PERGAMON

Computers & Graphics 26 (2002) 209–218

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

Visualization of very Large Datasets

Improving progressive view-dependent isosurface propagation

Zhiyan Liu*, Adam Finkelstein, Kai Li

Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544, USA

Abstract

Recently, we proposed a new isosurface extraction algorithm that extracts portions of the isosurface in a view-dependent manner by ray casting and propagation. The algorithm casts rays through a volume to find visible active cells as seeds and then propagates their polygonal isosurface into the neighboring cells. Small pieces of the isosurface are generated by distance-limited propagation and joined together to form the final surface. This paper presents our evaluation of several design choices of the algorithm. We have implemented these design choices and showed that by making right design decisions, we can substantially reduce the time to obtain most (such as 99.9%) of the isosurface. © 2002 Elsevier Science Ltd. All rights reserved.

Keywords: Data visualization; Isosurface extraction; View dependent; Level of detail; Ray casting

1. Introduction

Applications such as large-scale simulations typically generate scalar fields and store them as volumetric datasets. A 3D scalar field F can be represented by a volumetric dataset that has a set of data points and the corresponding scalar values sampled at each point in the set. To visualize the scalar field, a known method is to display isosurfaces where $F(x, y, z) = v$ for a given threshold v. To visualize the isosurfaces of massive datasets, the challenge is to develop an algorithm that extracts the isosurfaces efficiently, requires modest rendering power, and supports interactive, adaptive-resolution visualization on a high-resolution display system.

Much work has been done on extracting isosurfaces, but all the existing algorithms have certain drawbacks. The *marching cubes* [1] algorithm visits all n cells in the dataset and triangulates the isosurface in each *active* cell, i.e. a cell that has values above and below the given threshold. Marching cubes are simple and straightforward, but examining all the cells in the dataset can be unnecessarily time-consuming. Several subsequent algorithms reduce the time spent on finding the cells that intersect the isosurface. Wilhelm and Van Gelder used an *octree* [2] to leverage object–space coherence and discard sections of the dataset before examining them. Cignoni et al. proposed the *interval-tree* method [3] and Livnat et al. proposed to use *span space* [4]. In preprocessing, both these algorithms sort all the cells according to minimum and maximum values and construct a search tree; then, for a given threshold, these methods search the tree to find all the active cells. Itoh and Koyamada used the *extrema graph* [5] approach to find seeds on the isosurface and propagate from these seeds. In the worst case, the seed set could have size $O(n)$. Bajaj et al. described the *contour trees* method [6] for finding small seed sets for isosurface traversal.

Although these methods dramatically improve on the original marching cubes algorithm, they do not try to avoid generating occluded polygons, nor do they manage the level of detail. As a result, they all generate the complete isosurface at the finest data resolution (one voxel). For very large datasets, generating and rendering the whole isosurface will prevent users from viewing the dataset at an interactive frame rate. Extraction can be slow, and the sheer number of polygons in the isosurface may overwhelm the hardware-rendering capabilities.

*Corresponding author. Tel.: +1-609-258-5030 fax: +1-609-258-1771.

E-mail address: zhiyan@cs.princeton.edu (Z. Liu).

0097-8493/02/$ - see front matter © 2002 Elsevier Science Ltd. All rights reserved.
PII: S0097-8493(02)00052-3

Two isosurface visualization algorithms were recently proposed to generate only the visible portions of the surface. Parker et al. proposed a *ray-casting* algorithm for isosurface extraction [7] that intersects viewing rays with the data volume and then computes the isosurface without generating an intermediate polygonal representation. For each ray intersecting the isosurface, a cubic equation is solved to find the normal at the intersection point. This approach is simple and requires no special rendering hardware. The authors have parallelized the algorithm to run on a 128-processor SGI Origin shared-memory multiprocessor to offer interactive frame rates for a 512×512 display. However, the running time of the algorithm is proportional to the number of pixels in the display, it is therefore not well-suited for high-resolution displays.

Livnat and Hansen described WISE as a *view-dependent* isosurface extraction algorithm that uses hierarchical tiles and shear-warp factorization for visibility testing, which then renders the polygons utilizing the graphics hardware [8]. They also used a 512×512 display. Traversing the dataset in a front-to-back order, (meta) cells are projected on the screen and tested against the current screen coverage map for visibility in software. Occluded (meta) cells are discarded. Visible meta-cells are examined recursively. All the triangles inside a visible cell are extracted and sent to the graphics hardware, and the current screen coverage map is updated accordingly. When the resolution of the display increases, both the coverage map calculation time and the space requirement for the hierarchical visibility mask will grow proportionally. Very recently, Livnat and Hansen have proposed SAGE [9], a view-dependent algorithm that improves on the performance of WISE.

