J. Parallel Distrib. Comput. 68 (2008) 1319–1328

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Advanced interactive medical visualization on the GPU

Natalya Tatarchuk[a,*], Jeremy Shopf[a], Christopher DeCoro[b]

[a] Game Computing Applications Group, AMD GPG (O-CTO), United States
[b] Princeton University, United States

ARTICLE INFO

Article history:
Received 14 March 2008
Received in revised form
24 June 2008
Accepted 25 June 2008
Available online 28 June 2008

Keywords:
Isosurface extraction
Volume rendering
Real-time rendering
Medical visualization
Isosurface rendering
Programmable shaders
GPU
DirectX10
Geometry shaders
Stream-out
Marching cubes
Marching tetrahedra
Ray casting
Ray marching
Transfer function editing

ABSTRACT

Interactive visual analysis of a patient's anatomy by means of computer-generated 3D imagery is crucial for diagnosis, pre-operative planning, and surgical training. The task of visualization is no longer limited to producing images at interactive rates, but also includes the guided extraction of significant features to assist the user in the data exploration process. An effective visualization module has to perform a problem-specific abstraction of the dataset, leading to a more compact and hence more efficient visual representation. Moreover, many medical applications, such as surgical training simulators and pre-operative planning for plastic and reconstructive surgery, require the visualization of datasets that are dynamically modified or even generated by a physics-based simulation engine.

In this paper we present a set of approaches that allow interactive exploration of medical datasets in real time. Our method combines direct volume rendering via ray-casting with a novel approach for isosurface extraction and re-use directly on graphics processing units (GPUs) in a single framework. The isosurface extraction technique takes advantage of the recently introduced Microsoft DirectX® 10 pipeline for dynamic surface extraction in real time using geometry shaders. This surface is constructed in polygonal form and can be directly used post-extraction for collision detection, rendering, and optimization. The resulting polygonal surface can also be analyzed for geometric properties, such as feature area, volume and size deviation, which is crucial for semi-automatic tumor analysis as used, for example, in colonoscopy. Additionally, we have developed a technique for real-time volume data analysis by providing an interactive user interface for designing material properties for organs in the scanned volume. Combining isosurface with direct volume rendering allows visualization of the surface properties as well as the context of tissues surrounding the region and gives better context for navigation. Our application can be used with CT and MRI scan data, or with a variety of other medical and scientific applications. The techniques we present are general and intuitive to implement and can be used for many other interactive environments and effects, separately or together.

1. Introduction

In many medical fields, the application of volume rendering and isosurface extraction for rapid and meaningful visual representation of datasets such as CT, MRI and PET scans can make an important difference in the speed of surgical planning, diagnosis, and treatment. It is also a useful tool in surgical simulation and medical education. Instead of training on real specimens, virtual endoscopy provides a convenient and cheap alternative to practice the course of the surgery and has the advantage of already providing a visualization of the real data, which makes exact pre-operative planning possible. This visualization is based on a 3D scan of the respective body region, like a computed tomography (CT) or magnetic resonance imagine (MRI) scan or a rotational angiography. The resulting data from one (or more) of these scans is visualized in a way that allows interior views of the dataset, mimicking the real environment as closely as possible.

Minimally invasive procedures have gained increasing importance in medical practice because of the – in many cases – faster (and thus cheaper) process, the often easier and less painful way in which inner organs can be reached and the faster recovery of patients, which reduces the overall risk and helps to keep clinical costs low. These procedures have proven particularly useful in surgery, neurosurgery, radiology, and many other fields. In most cases, these procedures are performed using an endoscope, a fiber optic of small diameter that serves as a light source, with a small camera and one or more additional tools attached to it.

Medical visualization for dataset analysis and applications such as virtual endoscopy imposes stringent requirements for any

* Corresponding author.
E-mail addresses: natalya.tatarchuk@amd.com, natashat@bu.edu (N. Tatarchuk), jeremy.shopf@amd.com (J. Shopf), cdecoro@cs.princeton.edu (C. DeCoro).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.06.011

virtual system used in this domain. Such a system must exhibit the following properties:

- High-quality rendering at interactive rates
- Ability to move the camera viewpoint into the dataset
- Strict on-demand extraction of isosurface for analysis and interaction with the virtual surgery tools
- Delivery of a coherent context for the extracted surfaces

Interactive visual analysis of a patient's anatomy by means of computer-generated 3D imagery is crucial for diagnosis, pre-operative planning, and surgical training. The task of visualization is no longer limited to producing images interactive rates; it also includes the guided extraction of significant features to assist the user in the data exploration process. An effective visualization module has to perform a problem-specific abstraction of the dataset, leading to a more compact and hence more efficient visual representation. Moreover, many medical applications, such as surgical training simulators and pre-operative planning for plastic and reconstructive surgery, require the visualization of datasets that are dynamically modified or even generated by a physics-based simulation engine.

Previously existing systems either strive for interactive rendering of isosurfaces alone, generate images via volumetric rendering by itself, or produce high-quality renderings offline that can later be viewed with no further possibility of interaction. Though existing approaches can typically render isosurfaces directly on GPU, they are not able to extract and re-use polygonal surfaces for additional interaction (such as collision detection with the simulated surgical tool) or analysis of surface features (tumor analysis). Additionally, interactive direct volume representations would be highly desirable due to additional expressiveness provided by semi-transparent surfaces and the possibility of visualizing objects of interest without prior segmentation of the dataset. To date, no comprehensive framework has been presented that is capable of delivering both isosurface extraction and volumetric rendering in a single package with sufficient quality at truly interactive frame rates.

In this paper we present a set of approaches that allow interactive exploration of medical datasets in real time. We implement our methods using massively parallel architectures available to average consumers — the latest commodity GPUs. Recent generations of consumer GPU architectures are designed for high efficiency and high computational load, along with effective use of coherency and extensive memory bandwidth requirements. Furthermore, with the advent of programmable GPU pipelines, we can take advantage of the massive parallelism of GPUs, rather than relying on significantly slower-scaling CPU multi-core architecture. In this work, we take advantage of the huge memory bandwidth available on the most recent generation of consumer GPUs to process extensive datasets, such as the Visible Human Project® dataset (taking up more than 576 MB of video memory) at extremely fast frame rates. We utilize efficient pixel and geometry processing, taking advantage of dynamic load balancing in the latest GPU architectures (such as ATI Radeon HD 2000 and beyond series), and build our system to fully utilize Shader Model 4.0 capabilities available with the DirectX 10 GPU pipeline.

Our system combines direct volume rendering via ray-casting with isosurface extraction directly on the GPU. The volume rendering approach is reformulated to take advantage of parallel pixel processing of the GPU pipelines. The isosurface extraction sub-system takes advantage of the novel DirectX 10 GPU pipeline for dynamic surface extraction in real-time. This framework is able to process individual voxel portions of the input dataset in parallel using geometry shaders. Our approach will scale with the number of parallel single instruction, multiple data (SIMD) and texture units in a GPU generation.

We have developed a technique for real-time volume data analysis by providing an interactive user interface for designing material properties for organs in the scanned volume. We provide an interactive user-driven material design system for quick and intuitive organ classification based on material properties of the dataset. Intelligently combining isosurface extraction with direct volume rendering in a single system allows for surface properties as well as for the context of tissues surrounding the region and gives better context for navigation. Our application can be used with CT and MRI scan data, or with variety of other medical applications.

