# Progressive View-Dependent Isosurface Propagation

Zhiyan Liu, Adam Finkelstein, and Kai Li

Department of Computer Science, Princeton University

**Abstract** This paper proposes 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. We demonstrate that this progressive algorithm generates an approximate result quickly and refines it to the final correct image over time. In addition, the algorithm scales with the resolution of the display and supports adaptive-resolution visualization.

## 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 existing algorithms all have certain drawbacks. The *marching cubes* [11] 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 is 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* [15] 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* [8]. In preprocessing, both of 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 [1] 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 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.

Recently, two isosurface visualization algorithms are proposed to generate only the visible portions of the surface. Parker *et al*. proposed a *ray-casting* algorithm for isosurface extraction [13] 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 therefore is not well suited for high-resolution displays.

Livnat and Hansen described WISE, a *view dependent* isosurface extraction algorithm that uses hierarchical tiles and shear-warp factorization for visibility testing, and then renders the polygons utilizing the graphics hardware [9]. They also used a 512x512 display. Traversing the dataset in a front-to-back order, (meta) cells are projected to 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 [10], a view dependent algorithm that improves on the performance of WISE.

In this paper, we propose a new hybrid algorithm that shares several of the features of existing acceleration methods. 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*. [13]) 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 distance and viewing criteria to decide where to stop propagation, and thus generates only a small piece of the isosurface connected to each seed cell. These pieces are patched together to form a view-dependent region of the isosurface, which includes all the triangles that are visible as well as a small number of occluded triangles that are near the visible surface.

In addition to largely avoiding the rendering of occluded portions of the isosurface, our method supports two acceleration schemes suitable for interactive visualization. First, we show that we can quickly acquire a very good approximation of the visible isosurface from just a few initial seeds, and then progressively refine the surface as subsequent rays discover the remaining visible active cells. Second, for very high-resolution grids, we describe a form of adaptive-resolution rendering that relies on the octree organization of the data in order to extract and render triangles at a resolution chosen based on the scale of screen pixels relative to the voxel data.

## 2. The Algorithm

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 efficient and incremental while supporting adaptive-resolution visualization.

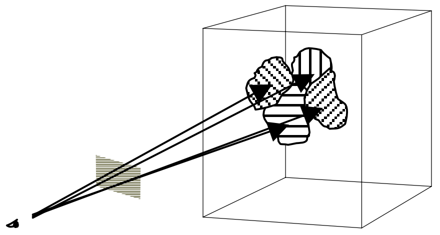


Figure 1: Illustration of the algorithm. Rays are cast into the dataset; a patch of surface is propagated from each seed cell found.

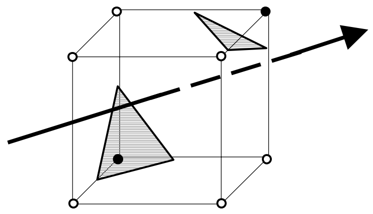


Figure 2: An example where the first active cell a ray intersects doesn't have an isosurface triangulation that intersects with the ray.

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's called an active cell, and the isosurface of threshold *v* will intersect this cell.

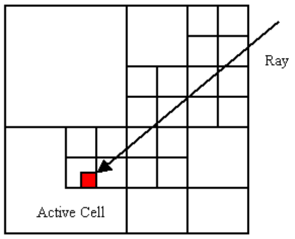
A key observation the propagation approach made was that if the vertices on a face of an active cell do not have the same sign, then the neighboring cell that shares the same face is also active. Therefore, the isosurface can be extended into the neighboring cell. This means that once an active cell 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 visible. An efficient propagation algorithm should traverse only the active cells that are visible.

Our algorithm executes in three stages, as Figure 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, 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 Figure 2 shows. The first active cell a ray intersects may have an isosurface triangulation that doesn't intersect with the ray, which means the triangulation won't render to the corresponding pixel. By finding the first active cell that actually renders to 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. That proves our algorithm is conservative.

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 [15] when it is first read into the memory. In the ray-tracing method [13], a 3-level hierarchy was used. This is a trade-off between time and space requirements. The Octree has an *O(logD)* 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, 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 to the left. 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 doesn't cover the corresponding pixel.