Recently, we proposed a new hybrid algorithm that shares several features of the existing acceleration methods [10]. The main idea is to use ray casting into an octree as a way to identify visible seed cells (rather than computing the complete isosurface as in the method of Parker et al. [7]) and then use propagation (as in [5]) to extend the isosurface from the seed cells. Unlike previous propagation methods that propagate to the whole isosurface, our method uses a cut-off angle and some viewing criteria to decide when and where to stop the propagation for each seed cell. The isosurface pieces will then be patched together to form a view-dependent region of the isosurface, which includes all the triangles that are visible and a small number of occluded triangles that are near the visible surface. We have demonstrated with the $512 \times 512 \times 209$ Visible Woman head CT data, that our implementation could extract 99.5% of the visible isosurface quickly by casting only $<1\%$ of the rays on a 1600×1200 display. The important feature of the algorithm is, it is progressive and resolution-insensitive. An interesting question is how to enhance this algorithm to make it substantially more progressive and resolution-insensitive.

This paper investigates this issue by first analyzing the algorithm we recently proposed and then evaluating several key design choices. The design choices we have compared include performing exact vs. approximate intersection calculations, using cut-off angle distance vs. using a maximum count to limit a propagation, different values of the maximum count, three different orders of ray casting, using or not using a screen bounding box, and caching vs. calculating interpolants and triangles. Our evaluation shows that some design choices and parameters affect the performance of the algorithm significantly, whereas some do not. By choosing the best design choices, we can substantially reduce the time to obtain most (e.g. 90%, 99%, or 99.9%) of the isosurface. Our experiments with the same dataset show improvements by more than a factor of 2.4.

The rest of the paper is organized as follows. Section 2 describes the algorithm and several design choices. Section 3 presents experimental comparisons of the design choices. Section 4 shows the final result. Section 5 compares our results with other approaches and Section 6 summarizes our study.

2. The algorithm and design choices

The proposed algorithm may be viewed as an extension to the propagation method [5]. Currently, it works with structured rectilinear datasets. The main contribution is to make the propagation algorithm view dependent in a manner that is both efficient and incremental, while supporting adaptive-resolution visualization.

For the convenience of the description, let us first define the active cell as follows. Given a threshold v, we mark all the data points in the dataset with one of the two signs: "$+$" indicates that the scalar value at that point is above the threshold, while "$-$" indicates that the scalar value is below the threshold. We only consider the non-degenerated case where no one data point has exactly the value v. If a cell has vertices of different signs, then it is called an active cell, and the isosurface of threshold v will intersect this cell.

A key observation the propagation approach made was that if the vertices on a face of an active cell do not have the same sign, then the neighboring cell that shares the same face is also active. Therefore, the isosurface can be extended into the neighboring cell. This means that once an active cell is found as the seed, propagating the isosurface from that cell is efficient because one can avoid touching and examining inactive cells. However, neither the extrema graph nor the contour trees algorithm generates seeds that are guaranteed to be

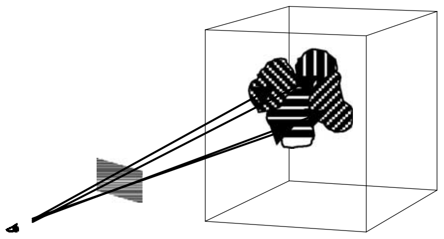


Fig. 1. Illustration of the algorithm. Rays are cast into the dataset; a patch of surface is propagated from each seed cell found.

visible. An efficient propagation algorithm should traverse only the active cells that are visible.

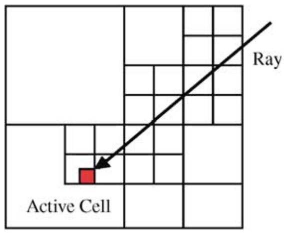
Our algorithm is executed in three stages, as Fig. 1 shows. For each pixel in the screen space, first a ray is cast from the eye through the pixel into the dataset and the intersection is calculated. Next, the first active cell that contains a portion of the isosurface that intersects with the ray (if it exists) is used as the seed and propagated for a certain distance. Third, all the active cells that have been visited in this pass are examined, case numbers are generated and the parts of the isosurface in these cells are triangulated using Marching Cubes method.

2.1. Ray casting

Our method uses ray casting to identify active cells as seeds for propagation. The ray-casting step finds the first active cell in which the isosurface triangulation intersects with the ray. This cell is guaranteed to be visible. If it has not been visited, then this active cell will be used as the seed for the propagation step. Note that this cell is not necessarily the first active cell a ray intersects, as Fig. 2 shows. The first active cell a ray intersects may have an isosurface triangulation that does not intersect with the ray, which means the triangulation will not render the corresponding pixel. By finding the first active cell that actually renders the corresponding pixel, we guarantee that for each ray cast, the corresponding pixel has the correct color. After a ray has been cast for each pixel, the final image will be correct. This proves that our algorithm is conservative.