The pipeline for our medical visualization system presented in this paper is as follows:

- We start by collecting the data. This is typically done by performing a CT or MRI scan on a patient. In our case, we simply used an existing data set from the Visible Human Project, collected by the U.S. National Institutes of Health National Library of Medicine.
- We then proceed to preprocess the data to compute gradients that are used for rendering correct lighting information at run time. This is done in an offline process.
- Once we have extracted and preprocessed the data, we can proceed to render it, using our on-demand isosurface extraction on the GPU and interactive volume rendering via ray-casting.
- At any point we can also interactively classify features by using our material user interface, designed to interactively specify, edit and save custom 2D transfer functions for each dataset.

2. Efficient isosurface extraction and rendering on GPU

An implicit surface representation, as opposed to explicit representation with a polygon mesh or parametric surface, is frequently the most convenient form of many modeling tasks. The high computational expense of extracting explicit isosurfaces from implicit functions, however, has made such operations a frequent candidate for GPU acceleration. Now, with recent advances in GPU architectures, we demonstrate the efficient implementation of an intuitive extraction algorithm using a hybrid marching cubes/marching tetrahedra approach. We are able to leverage the strengths of each method as applied to the unique computational environment of the GPU, and in doing so, achieve real-time extraction of detailed isosurfaces from high-resolution volume datasets. We are also able to perform adaptive surface refinement directly on the GPU without lowering the parallelism of our algorithm. In addition, we show that the complementary technique of inverse quadratic interpolation significantly improves surface quality.

Implicit functions are a powerful mechanism for modeling and editing geometry. Rather than the geometry of a surface given explicitly by a triangle mesh, parametric surface, or other boundary representation, it is defined implicitly by an arbitrary continuous function $f(x)$, $x \in R^3$. By defining an arbitrary constant c (referred to as an *isovalue*, and frequently 0), we can define our surface as the set of all points (*isosurface*) for which $f(x) = c$. For simplicity, we will frequently make the substitution $F(x) = f(x) - c = 0$, without loss of generality.

Apart from the area of medical visualization, deformable isosurfaces, implemented with level-set methods, have demonstrated great potential in visualization for applications such as segmentation, surface processing, and surface reconstruction. Specifically, these hold immense importance in the area of fluid dynamics and rendering applications ranging to simulating complex fluid flows around objects to detailed fluid dynamics for liquids interacting with objects. Isosurfaces are normally displayed using computer graphics, and are used as data visualization methods in computational fluid dynamics, allowing engineers to study features of a fluid flow (gas or liquid) around objects, such as aircraft wings. An isosurface may represent an individual shockwave in supersonic flight, or several isosurfaces may be generated showing a sequence of pressure values in the air flowing around a wing. Isosurfaces tend to be a popular form of visualization for volume datasets since they can be rendered by a simple polygonal model, which can be drawn on the screen very quickly. Numerous other disciplines that are interested in 3D data often use isosurfaces to obtain information about pharmacology, chemistry, geophysics, and meteorology.

The advantage of this representation is to allow arbitrary definition of the underlying function – and, thus, the implied surface – in a simple and direct manner. Such a representation can easily have arbitrary and dynamic topology, modified using Boolean and arithmetic operators, analyzed using traditional signal processing techniques, and used with simulations of natural phenomena. To perform these operations using traditional, explicit modeling would be impractically complex.

The disadvantage, however, is that such representations pose a challenge when rendering isosurfaces directly at real-time rates — especially given the rasterization pipelines used in GPUs, which are designed for triangle input data. Several classes of rendering techniques exist, such as *direct volume rendering* (volumes and implicit functions are synonymous in this context), which renders the volume without an intermediate boundary representation. Among such techniques, Westerman and Ertl [16] demonstrated a method for texture-based rendering of volume datasets defined on a uniform regular grid; later work proposed a generalized method for rendering volumes defined over tetrahedral grids [13].

While such techniques are limited to rendering applications, an alternate class of methods extracts an intermediate explicit isosurface (usually a triangle mesh) that can be used for further processing and analysis in addition to rendering. Examples include later use for collision detection, shadow casting, and animation. The most commonly used algorithms for isosurface extraction are derivatives of the *marching cubes* (MC) algorithm [9] and the closely related *marching tetrahedra* (MT) algorithm [14]. The algorithm we present is a hybrid of these two methods, such that we leverage the strengths of each method as applicable to the unique constraints and benefits of the GPU architecture.

Isosurface extraction is a compute-intensive method. Isosurface extraction on the GPU has been a topic of extensive research for the last several years. Much work has been done to generate highly efficient renderings at high frame rates. Prior to the DirectX10 generation of GPUs, the programming model lacked support for programmable primitive processing and the ability to output geometric quantities for later re-use. Thus, previous work lacked the ability to generate a polygonal surface directly on the GPU and re-use it for subsequent computation such as collision detection or optimization of volumetric rendering for surrounding organs. Much of the extraction work was redundantly performed regardless of whether the isovalue was dynamically changing or not, resulting in wasted computation. Nonetheless, a number of researchers succeeded at fast, interactive isosurface rendering.

Pascucci [11] rendered tetrahedra to the GPU, using the MT algorithm in the vertex shader to re-map vertices to surface positions. Subsequently, Klein et al. [7] demonstrated a similar tetrahedral method using pixel shaders, which at the time provided greater throughput. Other researchers instead implemented the marching cubes algorithm [3,6]. For an broad overview of both direct rendering and extraction methods, see the survey by Silva et al. [15].

All of these methods, however, were forced to use contrived programming techniques to escape the limitations of the previously restrictive GPU architecture. Using the geometry shader stage, executing on primitives post vertex shader and prior to rasterizer processing, we are able to generate and cull geometry directly on the GPU. This provides a more natural implementation of marching methods. With our approach the isosurface is dynamically extracted directly on the GPU and constructed in polygonal form and can be directly used post-extraction for collision detection or rendering and optimization. The resulting polygonal surface can also be analyzed for geometric properties, such as feature area, volume and size deviation, which is crucial for semi-automatic tumor analysis, for example, as used in colonoscopy. Our pipeline provides a direct method to re-use the result of geometry processing, in the form of a *stream-out* option, which stores output triangles in a GPU buffer after the geometry shader stage. This buffer may be re-used arbitrarily in later rendering stages, or even read back to the host CPU.

In our work, we present a hybrid method that leverages the strengths of both marching cubes and marching tetrahedra, relative to the unique abilities and constraints of the latest GPU architecture. In addition, we provide several complementary techniques that enhance the quality of the extracted surface. The contributions of our work are:

(1) Adaptive isosurface extraction optimized for the GPU programmable geometry pipeline;
(2) Improved resulting surface quality through quadratic root-finding while maintaining lower extraction grids for memory saving; and,
(3) Strict on-demand extraction of isosurface.