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

A typical way to visualize the data is to begin by casting sparse and evenly distributed rays in the screen space, then add more rays to increase the ray density gradually till a ray has been cast for each pixel or user interrupts the extraction.

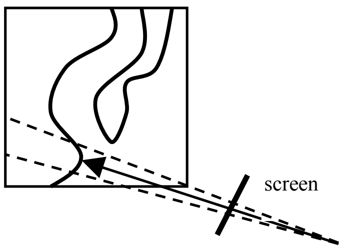
For a given screen resolution, the order of rays to be cast can be predetermined: the first ray goes from the center of the screen, the next 4 rays are each from the center of one of the 4 quads, and so on. The granularity of ray casting determines the speed and the precision of isosurface extraction. Fine-grained ray casting takes time, but it yields precise isosurface representation. Our design is to let user control the density of ray. 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 doesn't interrupt, a ray will be cast for each pixel and the correct isosurface will be generated.

## 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, they will be added to the queue.

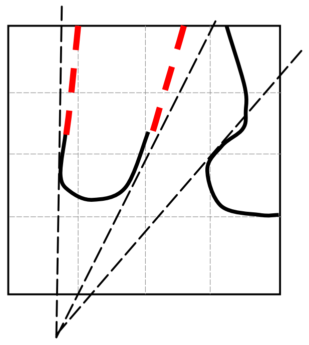
The propagation criteria need to be chosen carefully. The further the propagation proceeds, the fewer inactive cells the algorithm has to examine. On the other hand, even though the seed is visible, the cells that it propagates to are not necessarily visible. More propagation may increase the chance of traversing occluded cells. Also, expanding out of the screen space or to the back-faced side of isosurface is not desirable.

Our algorithm sets a cut-off angle for a ray and names it the propagation distance. At each propagation step, the algorithm calculates the angle between the ray we cast and the vector from the eye to the current cell and stops adding it to the queue when the cut-off angle is reached.



The figure to the left shows using a cut-off angle as the propagation distance. The propagation will not expand out of the region defined by the dashed lines. Suppose the first *m* rays have a combined propagation area that covers the whole screen, then after all the *m* rays are cast, the only possible reason for inaccuracy in the isosurface we computed is that there is an isolated part of the isosurface in front of the isosurface we expanded and it didn't intersect with any ray we've cast. To treat this situation we just cast more rays at different locations. If the user doesn't interrupt, at last one ray will be cast for each pixel and result is guaranteed to be correct. We may generate more triangles than that are actually visible, but it's interactive because the user gets approximate results that refine with time. Our results show that for several datasets, only a relatively small number of rays are needed.

In order to avoid generating occluded triangles, we calculate the angle between the current ray and the vector from the eye to the cell being propagated to detect whether the isosurface folds back. We name this angle the distance angle. If the isosurface folds back in a cell, that cell will not be added to the queue and the propagation from the cell will stop. The detection works as follows. If a cell C1 is examined and its neighbor C2 is added to the queue, then we say C1 is C2's predecessor in propagation, and C2 is C1's successor. Normally, the distance angle between the ray and the vector increases as the propagation proceeds. When the distance angle for a cell decreases comparing with its predecessor cell, then the isosurface is curving back and the cell should be discarded. However, to deal with bumpy surfaces, our algorithm has a small tolerance value. Only when the decrement exceeds the tolerance value do we stop.



Propagation is an efficient way of extracting triangles; we are willing to pay the small price of rendering a few more triangles to keep the propagation going. The decrement is calculated from the largest distance angle among all the (indirect) predecessors of a cell, so even if each time the decrement is very small, they can accumulate and stop the propagation. This solves the problem where the isosurface curves back and propagates in a direction that's almost parallel to the ray, as shown on the left. The dashed part of isosurface won't be propagated to, whereas a small groove is tolerated and propagation goes on.