2.1.1. Intersection calculation

In order to make the ray-casting step efficient, we preprocess the dataset to build a branch-on-need octree (BONO) proposed by Wilhelms and Van Gelder [2] when it is first read into the memory. In the ray-tracing method [7], a 3-level hierarchy was used. This is a trade-off between time and space requirements. The octree has an $O(\log D)$ level hierarchy, where D is the size of the longest side of the dataset. Thus, it uses more space, but the intersection computation is faster.



For each ray, first the algorithm runs recursively to find the first active cell that it intersects. The ray is first tested against the whole dataset. If it intersects the dataset and the threshold is between the overall minimum and maximum of the dataset, then the sub-regions in the dataset that intersect the ray are examined in a front-to-back order. The algorithm performs intersection tests and value comparisons recursively until it finds an active cell that the ray intersects, as shown above. If the algorithm exits without finding a cell, then the ray does not intersect with any active cell in the dataset.

Once the active cell is found, we proceed to test whether the isosurface triangulation inside it actually intersects with the ray. If not, the next active cell the ray intersects is found and tested. This is done till an active cell whose isosurface triangulation intersects with the ray is found or the ray exits the dataset, which indicates that the isosurface does not cover the corresponding pixel.

To decide whether the isosurface triangulation in an active cell intersects the ray, each triangle of the isosurface within the cell is tested for ray intersection. The triangulation contains from 1 to 5 triangles and if any triangle intersects the ray, we render all of them. To perform ray-triangle intersection tests efficiently, we use the method proposed by Möller and Trumbore [11]. The test algorithm is fast and requires minimum storage.

We use a hash table to record those cells that have already been visited. If the active cell found by the recursive algorithm has not been visited, then we will mark it and use this cell as the seed for the propagation

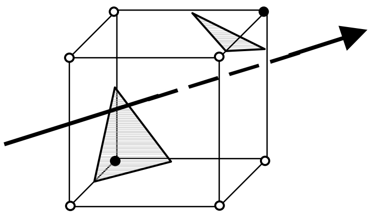


Fig. 2. An example where the first active cell of a ray intersects does not have an isosurface triangulation that intersects with the ray.

step. If the active cell has been visited, then no further action will be taken.

To accelerate the ray-casting step, we use integer coordinates for the data points. The eye and screen are mapped back to the object coordinate system using the current model view matrix. The coordinates of the data points are fixed and implicit: each data point is on a grid and has integer coordinates. The intersection is significantly faster than using the world coordinate system because every intersection test is with a cube with edges parallel to coordinate axes.

There are two design choices to do intersection calculation. The first is exact intersection, which involves isosurface triangulation and vector calculation. The potential disadvantage of this approach is that multiple active cells may be visited and tested before the seed is found. Another method is to approximate an intersection, where we accept the first active cell that intersects the ray as the seed and later check the ray to guarantee correctness. In Section 3.1, we will report our experimental results to compare the two approaches. Based on the results, we decided to use exact intersection.

2.1.2. *Ray-casting order*

The granularity of ray casting determines the speed and the precision of isosurface extraction. Fine-grained ray casting takes time, but it yields precise isosurface representation. Our design lets the user control the density of the rays. When the user changes the threshold or the viewpoint, all the calculations for the previous setup are immediately stopped and new ones begin. If the user does not interrupt, a ray will be cast for each pixel and the correct isosurface will be generated.

There are several ways to cast rays for the algorithm.

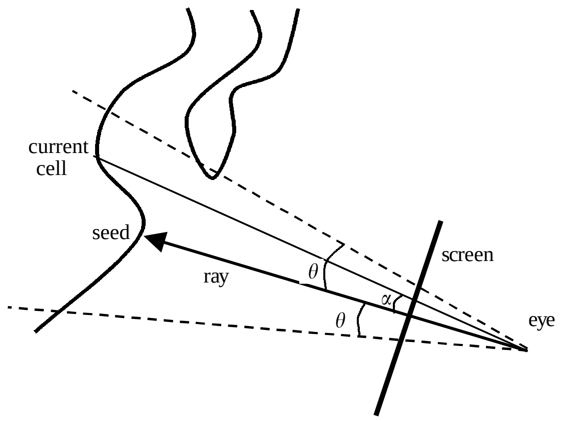
- Fixed refining grids: A straightforward way is to cast sparse and evenly distributed rays in the screen space, then cast rays on progressively finer grids to increase the ray density gradually till a ray has been cast for each pixel or the user interrupts the extraction. This method uses a predetermined hierarchy of grids to determine the ray-casting order. The rays are organized in a quad-tree fashion. The first level consists of one ray that moves from the center of the screen. The second level has 4 rays that are each from the center of one of the 4 quadrants. In general, each level of rays forms a uniform grid that doubles the resolution and contains 4 times as many rays as the last level. At the deepest level, rays simply fill in the pixels that have not been accounted for.
- Screen read-back: The second way to cast rays is to dynamically decide the ray-casting order based on the pixels rendered so far. After casting all the rays of a particular level, the algorithm will read back all the pixels of the screen. This method first casts rays whose corresponding pixels have not been rendered and then cast rays whose corresponding pixels have been rendered.
- Dynamic grid: Another method is to dynamically decide the grid resolution based on the first piece of isosurface generated. When the first piece of new isosurface is generated, the algorithm will test nearby rays to decide the rough size of the surface and dynamically generate a grid in the hope of covering the screen space quickly without leaving any isosurface gaps. The rest of the rays are then cast in the fixed refining grids order.