Our isosurface extraction pipeline (Fig. 2) starts by dynamically generating the voxel grid to cover our entire volume or a section of it. Using geometry shader and stream-out features, we tessellate the volume into tetrahedra on-the-fly. This allows us to adaptively generate and sample the grid based on the demands of the application. Each input voxel position is dynamically computed in the vertex shader. The geometry shader then computes six tetrahedra spanning the voxel cube. As an optimization, we only generate tetrahedra for voxels containing the isosurface, providing memory savings. Once we have the tetrahedra, we then use the marching tetrahedra algorithm to dynamically extract polygonal surface from our scalar volume consisting of material densities. In both passes for tetrahedral mesh generation and isosurface extraction, we use the geometry amplification feature of the geometry shader stage directly on the GPU.

We utilize the efficiency of parallel processing units on the GPU more effectively by separating isosurface extraction into two passes. Given a set of input vertices, a *geometry shader* program will execute in parallel on each vertex. Given that each individual instance of this program is, in fact, serial on a given SIMD unit, we maximize each effective SIMD utilization by separating extraction into two phases — fast cube tetrahedralization and a marching tetrahedra pass — reducing serialization of each individual geometry shader instance. Thus, we first execute on all vertices in our grid in parallel, generating tetrahedra, and then execute on each tetrahedra in parallel, generating polygons. This also allows the optimal balance between parallelization of polygonal surface extraction with efficient memory bandwidth utilization (the tetrahedra, exported by the first pass, consist of just three four-component floating point values).

2.1. Isosurface extraction using marching methods

Our method is based on both the *marching cubes* and the *marching tetrahedra* algorithms for isosurface extraction. The domain in R^3 over which F is defined is tessellated into a grid at an arbitrary sampling density. In both methods, for each edge $e = (x_0, x_1)$ in the tessellation, we will evaluate $F(x_0)$ and $F(x_1)$;

by the intermediate value theorem, if the signs of $F(x_0)$ and $F(x_1)$ differ, a isosurface vertex must intersect e. Considering all edges, we can connect vertices to form triangles.

Each possible assignment of signs to grid cells can be assigned a unique number. This is subsequently used to index into a lookup table, which will indicate the number and connectivity of isosurface triangles contained in the cell. In the marching cubes method, each cell has either vertices, and therefore there exist 2^8 possible assignments of the sign of $F(x)$. Each cube typically produces up to 6 triangles (as described in [9]). Therefore a straightforward lookup table holds $2^8 \cdot 6 \cdot 3 = 4,608$ entries. The size of this table presents an important consideration for parallel implementations on the GPU. The edge lookup tables are typically stored in SIMD-specific memory for efficient and coherent access by each shader program instance. However, constructing large tables may result in additional register pressure. This would significantly reduce the number of parallel threads simultaneously running on the GPU. Smaller lookup tables ensure higher order of parallelization because the GPUs are able to schedule a higher number of threads due to a higher number of available registers. Furthermore, Ning and Bloomenthal [10] have demonstrated that MC can generate incorrect connectivity (even if the inherent ambiguities are avoided by careful table construction). Therefore, straight marching cubes polygonization would need to handle the undesirable topology in a special-case manner for each geometry shader invocation, increasing serialization for each instance of the program and significantly impairing performance of the resulting algorithm.

The marching tetrahedra method, by its use of a simpler cell primitive, avoids these problems. There exist only 16 possible combinations, emitting at most two triangles, and no ambiguous cases exist. This tiny lookup table allows effective use of the fast access SIMD registers and results in much higher utilization of the parallel units on the GPU. One additional consideration for the related class of applications is that the cube is more intuitive for grid representation and adaptive sampling, with stronger correspondence to the original sampling. We note that the tetrahedron is an irregular shape and does not share these advantages. Straightforward tetrahedralization of a cube requires between 4 and 6 tetrahedra per cube, thereby requiring a corresponding factor of increase in the number of function evaluations required for the resulting mesh, if no sharing is done between primitives. To deal with this consideration, we introduce our hybrid method.

2.2. *Hybrid cubes-tetrahedra extraction*

The GPU architecture excels at large-scale parallelism; however, we must remember that the programmable units perform in *lock-step*, so we must carefully parallelize our computations for maximum performance.

The general strategy is to use marching cubes to exploit the additional information present in cubes, as opposed to tetrahedra. As we mentioned earlier, straightforward tetrahedralization is a relatively complex program in GPU terms, and would reduce thread parallelization. Rather than perform triangulation directly, it is preferable to adaptively tetrahedralize the input cubes. We perform final triangulation of the output surface using the simpler extraction operation on the tetrahedral grid. Our method uses the following steps:

Pass 1: Domain voxelization

(1) Dynamically voxelize the domain
(2) Tessellate cubes into tetrahedra near the surface
(3) Output tetrahedra as points to stream-out buffer

Pass 2: Marching tetrahedra

(1) Perform marching tetrahedra on generated tetrahedra
(2) Identify edges intersecting surface
(3) Fit a parabola on the edge and find root, by either:
    (a) Performing third function evaluation along edge
    (b) Using function gradients to estimate a parabola
(4) Output each isosurface triangle to a stream-out buffer for later re-use and rendering or straight to rasterization for immediate results

Voxelize input domain. In many cases (for example with volumetric data generated with medical imaging tools or physical simulations) the input data itself is specified on a regular (cubic) grid. Therefore, from the perspective of reducing function evaluations (corresponding to texture reads on the GPU) it is most practical to evaluate the function exactly at those points, and generate output triangles accordingly.

Although tetrahedral meshes provide a straightforward and efficient method for generating watertight isosurfaces, most preprocessing pipelines do not include support for directly generating tetrahedral meshes. Furthermore, we would like to support isosurface extraction on dynamic meshes and thus wish to generate tetrahedra directly on the GPU (for example, for particle or fluid simulations).

We start by rendering a vertex buffer containing n^3 vertices, where n is the grid size. Using automatically provided primitive ID, we generate voxel corner locations directly in the vertex shader. Subsequent geometry shader computes the locations of the voxel cube corners. We can then evaluate isosurface at each voxel corner.

Cube tetrahedralization. In the first pass' geometry shader we tessellate each cube voxel containing the surface into at most six tetrahedra. We can either re-use already computed isosurface values, or, to reduce stream-out memory footprint, simply repeat evaluation of tetrahedra corners in the next pass. Prior to tetrahedralization we compare the signs of the voxel corners to determine whether a given voxel contains the isosurface. Using the geometry shader's *amplification* feature, we dynamically generate tetrahedra for only those voxels that contain the isosurface. We output tetrahedra as point primitives into stream-out buffers, typically storing only the (x, y, z) components of tetrahedra corner vertices for efficient use of stream-out functionality.

Adaptive isosurface refinement. We have a number of options for adaptive polygonization of the input domain. We can perform an adaptive subdivision of the input grid, during the first pass of domain voxelization. While sampling the isosurface, we can choose to further subdivide the voxel, refining the sampling grid in order to detect new isosurface parts that were missed by the original sampling grid. This uses straightforward octree subdivision of the input 3D grid and will generate smaller-scale tetrahedra for the new grid cells on the finer scale if the original voxel missed the isosurface.

