the destination pbuffer holds the results.

4 Results

We have built a two- and three-dimensional, GPU-based level set solver and used it to create an image and volume segmentation application. To our knowledge, this is the first level set solver implemented on a GPU that includes curvature flow. For two dimensions and small three-dimensional datasets, the program runs at interactive

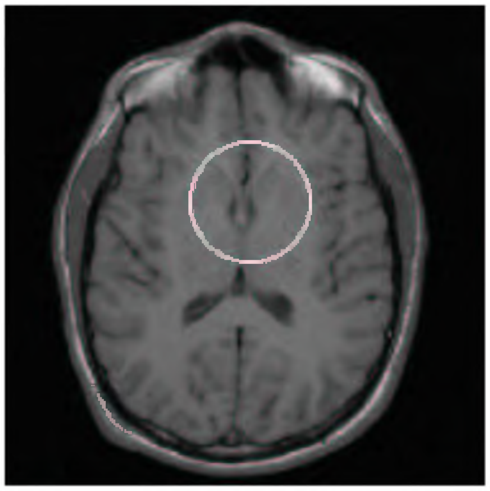

Figure 5: *Initial level set solution (white line) superimposed on a slice of MRI data courtesy of David Weinstein.*

rates. This allows users, for the first time, to use parameters such as the fraction of curvature flow as visualization parameters to achieve the desired segmentation.

Figure 5 shows the source image for a 2D segmentation with the initial level set solution as a white circle near the center. The segmentation was accomplished in three stages. Each stage consisted of enough time steps (500) to allow the level set solution to converge. The grayscale thresholding constants, I_{lo} and I_{hi} were set to 0.3 and 0.55 respectively for all three stages. In the first stage, the speed function was composed entirely of G with no curvature flow included. This created the noisy image surface shown in figure 6. The second stage (figure 7) used a speed function with 50% curvature flow. Note that much of the fine detail has been lost due to the smoothing effect of the curvature. In the last stage, only 5% curvature was used in the speed function (figure 8) in order to regain much of the fine detail present in 6 but without the noise. These segmentations qualitatively match the results of a floating-point, software level set solver.

In the current version of the solver, one time step on a 256 x 256 image takes 4 milliseconds with curvature flow enabled and 2 milliseconds without it. This is approximately the same speed as a highly-optimized, sparse-field software implementation. This result is disappointing but not surprising, given the fact that the GPU-based solver is computing an update at every pixel whereas the software implementation is only computing updates on or near the isosurface.

Figure 10 shows a brain surface extracted from a $256x256x175$, MRI data set (figure

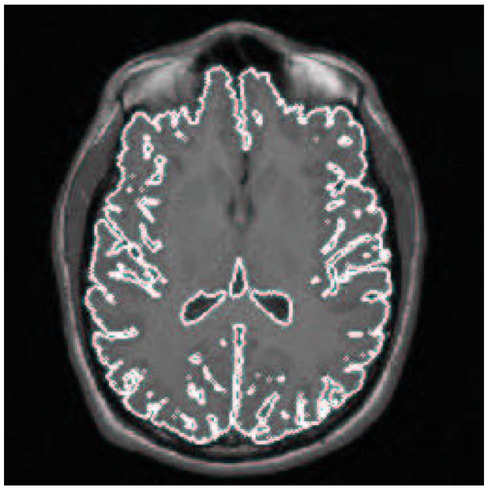

Figure 6: *Level set solution after the first segmentation stage (white line). Grayscale threshold values were set to* $I_{lo} = 0.3$ *and* $I_{hi} = 0.55$*, and no curvature flow was used.*

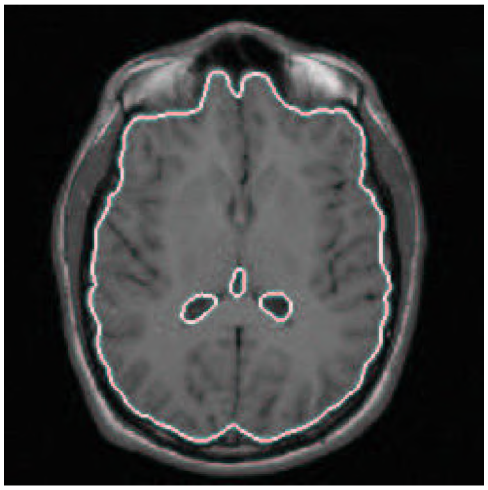

Figure 7: *Level set solution after the second segmentation stage (white line). Grayscale threshold values were set to* $I_{lo} = 0.3$ *and* $I_{hi} = 0.55$*, and* 50% *curvature flow was used.*