It is not obvious which of these methods work well. To evaluate the ray-casting order, we have implemented all three ray-casting schemes and compared them in Section 3.2. The dynamic grid performs the best in our experiments.

2.2. *Propagation*

Our algorithm uses a queue for propagation. Initially, the active cell found in the ray-casting step is the only one in the queue. For each cell in the queue, the algorithm dequeues it, sends it to the triangulation step, and checks all its active neighbors. If the active neighbor cells have not been visited and satisfy certain propagation criteria, then they will be added to the queue.

An important issue is propagation distance, which determines how far the propagation for each seed cell should go. The further the propagation proceeds, the fewer inactive cells the algorithm has to examine. On the other hand, although the seed is visible, the cells that it propagates to are not necessarily visible. More propagation may increase the chance of traversing occluded cells. Also, expanding out of the screen space is not desirable.



We have considered two design choices. As shown by the figure on the left, the first way is to calculate the angle α between the current ray and the vector from the eye to the cell being propagated. We set a cut-off angle θ

and use it as the propagation distance. At each propagation step, the algorithm calculates α for the current cell and stops adding it to the queue if $\alpha > \theta$. This method is fairly conservative in terms of generating visible triangles, but it requires vector calculations for each cell visited, which introduces overhead to each propagation step.

Another method is to simply use a maximum count m as the propagation distance. In this case, m is the maximum number of cells the propagation from a seed cell can visit. It is often possible that each pass of propagation will visit fewer than m cells. The ray can hit a gap in the current isosurface and the propagation will stop once the gap is filled and all the neighboring cells have already been visited. Or the ray can hit an isolated isosurface component that contains less than m cells. The advantage of this approach is its simplicity.

We have evaluated these two approaches in Section 3.3. We decide to use a maximum count because of the significant improvement in performance. We also investigated how to choose the value of m in Section 3.6.

2.3. Adaptive-resolution isosurface

For a large dataset, it is possible that a faraway cell is of sub-pixel size when projected on the screen. If it is an active cell, then more than one triangle in the isosurface will be rendered onto the same pixel. That is a waste of computing resources and does not increase the quality of image. Our algorithm detects such cases and reduces the data resolution to $2 \times 2 \times 2$ (treating a meta-cell that consists of 8 cells as a single cell and ignoring all the inside values) or even lower. This is feasible because our ray-dataset intersection walks down an octree hierarchy. We can stop at any resolution if the (meta-) cell projects to less that one pixel on the screen. Given the eye position, screen position, and screen resolution, we can compute an array D, such that for a meta-cell of size $2^i \times 2^i \times 2^i$, if its distance from the screen plane is larger than $D[i]$, then it should be treated as one single cell. When the propagation crosses the resolution boundary defined by D, our algorithm stops at the boundary. At the resolution boundary, there will be cracks in the actual representation of the isosurface, i.e. the triangles from different resolutions may not connect to each other, but the cracks will not be visible because they are of sub-pixel size. Every pixel that the isosurface covers will be rendered by the active cell that is found in the intersection phase using the ray that goes through the center of the triangle. This is similar to [8], where the set of triangles that are rendered is only a subset of all the visible triangles, and where a single point is used to represent a faraway meta-cell. The view-dependent methods generate results that the user perceives as identical with the complete representation from his current viewpoint.

Since in many cases the whole dataset is positioned in the same LOD region, a single test can be done to reduce the number of dynamic LOD decisions made.

3. Performance evaluation

We have implemented the algorithm on a PC running Windows 2000, and conducted experiments to evaluate several design choices described in the previous section. The PC hardware includes a 933 Mhz Pentium III CPU, an NVIDIA GeForce 3 graphics card, and 512 MB of main memory.

We applied our algorithm to the head section of the Visible Woman CT data, using a $512 \times 512 \times 209$ dataset, which is at its original data resolution. The visualization is done in full screen mode with a screen resolution of 1600×1200. In the absence of user interrupt, 1,920,000 rays will be cast, one from each pixel.

To quantitatively measure how close an intermediate representation of the isosurface is to the correct and final representation, for each pixel that has been rendered in the final representation, we check whether it has the same color in the intermediate image, and if so name it as a final pixel. The percentage of the final pixels among all the rendered pixels C indicates the correctness of the intermediate image.

In the following comparisons, we record points where $C = 90, 99, 99.9$ and the end of calculation, when every ray has been cast. Note that 100% correctness can happen before the end of the calculation. For each point, the elapsed time, number of triangles generated, and number of rays cast are measured as a way to compare different design choices.