We can additionally add adaptive tetrahedra refinement in the subsequent pass during cube tetrahedralization at very little cost. This allows us to generate new sampling points inside the existing grid cells in which the isosurface has already been detected by the previous pass. In the six-tetrahedra tessellation used, each tetrahedron shares a common longest edge along the diagonal of the cube. At the cost of one function evaluation at the edge midpoint, we can perform a longest-edge subdivision for each of the six tetrahedra. We emit only those of the resulting 12 tetrahedra that contain the surface, which are a subset of those previously discarded whose signs differ from the sign of the center point. By performing one additional evaluation and at most six comparisons, we can perform a two-level adaptive simplification. Note also that as the subdivision is always to the center shared edge, the tetrahedra of this cell will be consistent with those of neighboring cells and avoid cracks in the output surface.

Marching tetrahedra. Using the *DrawAuto* functionality of DirectX 10, we render the stream-out buffer as point primitives in the next pass, performing the MT algorithm in the geometry shader stage. We identify the edges of the current tetrahedron that contain the output surface. As stated, MT is preferable for GPU implementation due to the significantly reduced lookup table sizes and output complexity. However, by the use of our hybrid method, which re-uses function evaluations from the initial cube grid, we can avoid redundancy; also, our adaptive subdivision step significantly reduces the number of primitives generated. Thus, our hybrid method is able to utilize the strengths of both methods as adapted to the GPU pipeline.

Fit a parabola along edges. While the MT rules identify the edges that will result in an output vertex, they do not specify *where* along that edge the vertex will lie.

For an edge $e = (x_1, x_2)$, the vertex v_e will fall along the line segment $x_1 + (x_2 - x_1)t$, $t \in [0, 1]$, and ideally, the choice of t is such that $F(v(t)) \approx 0$, so as to best approximate the isosurface. We can therefore consider finding the optimal vertex as a root-finding problem over t. The traditional approach uses the (linear) secant method,

$$t = \frac{-F(x_1)}{F(x_2) - F(x_1)}. \tag{1}$$

We have found, however, that significantly better visual quality results from using the inverse quadratic interpolation method (more familiar as a step of Brent's method [12]). This method fits a parabola to the function, and estimates the root by solving a quadratic equation. We illustrate the difference between the two methods in Fig. 3. Our quadratic root-finding method performs faster than applying an additional level of subdivision; in fact, it frequently produces better results than a corresponding increase in the sampling density. We provide two methods to fit such a parabola, according to the demands of the particular application.

Option 1: Perform a third evaluation per edge. The simplest way to fit a parabola along the edge is to perform a third function evaluation $F(x_3)$. The three points will then uniquely define the parabola. For a first approximation, we use the result of the secant method for x_3. Solving the quadratic defined by the points, we have:

$$\begin{aligned} v = {} & \frac{F(x_2)F(x_3)}{(F(x_1) - F(x_2))(F(x_1) - F(x_3))}x_1 \\ & + \frac{F(x_1)F(x_3)}{(F(x_2) - F(x_1))(F(x_2) - F(x_3))}x_2 \\ & + \frac{F(x_1)F(x_2)}{(F(x_3) - F(x_1))(F(x_3) - F(x_2))}x_3. \end{aligned}$$

Because this additional evaluation is only performed along edges that are known to output a vertex, we are able to selectively restrict these additional evaluations to locations in which they are useful rather than simply increasing the function grid size, which would lead to superfluous evaluations.

As we optimize the grid tessellation in previous phases, an additional evaluation is not a bottleneck in our implementation, and thus provides additional quality with no performance penalty. Should this be a limitation for certain applications, we propose an alternate method that avoids additional evaluations.

Option 2: Estimate parabola with gradients. Frequently, the function gradient $\nabla F(x)$ is either known, or can be computed easily along with $F(x)$. With static volume textures, ∇F is often computed beforehand, and stored with F in a single RGBA texture. Similarly, with common implicit functions, such as metaballs, the additional work required to compute the gradient is minimal. Finally, many applications will already be required to evaluate the gradient for use as a shading normal. In all such cases, we can use the gradients to estimate a parabola, obviating the need for the additional function evaluation.

We seek a parabola that interpolates both endpoints, and makes the best least-squares approximation to the gradients at each endpoint. We can restrict the problem the to line $v(t) = x_1 + (x_2 - x_1)t$, defining $F'(x)$ as the directional derivative $D_{x_2 - x1}F(x) = \nabla F(x) \cdot (x_2 - x_1)$. The class of parabola interpolating the endpoints, and its derivative, is:

$$F(t) = F(x_1) + (F(x_2) - F(x_1) - b)t + bt^2 \tag{2}$$

$$F'(t) = F(x_2) - F(x_1) - b + 2bt. \tag{3}$$

We seek a choice of b that minimizes the least-squares error between the function derivatives relative to the actual derivatives, where the relationship is given by the following system of equations:

$$F'(x_1) \approx F(x_2) - F(x_1) - b \tag{4}$$

$$F'(x_2) \approx F(x_1) - F(x_2) + b. \tag{5}$$

The least squares solution is the average of both solutions, or $b = 0.5(F'(x_2) - F'(x_1))$. We can now solve Eq. (2) directly using the quadratic equation, which is guaranteed to have exactly one root in the interval. Note that if $F'(x_1) = F'(x_2)$, this is equivalent to performing the linear secant method.

Storing extraction results for subsequent re-use. We can intelligently generate isosurface on demand, either as a function of the implicit domain changes or when the user modifies isovalue, using our material editor interface. After computing MT, we can output isosurface triangles into a GPU stream-out buffer for re-use in the later passes or even storage on disk. This capability is a critical feature of our method that is enabled by the latest GPU pipeline. While the extraction already runs at real-time rates, the actual frame rate perceived by the user will be dramatically faster, as most frames are able to re-use the geometry from the stream-out buffer. This frees up GPU resources for additional computation, such as, for example, combining high-quality direct volume rendering with isosurface rendering for better context guidance during medical training simulations (as seen in Fig. 1). Furthermore, we utilize the extracted polygonal surface to improve the rendering speed of our volumetric renderer. We render the isosurface faces as the starting positions for ray-casting in the direct volume rendering algorithm (as an optimization for ray-casting). The isosurface can also be rendered directly into a shadow buffer for shadow generation on the surrounding organs.

3. Direct volume rendering

To provide context to our isosurface visualization, we render the surrounding data directly. This is achieved by casting rays from the viewer, through each pixel of the screen, and sampling the volume data at a constant rate. We refer the reader to the seminal direct volume rendering papers by Drebin et al. [1] and Levoy [8] for more in-depth discussion.

Our implementation of direct volume rendering is a GPU ray-casting method based on the work of Krueger and Westermann [4]. The algorithm begins by calculating ray directions in normalized texture coordinate space. This is performed by rendering the bounding box with a per-vertex color equal to the texture coordinate at that corner. First, the back faces of the bounding box are rendered to a screen sized texture to determine the exit point $Coord_{back}$ of the ray for each pixel. Subsequently, the front faces of the bounding box are rendered to texture to determine the entry

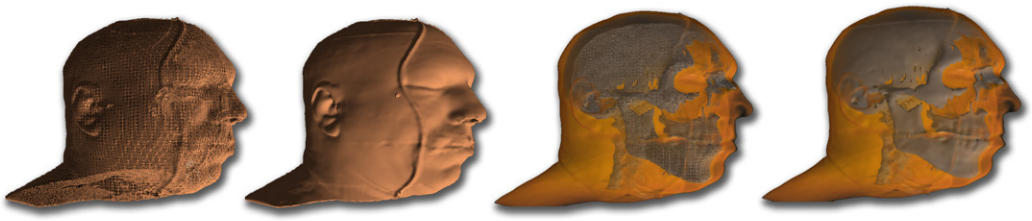


Fig. 1. We show the result of extracting a series of highly detailed isosurfaces at interactive rates. Our system implements a hybrid cube-tetrahedra method, which leverages the strengths of each as applicable to the unique architecture of the GPU. The left pair of images (wireframe and shaded, using a base cube grid of 64^3) show only an extracted isosurface, while the right pair displays an alternate isosurface overlaid with a volume rendering.