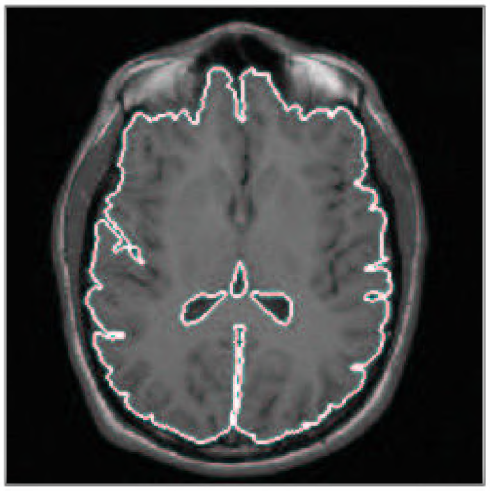

Figure 8: *Level set solution after the third segmentation stage (white line). Grayscale threshold values were set to* $I_{lo} = 0.3$ *and* $I_{hi} = 0.55$, *and* 5% *curvature flow was used.*

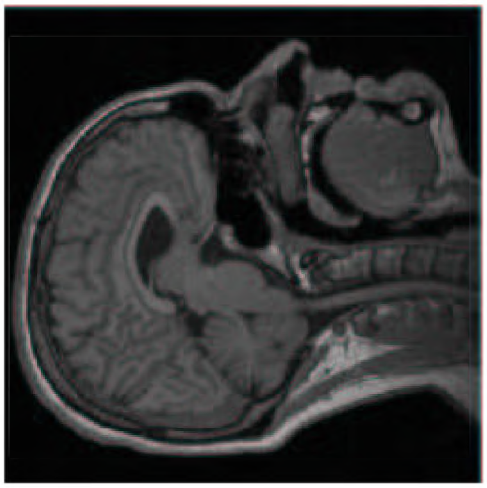

Figure 9: *Slice of the* $256x256x175$ *MRI data from which figures 10 and 11 were segmented.*

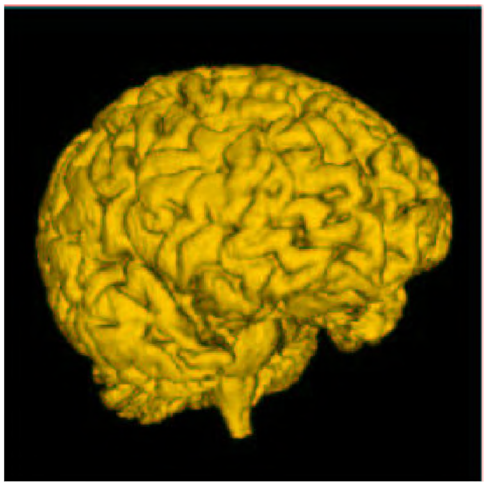

Figure 10: *Brain surface segmented using the GPU-based level set solver. Both the data-driven and curvature speed terms were used to obtain this surface.*

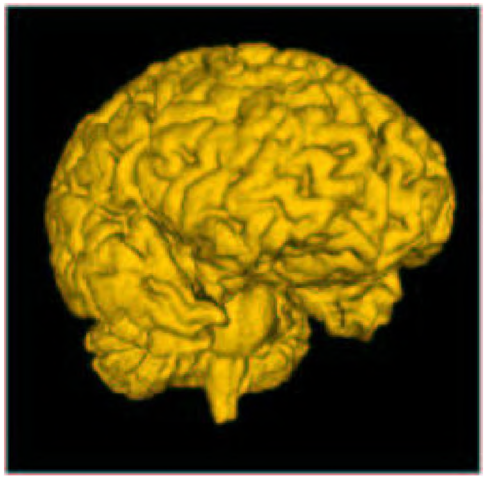

Figure 11: *Brain surface segmented using a CPU, software-based solver. The same data-driven and curvature speed terms used for the GPU-based segmentations were also used to obtain this surface.*

9) using our GPU-based level set solver. Figure 11 shows the brain surface extracted from the same data set using a floating-point, software solver. Note that the surfaces are qualitatively similar. Our solver is approximately two times faster than the software implementation despite that it is performing roughly ten times as many calculations. If the GPU solver were only processing the voxels near the isosurface, the theoretical speedup is greater than twenty times over the software version. Note that GPUs are also increasing in power at a faster rate than CPUs so this gap will continue to increase over time.