The basic implementation is the one reported in our conference version of the paper [10]. Our original implementation performs exact ray-isosurface intersection for each ray, uses a cut-off angle to control the propagation, and casts rays in fixed refining grids.

3.1. Exact intersection vs. approximate intersection

In Section 2.1.1, we discussed two approaches to intersection calculation for ray casting: exact intersection and approximate intersection. Table 1 shows the performance of both approaches. Both tests use a cut-off angle θ of 2°.

Although approximate intersection appears to require less computation, some rays have to be checked twice. As a result, the total running time is actually longer than the exact intersection approach. The difference in performance is fairly small with the exception of total running time.

Table 1
Comparing two intersection choices for ray casting

C	Time (s)	Δs (K)	Rays
Exact			
90	1.37	654	833
99	1.72	784	4456
99.9	1.78	786	8711
100	13.8	809	1.92 M
Approximate			
90	1.44	683	833
99	1.73	774	4508
99.9	1.87	776	4855
100	18.7	798	1.92 M

Table 2
Comparing two propagation distance methods: maximum count and cut-off angle

C	Time (s)	Δs (K)	Rays
Maximum count			
90	0.80	380	209
99	0.98	463	879
99.9	1.38	591	18,013
100	13.2	640	1.9 M
Cut-off-angle			
90	1.37	654	833
99	1.72	784	4456
99.9	1.78	786	8711
100	13.8	809	1.92 M

3.2. *Cut-off angle vs. maximum count*

We have considered two design choices of propagation distance to control the extent of the propagation for each seed cell: cut-off angle and maximum count. Table 2 shows the results of the two approaches with best parameters of m and θ. The left column of Table 2 gives the performance of using a maximum count of 8000. The right column is the performance of the cut-off angle approach with $\theta = 2^\circ$.

It is clear that the maximum count approach is superior to the cut-off angle approach. The running time is substantially less than the cut-off angle approach. This is because the cut-off angle approach requires more complex calculations than the simple maximum count and may generate large numbers of invisible triangles when propagating to a piece of occluded isosurface that runs almost parallel to the ray.

3.3. *Three ray casting schemes*

Three ray casting schemes are considered: fixed refining grids, dynamic approach based on screen read-back, and dynamic grid based on the first piece of isosurface. Table 3 shows the results. In all three tests, a maximum count of 8000 is used.

The results suggest that the three ray-casting schemes give similar performance. The dynamic grid scheme has the best running time to reach $C = 99.9\%$. Since the improvement is not significant, we still use the Fixed Refining Grids for its simplicity. We intend to experiment and improve the performance of Dynamic Grid in the future.

3.4. *Caching interpolants and triangles*

Isosurface triangulation is expensive because interpolation involves floating point divisions. An internal edge is shared by 4 cells so that it will be visited 4 times. We use a hash table to record the interpolants.

Each vertex in the isosurface is shared by 6 triangles on average. Instead of copying the coordinates and normally repeating it we use indexed face sets as the representation for a patch of isosurface generated from one pass of propagation. The interpolant cache is reset at the beginning of each pass of propagation. When an item in the hash table is hit, only its index in the vertex array is returned. In our experiments, indexed face sets save 60–70% storage space and render faster.

All the triangles generated are cached to accelerate ray-triangle intersections. All the face sets are stored in a scratch space so that there is no overhead for triangle caching.

Table 3
Experiment results of three ray-casting orders

C	Time (s)	Δs (K)	Rays
Fixed refining grids			
90	0.80	380	209
99	0.98	463	879
99.9	1.38	591	18,013
100	13.2	641	1.92 M
Screen read-back			
90	0.81	380	209
99	0.98	463	879
99.9	1.34	575	18,013
100	13.5	641	1.92 M
Dynamic grid			
90	0.73	340	118
99	1.03	486	1016
99.9	1.23	552	8636
100	13.3	650	1.92 M

The performance improvements by using interpolant and triangle caching are shown in Table 4.

3.5. Screen-bounding box

When the isosurface covers only a small part of the screen, many ray-isosurface intersection tests will return negative. In some of the tests, the ray does not intersect the dataset at all. CPU cycles are wasted without contribution to the results. We partially solved this problem by projecting the dataset onto the screen and finding the screen-bounding box for the dataset. Rays outside the bounding box will not be cast. In our experiments, this optimization shortens the running time for remote viewpoints. When the bounding box covers the full screen, this optimization introduces negligible overhead by doing one more comparison per ray. We scale down the size of dataset on the screen by different percentages and show the respective reductions in running time by using screen bounding box in Table 5. Since in the normal viewpoint we use the dataset which covers almost the whole screen, there is only minimal improvement.

3.6. Impact of m

The choice of m can affect the performance of the algorithm. Table 6 below shows how the position of point $C = 99.9\%$ and total number of triangles rendered the change with m.