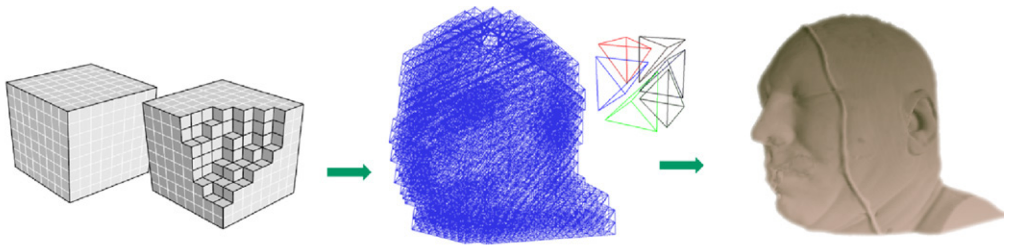


Fig. 2. Isosurface extraction pipeline.

point $Coord_{front}$ (See Fig. 4 for an example of these textures). The ray direction in texture coordinate space for each pixel can then be calculated as $Coord_{back}$- $Coord_{front}$. Ray marching is then performed in a pixel shader by marching a ray from $Cooord_{front}$, advancing by a fixed step size, for n steps. n is the number of steps required to advance to the exit point (length($Coord_{back}$- $Coord_{front}$) / $step\_size$).

At each ray marching step, the absorption of color by the material at the current volume location of the ray must be accounted for. The color of the current location in the volume is modulated by the accumulated opacity of the ray and the opacity of the location. The result is added to the accumulated color. The following equations model describe this operation:

$$C_a = C_a + (1.0 - \alpha_a)\alpha(\mathbf{x})C(\mathbf{x})$$
$$\alpha_a = \alpha_a + (1.0 - \alpha_a)\alpha(\mathbf{x}),$$

where $C(\mathbf{x})$ and $\alpha(\mathbf{x})$ are the color and opacity at volume location $\mathbf{x}$, and C_a and α_a are the accumulated color and opacity for the ray. The mapping of scalar volume data to color and opacity is described in Section 3.2.

3.1. Lighting model

The simplest volume rendering model assumes an emission-absorption model. Emission-absorption models ignore effects such as shadowing and multiple scattering. It is possible to use only an albedo value for emission, but we chose to model emission with a local lighting model ($C(\mathbf{x})$ below) to highlight surfaces and features present in the data.

$$\begin{aligned} C(\mathbf{x}) &= k_a A(\mathbf{x}) + k_d((1.0 - G)A(\mathbf{x}) \\ &\quad + G(1.0 - k_a)(N(\mathbf{x}) \cdot L * .5 + .5)) + k_s(N(\mathbf{x}) \cdot H)^n G \end{aligned}$$

$A(\mathbf{x}) =$ albedo

$N(\mathbf{x}) =$ gradient of scalar at x

$H =$ halfway vector between view and light directions

$k_a =$ ambient reflection coefficient

$k_d =$ diffuse reflection coefficient

$k_s =$ specular reflection coefficient

$n =$ specular exponent

$G =$ normalized gradient magnitude [0.0, 1.0].

We chose a warped Lambertian diffuse lighting model. This warped diffuse term scales and biases the dot product of the gradient and the lighting direction to 0.0 to 1.0 instead of -1 to 1. This allows all regions to be visible regardless of current lighting direction. We also add a specular term that is weighted by the gradient magnitude. We only want specular contributions in areas of high gradient. Additionally, we linearly interpolate between the albedo and diffuse color based on the gradient magnitude, as described by [5]. This places less visual importance on homogeneous regions and more on features identified by varying gradient magnitude.

3.2. Transfer function specification

To allow the user to interactively classify data, we provide an intuitive interface to map scalar volumetric data to color and opacity values. This mapping is referred to as a *transfer function*. An interactive and intuitive transfer function editor is an important part of the classification process.

All transfer functions must have at least one dimension, usually density. We have chosen a 2D transfer function based on density and gradient magnitude. Gradient magnitude can be thought of as the measure of "surfaceness" of a location. This extra dimension allows for more flexibility. For example, brain matter may have the same density as the outer dermis, but the gradient magnitude allows the user to assign different colors and opacities to regions with these properties.

Our user interface (shown in Fig. 5 consists of several scalable and translatable box widgets overlaid on a histogram of the volume data. The box widgets correspond to a distribution function and have an associated opacity and color that is defined by user selection. At each frame of rendering, the distribution function of each widget is multiplied by the associated color and opacity and rendered into a 2D lookup texture. This 2D texture (Fig. 6) defines the transfer function. At each volume sample taken while ray marching, this texture is indexed to map density values to color and opacity.

3.3. Aliasing reduction

According to the Nyquist theorem, our sampling rate must be twice the highest frequency in the data to reconstruct the original

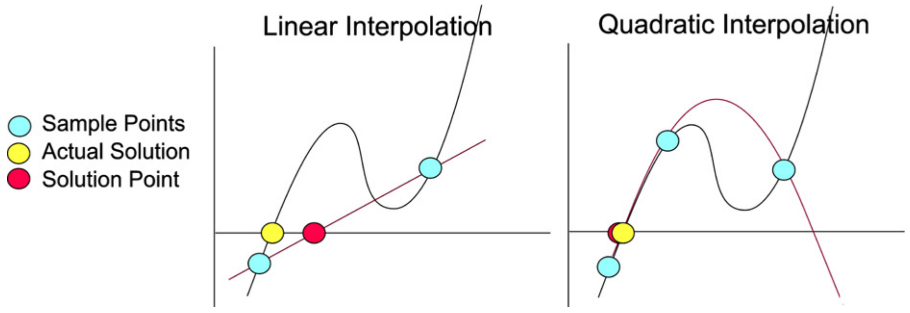


Fig. 3. **Linear vs. Quadratic Root-finding.** The traditional approach to finding the intersection point of the isosurface is to linearize the implicit function, and solve for the root (left). By taking an additional sample, so as to fit a parabola and find its root (right), we can make a better approximation of the actual zero-crossing.

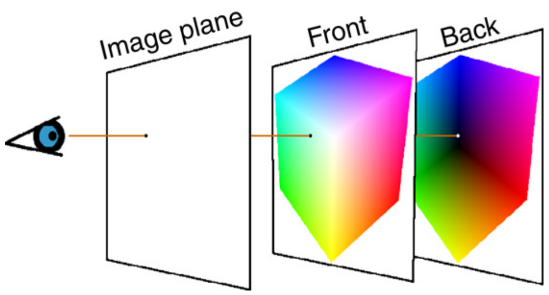


Fig. 4. Ray directions are calculated in texture coordinate space from the texture coordinates of the front and back faces of bounding geometry.

signal. It would be sufficient to sample the data at twice the span of a voxel. However, the transfer function and lighting calculations introduce additional high frequencies.