5 Discussion

We have implemented a two- and three-dimensional level set solver, that includes mean curvature flow, on the ATI Radeon 8500. All computations are performed in 8-bit, fixed-point arithmetic. In 2D, this solver runs at approximately the same speed as an optimized software implementation. A 3D example running on a $256x256x175$ data set runs at twice the speed of the CPU-based solution. Although these results are encouraging, they are merely a proof of concept.

Future work will take advantage of more the powerful fragment programs of the next generation of GPUs. It currently takes sixteen render passes per slice for a single PDE time step update. This number will be significantly reduced on new GPUs due to an increased instruction limit, a richer instruction set, and the ability to output more than four values per pass.

Our solver is computing PDE updates at all pixel/voxel locations, and is thus not taking advantage of the sparse nature of the moving wavefront. We estimate that speedups of at least twenty five times are possible by computing only those values near the surface. Future work will explore the use of depth and stencil culling, hierarchical spatial decompositions, compression of the dynamic texture data, and other possibilities to achieve this optimization.

Acknowledgements

The authors would like to thank Jeff Royle, Evan Hart, and Jason Mitchell at ATI Technologies, Inc. for many discussions related to this work. We would also like to thank Joe Kniss and Milan Ikits for their many useful discussions on the software design. This work has been funded by grant #N000140110033 from the Office of Naval Research and grant #ACI0089915 from the National Science Foundation.

References

[1] 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.

[2] A. Marquina and S. Osher, "Explicit algorithms for a new time dependent model based on level set motion for non-linear deblurring and noise removal," *SIAM Journal on Scientific Computing*, vol. 22, pp. 387–405, 2000.

[3] T. Tasdizen, R. Whitaker, P. Burchard, and S. Osher, "Geometric surface smoothing via anisotropic diffusion of normals," in *IEEE Visualization 2002*, p. To appear., October 2002.

[4] K. Museth, D. Breen, R. Whitaker, and A. Barr, "Level-set surface editing operators," in *ACM SIGGRAPH*, p. To appear., 2002.

[5] R. Malladi and J. A. Sethian, "A unified approach to noise removal, image enhancement, and shape recovery," in *IEEE Transactions on Image Processing*, vol. 5, pp. 1554–15568, 1996.

[6] V. Elangovan and R. Whitaker, "From sinograms to surfaces: A direct approach to the segmentation of tomographic data," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, pp. 213–223, Oct. 2001.

[7] R. Whitaker, "Reconstructing terrain maps from dense range data," in *IEEE International Conference on Image Processing*, pp. 165–168, October 2001.

[8] N. Foster and R. Fedkiw, "Practical animation of liquids," in *ACM SIGGRAPH*, pp. 23–30, 2001.

[9] D. Q. Nguyen, R. Fedkiw, and H. W. Jensen, "Physically based modelling and animation of fire," in *ACM SIGGRAPH*, p. To Appear., 2002.

[10] D. Enright, S. Marschner, and R. Fedkiw, "Animation and rendering of complex water surfaces," in *ACM SIGGRAPH*, p. To Appear., 2002.

[11] 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.

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

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

[14] E. Lindholm, M. J. Kilgard, and H. Moreton, "A user-programmable vertex engine," in *ACM SIGGRAPH*, pp. 149–158, 2001.

[15] A. T. Inc. http://www.ati.com/developer/index.html.

[16] N. Corporation http://developer.nvidia.com.

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

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

[19] J. Hart, "Multipass programming for personal high-performance computing." NSF Grant 0113968, 2001.

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

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

[22] M. Corporation, "Direct3D." http://www.microsoft.com/directx.

[23] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan, "A real-time procedural shading system for programmable graphics hardware," in *ACM SIGGRAPH*, pp. 159–170., 2001.

[24] O. ARB, "OpenGL 2.0 shading language." http://www.opengl.org.

[25] N. Corporation, "Cg language." http://www.cgshaders.org/articlesCg_Specification.pdf.

[26] M. Corporation, "High level shading language." http://www.microsoft.com/directx.

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

[28] M. Grayson, "A short note on the evolution of surfaces via mean curvatures," *Journal of Differentail Geometry*, vol. 58, p. 555, 1989.

[29] N. Stewart, "OpenGL C++ toolkit." http://www.nigels.com/glt.

[30] SGI, "Open inventor." http://oss.sgi.com/projects/inventor/.