Generally speaking, too small an m causes more rays to be cast to generate the approximate result, while too big an m propagates to more invisible triangles. Both these affect the performance of the program. The best choice of m also varies with the threshold. Certain profiling of dataset should be done to decide m offline.

Table 4
The left and right columns show the performance of our program with interpolant and triangle caching turned on and off, respectively

C	Time (s)	Δs (K)	Rays
Caching on			
90	0.64	380	209
99	0.78	463	879
99.9	1.13	591	18,013
100	11.7	641	1.92 M
Caching off			
90	0.80	380	209
99	0.98	463	879
99.9	1.38	591	18,013
100	12.9	641	1.92 M

Table 5
The effect of using screen-bounding box

Size (% of normal)	100	50	25	10
Reduction in running time (%)	3	24	39	49

4. Final results

In this section we present test results from our optimized implementaion. Graph 1 shows, in an experiment of extracting the visible woman's skin, how the correctness, the number of active cells visited, the number of triangles generated, and the percentage of rays cast change as the computation proceeds. The three vertical lines show the positions of points where $C = 90$, 99, 99.9. The corresponding statistics are shown in Table 7. The corresponding screen images are shown in the color plate 1.

Graph 2 and Table 8 show the result from another experiment that extracts the bone structure from visible woman's head. The similarity between the graphs shows that our algorithm behaves consistently. Both cases show that the proposed algorithm works progressively and efficiently. After casting a few rays, our algorithm generates most of the isosurface.

In the skin extraction case, over 90% of the isosurface is extracted in 0.52 s with only 79 rays cast, whereas 99.9% of the isosurface is extracted in about 1.07 s with about 0.7% rays cast. To obtain 100% pixels, it took 11.7 s. The bone extraction case has similar curves but it took 0.99 s to obtain 90% of the isosurface and 2.65 s to extract 99.9% of the isosurface. The total extraction

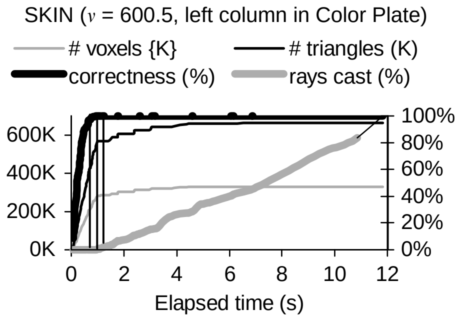
Table 6
The effect of various maximum count values

m (K)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$C = 99.9$ (s)	3.73	1.52	1.31	1.33	1.20	1.07	1.10	1.13	1.07	1.17	1.18	1.15	1.18	1.12	1.26
$\#T$ (K)	506	529	558	562	609	620	639	641	661	669	636	659	685	738	768

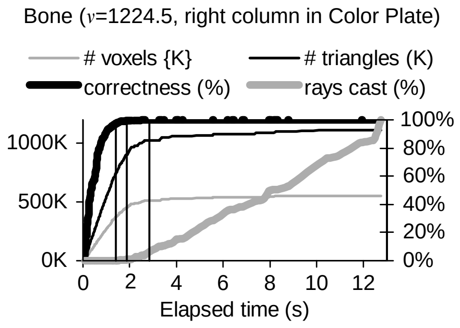
Table 7
Different points in graph 3, $m = 9000$

C	Time (s)	Δs (K)	Rays
90	0.52	306	73
99	0.79	461	947
99.9	1.07	569	13,361
100	11.7	661	1.92 M

took 12.7 s. This is because the isosurface of the bone has about 70% more triangles to render than the skin.



Graph 1: The front full view of the skin.



Graph 2:The front fullview of the bone.

The last 0.1% of pixels took much longer time to extract in both cases, but they make very little difference on the screen.

5. Comparisons with other algorithms

It is difficult to compare our approach quantitatively with other approaches, without implementing them all on the same hardware platform. However, we can make some qualitative comparisons. Here, we focus on view-dependent work.

Our approach allows viewers, to see the shapes of the isosurface after casting only a few rays, whereas in the naïve implementation of ray tracing the visual quality is