Pre-integrated volume rendering approaches such as that presented by Engel [2] provide solutions for correctly integrating these additional frequencies. However, the original pre-integrated volume rendering approach handles only a one dimensional transfer function and no lighting. There is no efficient way to pre-integrate the additional dimensionality of a 2D transfer function and a lighting function.

To reduce the visual impact of the aliasing that arises due to under-sampling, we introduce a stochastic sampling technique to our ray-caster. By offsetting the start location of each ray by a small random amount, we substitute banding artifacts for less visually distracting high-frequency noise. Note that this high-frequency signal is noticeable only during extreme close-ups. Fig. 7 illustrates the visual benefit of jittering rays.

3.4. *Incorporating isosurface for ray-casting optimization*

The purpose of volume rendering the data surrounding the extracted isosurface is to provide context. In our application, we treat the extracted surfaces as opaque. Thus, we can safely terminate our rays at the isosurface. This is achieved by rendering the extracted isosurface's 3D texture coordinates into the exit point texture $Coord_{back}$. An example exit point texture is provided in Fig. 8.

This straight-forward integration of the extracted polygonal surface as the ray exit points results in unpleasant aliasing artifacts for the volume rendering. This is caused by the fact that during the

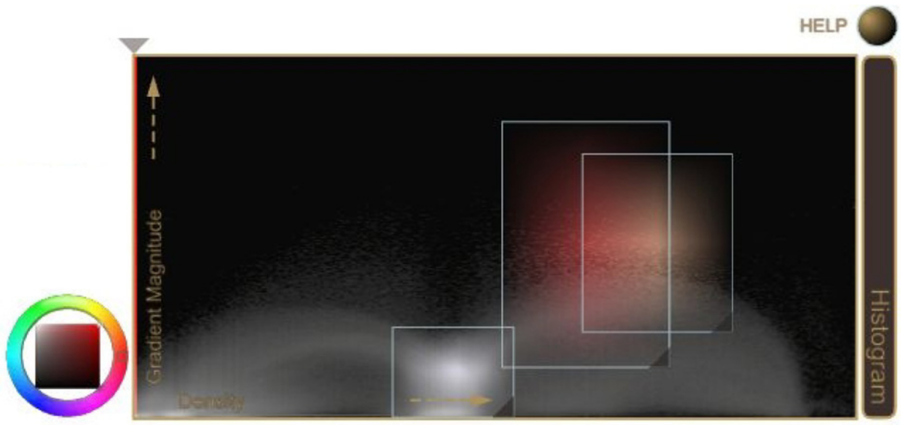


Fig. 5. Our transfer function editing interface.

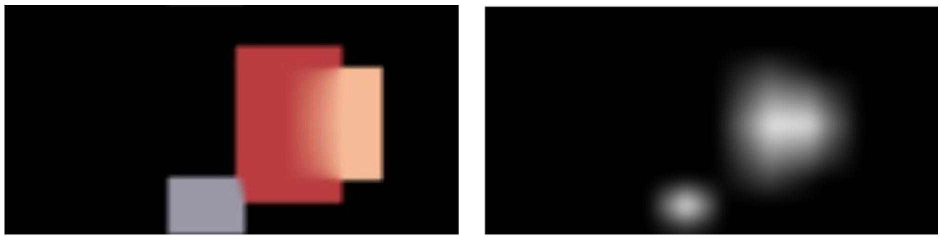


Fig. 6. **Left:** Transfer function color look-up texture. **Right:** Opacity look-up texture.

Table 1
Timing results for continuous isosurface extraction including CPU and GPU timing comparisons

Dataset	Grid		Time (ms)	Faces	FPS	Face/sec	GPU:CPU
Head	64^3	GPU	6.3	122 K	158 fps	19.2 M	9.9:1
($c = 0.47$)		CPU	66.8	130 K	15 fps	1.95 M	
	32^3	GPU	2	26 K	489 fps	13 M	4.5:1
		CPU	9.3	27 K	107 fps	2.91 M	
Thorax	64^3	GPU	8.07	192 K	124 fps	24 M	9.9:1
($c = 0.3$)		CPU	81	197 K	12.4 fps	2.43 M	
	32^3	GPU	2.5	40 K	400 fps	16 M	5.3:1
		CPU	12.3	37.2 K	81.5 fps	3.04 M	
Abdomen	64^3	GPU	8.5	192 K	116 fps	22.3 M	9.4:1
($c = 0.3$)		CPU	79.9	189 K	12.52 fps	2.37 M	
	32^3	GPU	3	42 K	337 fps	14.1 M	4.3:1
		CPU	12.4	40.1 K	81 fps	3.27 M	

See the supplementary video (http://ati.amd.com/developer/gdc/2007/MedViz-SiggraphMovie-H.264.mov) for more results.

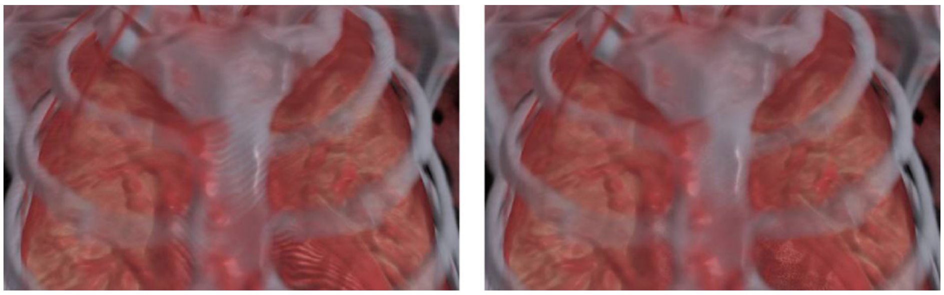


Fig. 7. Jittering ray starting positions reduces the visual impact of aliasing.

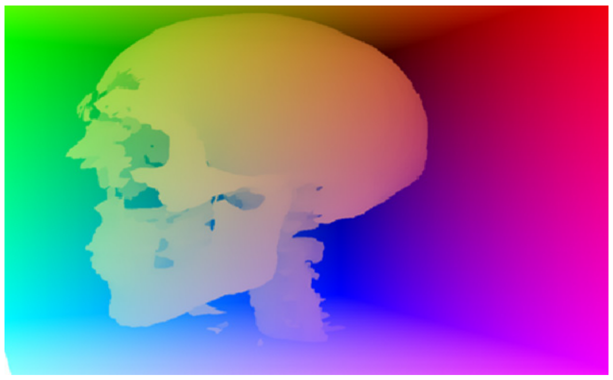


Fig. 8. Incorporating the isosurface into the volume rendering is achieved by rendering the isosurface into the exit point texture, as described in Section 3.