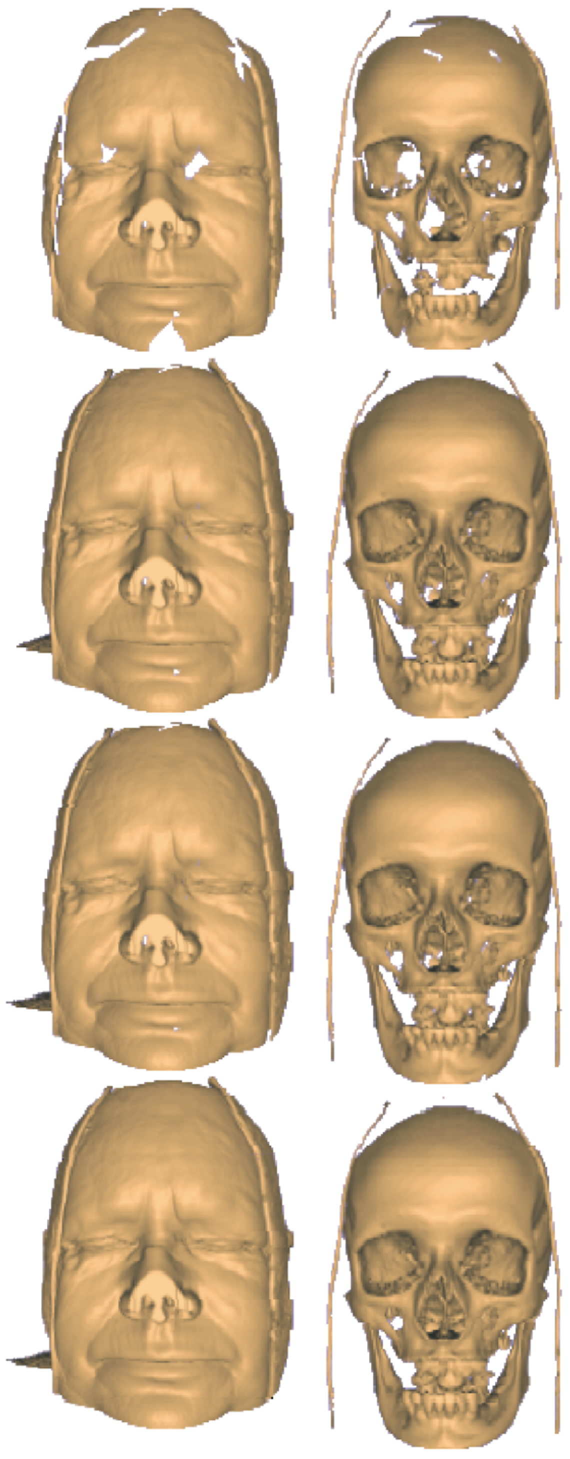


Plate 1. The left column shows four progressively refined images of the skin surface generated at points $C = 90\%$, 99%, 99.9% and the end of calculation in graph 1. The right column shows four images of the bone surface generated at points $C = 90\%$, 99%, 99.9% and the end of calculation in graph 2.

Table 8
Different points in graph 4, $m = 5000$

C	Time (s)	Δs (K)	Rays
90	0.99	523	801
99	1.65	830	4738
99.9	2.65	1070	73,099
100	12.7	1112	1.92 M

linear in both the number of rays cast and the elapsed time. Also, our approach leverages the cost-effective rendering performance of PC graphics cards. Similar to the ray-tracing approach, our algorithm is image–space based and can be parallelized.

To perform a crude comparison with the WISE method of Livnat and Hansen [8], we ran our algorithm using the same size display (512×512 pixels) as they used in the recent experiments [9,12] with the same visible woman dataset. The results of their experiments indicate that running on an SGI Onyx 2 they extract 344,628 triangles in 35.8 s, and then render this surface in 0.6 s. In contrast, our method, running on a Pentium III 933 MHz PC with GeForce 3 graphics card, extracts and renders 463,600 triangles in 4.8 s ($m = 500$) or 614,937 triangles in 2.4 s ($m = 9500$). Based on the relative clock rates on the platforms, we expect that our performance would be better on the SGI than that of the WISE algorithm. Very recent work by Livnat and Hansen introduces SAGE [9,12], a view-dependent algorithm that improves on the performance of WISE (and given the relative hardware difference probably also exceeds the performance of our algorithm), with a reported extraction time of 4.4 s and a rendering time of 0.3 s for the same dataset. Since our visibility test is more conservative, our method extracts and renders many more triangles than the WISE and SAGE algorithms. However, our experiments indicate that triangle rendering is not a bottleneck, and inexact visibility allows us to quickly find large portions of the surface. Our algorithm very quickly provides a good approximation of the surface: 99.9% correctness is achieved after only 1.01 s on the 512×512 display ($m = 9500$).

Finally, taking the 99.9% entry from Table 7 (1.07 s), we see that while the number of the pixels increases by more than a factor of 7 (262,144–1,920,000), the time to compute the approximate surface increases only slightly. To extract 99.9% of the isosurface requires casting 13,361 rays, which is $<0.7\%$ of the total pixels on the screen. This means that the progressive aspect of our algorithm is very insensitive to screen resolution and therefore scales well for very high-resolution displays.

6. Conclusions and future work

This paper studies how to improve a newly proposed isosurface extraction algorithm based on ray casting and propagation. We have learned the following from our evaluation of the design choices of the algorithm:

- Exact intersection calculation is slightly better than approximate intersection calculation.
- Using a maximum count instead of a cut-off angle to control propagation can substantially reduce the running time to extract most of the isosurface. Maximum count approach produces a smaller number of triangles than the cut-off angle approach.
- Ray-casting order does matter slightly. The dynamic grid performs the best. There might be other improvements to this method.
- Caching interpolants and triangles can improve the performance.
- Using a bounding box for the dataset can effectively reduce the number of rays required.
- The maximum count value can impact the performance slightly.

By carefully making design decisions, we can improve the algorithm significantly. The resulting algorithm improves the progressive aspect substantially. In extracting the same skin isosurface, our old implementation reaches $C = 90\%$ and 99.9% at 1.26 and 3.03 s respectively. To extract 90% of the isosurface, the new approach improves over the original one by a factor of 2.4. To extract 99.9%, it improves by a factor of 2.8 and requires casting only a few number of rays, $<0.7\%$ of the pixels of the display screen.

Our algorithm is suitable for high-resolution displays. The results reported in this paper were generated using a PC at full screen (1600×1200) resolution. We would like to adapt the algorithms presented here for use with tiled displays, as part of the Princeton Display Wall Project [13]. As an initial step, we ran the program on a large-scale (18-foot) display surface covered by 24 tiled projectors arranged on a 6×4 grid, yielding more than 20 million pixels. A server PC drives each projector, and each PC runs a copy of the isosurface extraction algorithm. When the isosurface is spread over several projectors, we find a corresponding performance improvement because each PC has a partial view of the surface and has fewer triangles to extract and render. However, when the isosurface falls entirely within one projector, the performance drops to that of a single PC. To address this problem, we are now working on a load-balanced parallel version of the algorithm.

Currently, the entire dataset resides in memory, which limits the size of dataset that can be visualized. Since surface propagation has a strong data-locality, we believe that it will be possible to adapt an out-of-core version of our algorithm.

Remote data visualization has become an important area of research because massive amounts of data are generated and distributed over the network. Since our algorithm aims to reduce the number of triangles generated as well as maintain a fast extraction speed, we believe that it is suitable for remote data visualization. Moreover, surface propagation yields triangle patches that should perform well under geometry compression. Finally, we intend to exploit data-locality due to frame-to-frame coherence in interactive data exploration when adapting our algorithm for remote visualization.

Acknowledgements

This work is supported in part by Intel Corporation, the National Science Foundation (under grant CDA-9624099 and EIA-9975011) and Department of Energy (under grant ANI-9906704 and DE-FC02-99ER25387). The National Library of Medicine provided the Visible Woman dataset. We are also grateful for advice from Tom Funkhouser as well as helpful discussions with Charles Hansen and Yarden Livnat.

References

[1] Lorensen WE, Cline HE. Marching cubes: a high-resolution 3d surface construction algorithm. In: Proceedings of the 14th Annual Conference On Computer Graphics, 1987. p. 163–9.

[2] Wilhelms J, Van Gelder A. Octrees for faster isosurface generation. ACM Trans Graphics 1992;11(3):201–27.

[3] Cignoni P, Marino P, Montani C, Puppo E, Scopigno R. Speeding up isosurface extraction using interval trees. IEEE Trans Visualization Comput Graphics 1997; 3(2):158–70.

[4] Livnat Y, Shen H-W, Johnson CR. A near optimal isosurface extraction algorithm using the span space. IEEE Trans Visualization Comput Graphics 1996;2(1): 73–84.

[5] Itoh T, Koyamada K. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. IEEE Trans Visualization Comput Graphics 1995;1(4):319–27.

[6] Bajaj CL, van Kreveld M, van Oostrum R, Pascucci V, Schikore DR. Contour trees and small seed sets for isosurface traversal. In: Proceedings of the 13th Annual ACM Symposium on Computational Geometry. Nice, France: ACM Press, 1997. p. 212–9.

[7] Parker S, Shirley P, Livnat Y, Hansen C, Sloan P-P. Interactive ray tracing for isosurface rendering. In: Proceedings of IEEE 1998 Conference on Visualization, 1998. p. 233–8.

[8] Livnat Y, Hansen C. View dependent isosurface extraction. In: Proceedings of IEEE 1998 Conference On Visualization, 1998. p. 175–80.

[9] Livnat Y, Hansen C. On view dependent isosurface extraction for large scale data, in preparation.

[10] Liu Z, Finkelstein A, Li K. Progressive view-dependent isosurface propagation. In: Proceedings of the Joint Eurographics—IEEE TCVG Symposium on Visualization, 2001. p. 223–32.

[11] Möller T, Trumbore B. Fast, minimum storage ray-triangle intersection. J Graphics Tools 1997;2(1): 21–8.

[12] Livnat Y.Noise, Wise and sage: algorithms for rapid isosurface extraction. Ph.D. Dissertation, Univeristy of Utah, December 1999.

[13] Li K, Chen H, Chen Y, Clark DW, Cook P, Damianakis S, Essl G, Finkelstein A, Funkhouser T, Klein A, Liu Z, Praun E, Samanta R, Shedd B, Singh JP, Tzanetakis G, Zheng J. Early experiences and challenges in building and using a scalable display wall system. IEEE Comput Graphics Applications 2000;20(4):671–80.