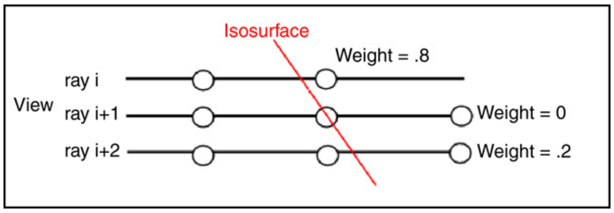


Fig. 9. Using a fixed sampling rate requires the last sample to be weighted by the fraction of the last step that is outside of the isosurface.

ray-casting computation, while computing each ray intersection for a given marching step, the new exit point may simply be missed. The isosurface coordinates may not lie exactly on the sample point of a marching ray. This will cause banding artifacts along the interface between volume rendering and isosurface rendering. To counteract the last incorrect sample, which will be inside the surface, we weight it by the fraction of the step that is outside of the surface (Fig. 9).

4. Results and conclusions

We collected results for continuous isosurface extraction and volume rendering for all seven sections of the Visible Human Project dataset (as shown in Table 1). Each section contains 256^3 samples on a regular grid. Storing the density and gradient information for each dataset portion results in a 576 MB memory footprint of 3D textures. This dataset can be rendered in its entirety on the GPU. However, due to input data resolution discrepancy, we chose to render the subsets as separate datasets. We used several high-resolution off-screen buffers for rendering ray marching front and exit points as well as some intermediate information, taking up 120 MB of V-RAM. The stream-out buffer used for storing the polygonal surface for the extracted isosurface is 85 MB.

4.1. Isosurface extraction results and analysis

Previous methods for GPU-based isosurface extraction have been forced to use contrived implementations to escape the limitations of the earlier programmable graphics pipelines. Such methods, while often performant, are complex to re-implement and modify and typically do not support re-use without significant effort.

We have shown that, with the availability of the latest-generation GPU architectures, it is possible to take advantage of the massive parallelism available on the GPU by implementing flexible and reasonable implementations of marching methods, while maintaining optimal performance characteristics. We found that our hybrid method of dynamic domain voxelization followed by a tetrahedralization pass results in high-performance and high-quality results (another example in Fig. 11). In our development of this system, we have explored various types of methods, including using standard marching cubes or marching tetrahedra directly. We settled on the final hybrid algorithm presented here after extensive testing and analysis of performance bottlenecks. The direct implementation of marching tetrahedra requires

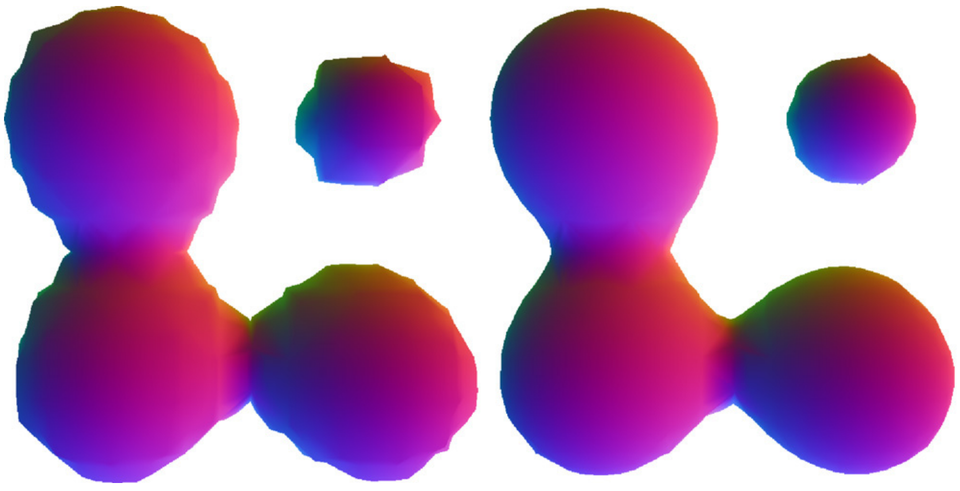


Fig. 10. Results comparison of root-finding methods. Linear secant method (left) results in surface missing important fine-grain detail (shown by the rough silhouettes of the smooth input metaballs). Quadratic root finding (right) results in a significantly higher-detail (and, thus, in this case, smoother) surface as compared to the linear secant method.

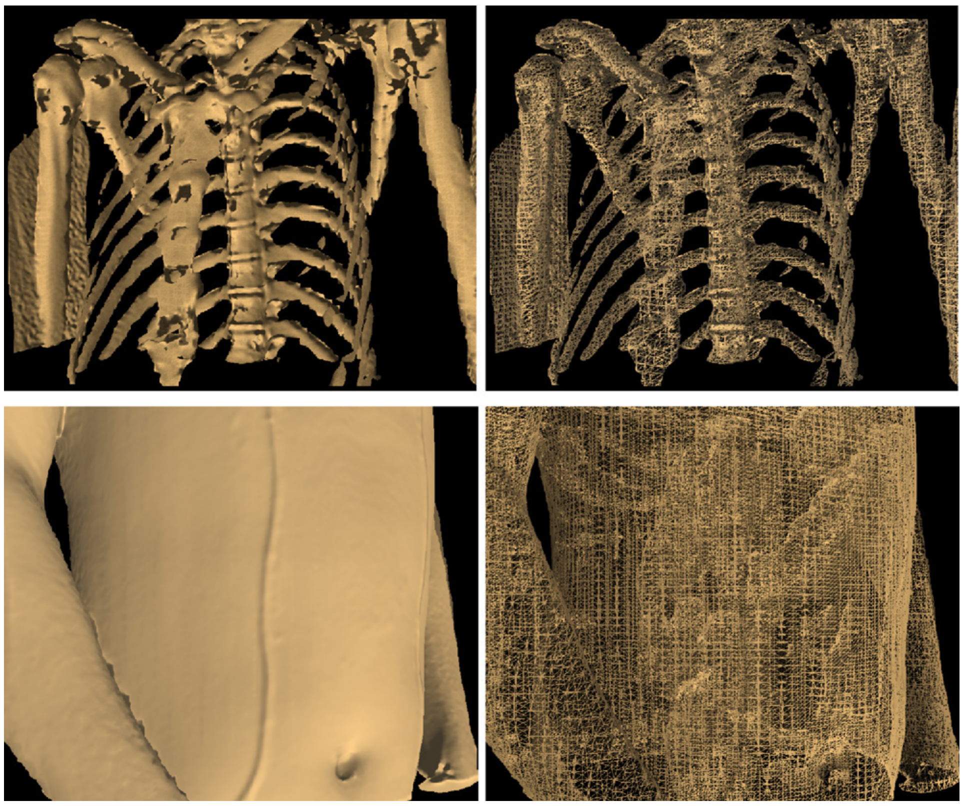


Fig. 11. Isosurface extraction on the Visible Human project dataset. These images show examples of isosurface extraction on two volumetric datasets from the Visible Human Project, each containing 256^3 slices of density data, extracted on a 64^3 grid.

large amounts of redundant isofunction computations, reducing parallelization. The marching cubes performance was strongly reduced by the large look-up tables sizes, as well as extraneous computations for incorrect topology fix-up. Therefore, this hybrid approach proved to be an excellent tradeoff between the two.

An earlier implementation performed the tetrahedral tessellation and surface extraction in a single pass. However, this severely limited parallelism. The difference in running time between a cube that is culled, and one that is both tetrahedralized and used for isosurface extraction in a single pass, creates a significant bottleneck for GPU resources. Instead, by moving the marching tetrahedra computation into another pass, we fully utilize GPU programmable units parallelization for voxelizing the domain and generating tetrahedra near the surface. Similarly, the marching tetrahedra pass exhibits the same advantages.

All timing results include resulting polygonal surface rendering cost and were collected on a Microsoft(R) Windows(R) Vista SP1 PC with AMD Athlon (TM) 64 X2 Dual-Core processor running at 2.4 GHz with 2 GB RAM and an ATI Radeon (TM) HD 4870 graphics card.

While a common approach to smoothing extracted isosurfaces is to use linear root-finding, our quadratic root-finding approach produces significantly higher quality visual results (Fig. 10). In our example surface detail quality is measured by surface smoothness. However, for different datasets, using this approach results in higher quantity of surface details recovered (thus, not necessarily a smoother surface). This additional computation has very minimal overhead, requiring less than 30 additional scalar ALU operations in the marching tetrahedra extraction shader, and an additional texture fetch, and, as such, having a very slight impact on performance. Note that the quality of quadratic root-finding for recursion depth $d = 8$ exceeds that for linear root-finding for several additional levels of subdivision ($d = 10$).

4.2. *Volumetric rendering via ray-casting results*

Our ray-caster is implemented utilizing Shader Model 3.0 functionality. We dynamically compute per-step lighting results, integrating the resulting illumination using on the order of 600 ray marching steps for each ray. We found that the limiting factors for the ray-caster performance are the application resolution (resulting in higher fill rates for higher resolution) and the number of steps taken for each ray computation. Integrating isosurface mesh as our ray exit points allows us significantly accelerate the resulting rendering, often by an order of magnitude in speed for given viewpoints.

4.3. *Conclusions and future work*

We have presented a set of approaches that allows interactive exploration of medical datasets in real-time. Our technique is based on combining direct volume rendering via ray-casting with isosurface extraction on the GPU. The latter takes advantage of the programmable DirectX 10 pipeline for dynamic surface extraction in real-time using geometry shaders. We have optimized our algorithm to take advantage of the massively parallel GPU architecture, while tuning it to the strengths and constraints of this model. Additionally we have developed a technique for real-time volume data analysis by providing an interactive user interface for designing material properties for organs in the scanned volume. Combining isosurface with direct volume rendering allows for visualization of surface properties enhanced by the context of tissues surrounding the region and gives better context for navigation. Our resulting application is both easy to use and results in high frame rates. We hope to see these techniques propagate in standard applications for dataset analysis for patients and surgery planning.

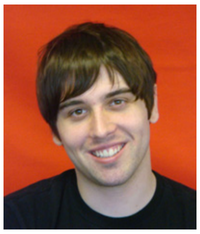
Acknowledgments

We thank Daniel Szecket for his work on the user interface used for our system. We also extend special thanks to Dan Abrahams-Gessel and the Game Computing Applications group, as well as the anonymous reviewers for their help, suggestions and review of this work.

References

[1] L.C.R.A. Drebin, P. Hanrahan, Volume rendering, Computer Graphics 22 (4) (1988) 65–74.
[2] K. Engel, M. Kraus, T. Ertl, High-quality pre-integrated volume rendering using hardware accelerated pixel shading, in: Workshop on Graphics Hardware.
[3] F. Goetz, T. Junklewitz, G. Domik, Real-time marching cubes on the vertex shader, in: Proceedings of Eurographics 2005, 2005.
[4] R.W. Jens Krueger, Acceleration techniques for GPU-based volume rendering, IEEE Visualization (2003) 287–292.
[5] C.H.P.S. Joe Kniss, Simon Premoze, A. McPherson, A model for volume lighting and modeling, IEEE Transactions on Visualization and Computer Graphics 9 (2) (2003) 150–162.
[6] G. Johansson, H. Carr, Accelerating marching cubes with graphics hardware, in: CASCON '06: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research, ACM Press, New York, NY, USA, 2006, p. 39.
[7] T. Klein, S. Stegmaier, T. Ertl, Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids, pg 00 (2004) 186–195.
[8] M. Levoy, Display of surfaces from volume data, IEEE Computer Graphics and Applications 8 (3) (1988) 29–37.
[9] W.E. Lorensen, H.E. Cline, Marching cubes: A high resolution 3D surface construction algorithm, in: Computer Graphics (Proceedings of SIGGRAPH 87), vol. 21, Anaheim, California, 1987, pp. 163–169.
[10] P. Ning, J. Bloomenthal, An evaluation of implicit surface tilers, IEEE Computer Graphics and Applications 13 (6) (1993) 33–41.
[11] V. Pascucci, Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping, in: Proceedings of VisSym 2004, 2004.
[12] W.H. Press, S.A. Teukolsky, W.A. Vetterling, B.P. Flannery, Numerical Recipies in C++, Cambridge University Press, 2002.
[13] S. Rottger, M. Kraus, T. Ertl, Hardware-accelerated volume and isosurface rendering based on cell-projection, in: VIS '00: Proceedings of the Conference on Visualization '00, IEEE Computer Society Press, Los Alamitos, CA, USA, 2000, pp. 109–116.
[14] P. Shirley, A. Tuchman, A polygonal approximation to direct scalar volume rendering, SIGGRAPH Comput. Graph. 24 (5) (1990) 63–70.
[15] C. Silva, J. Comba, S. Callahan, F. Bernardon, A survey of GPU-based volume rendering of unstructured grids, in: 17th Brazilian Symposium on Computer Graphics and Image Processing.
[16] R. Westermann, T. Ertl, Efficiently using graphics hardware in volume rendering applications, in: SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, ACM Press, New York, NY, USA, 1998, pp. 169–177.

Natalya Tatarchuk is a graphics software architect and a project lead in the Game Computing Application Group at AMD Graphics Products Group Office of the CTO. Her passion lays in pushing hardware boundaries investigating innovative graphics techniques and creating striking interactive renderings. She works closely with AMD's hardware architects as well as leading developers in the graphics community. In the past she has also been the lead for the tools group at ATI Research. Prior to that Natalya worked on 3D modeling software, and scientific visualization, among other projects. She has published papers in various computer graphics conferences and articles in technical book series such as ShaderX and Game Programming Gems, and has presented talks at SIGGRAPH and at Game Developers Conferences worldwide. Natalya holds BAs in Computers Science and Mathematics from Boston University and an M.S. in Computer Science from Harvard University.



Jeremy Shopf is a senior software engineer in the Game Computing Application Group at AMD Graphics Products Group (O-CTO) where he works on graphics demos and novel rendering techniques as part of the demo team. Prior to working at AMD, Jeremy was a graduate student researching perceptually driven rendering techniques as a member of the VANGOGH research lab at the University of Maryland Baltimore County.

Christopher DeCoro is a Ph.D. candidate in Computer Science at Princeton University. His current work focuses on control and approximation of rendering algorithms; allowing creative and artistic flexibility over both appearance and rendering detail. He also investigates techniques in geometry approximation, material representation, and applications of data-driven classification algorithms, including problems of computer vision and music information retrieval. Christopher was awarded the ATI Research Fellowship in 2007–2008; he also interned at the 3D Applications Research Group of ATI Research, where he explored novel applications of emerging GPU architectures. He obtained his Master's degree in Computer Science from Princeton University in 2004, subsequent to earning his Bachelor's degree from the University of California, Irvine, in 2002.