### 2.3 Adaptive-resolution isosurface

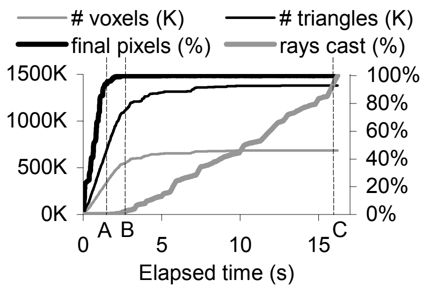
For a large dataset, it is possible that a far-away cell is of sub-pixel size when projected to the screen. If it is an active cell, then more than one triangle in the isosurface will render onto the same pixel. That is a waste of computing resources and does not increase the quality of image. Our algorithm detects such cases and reduces the data resolution to 2×2×2 (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 eye is larger than *D[i]*, then it should be treat 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 won't be visible because they are of sub-pixel size. Every pixel that the isosurface covers will be rendered to by the active cell that's found in the intersection phase using the ray that goes through the center of the triangle. This is similar to [9], 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 user perceives as identical with the complete representation from his current viewpoint.

## 3. Results

We have implemented the algorithm above on a PC that runs Windows 2000, and conducted experiments with the implementation.  The PC hardware includes a 933Mhz Pentium III CPU, an NVIDIA GeForce 256 graphics card, and 512 MB of main memory.

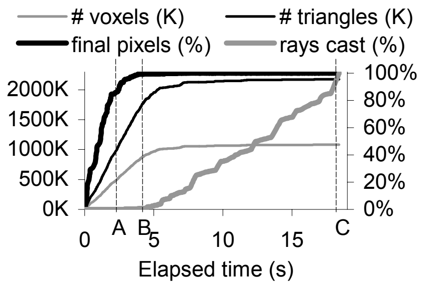
We applied our algorithm to the head section of the Visible Woman CT data, using a 512×512×209 dataset, which is at its original data resolution. A cut-off angle of 1.81 degrees was used for all the experiments. The visualization is done in full screen mode with a screen resolution of 1600x1200. In the absence of user interrupt, 1,920,000 rays will be cast, one from each pixel.

SKIN ($v$ = 600.5, left column in Color Plate)



Graph 1: The front full view of the skin.

Bone ($v$=1224.5, right column in Color Plate)



Graph 2: The side full view of the bone.

| Point | Time (s) | Δs (K) | Pixels (%) | Rays (K) |
|---|---|---|---|---|
| A | 1.1 | 514 | 85.3 | 0.2 |
| B | 2.4 | 1,068 | 99.5 | 15.6 |
| C | 16.2 | 1,380 | 100.0 | 1,920 |

Table 1: Different points in Graph 1.

| Point | Time (s) | Δs (K) | Pixels (%) | Rays (K) |
|---|---|---|---|---|
| A | 2.2 | 953 | 85.7 | 0.6 |
| B | 4.3 | 1,796 | 99.5 | 15.1 |
| C | 18.4 | 2,175 | 100 | 1,920 |

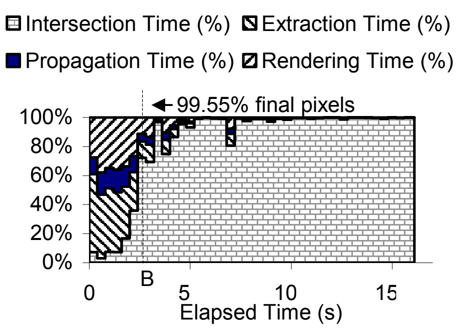
Table 2: Different points in Graph 2.

To quantitatively measure how close the an intermediate representation of the isosurface is to the correct and final representation, for each pixel that has been rendered to in the final representation, we check whether it has the same color in the intermediate image, and if so name it a final pixel. The percentage of the final pixels among all the rendered pixels indicates the correctness of the intermediate image. Graph 1 shows in an experiment of extracting the Visible Woman's skin, how the percentage of final pixels, the number of active cells visited, the number of triangles generated, and the percentage of rays cast change as the computation proceeds. The statistics for points A, B, and C are shown in Table 1. The corresponding screen images are shown in the color plates.

Graph 2 and Table 2 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 85% of the isosurface is extracted in 1.1 seconds with only 240 rays cast (point A), whereas 99.5% of the isosurface is extracted in about 2.4 seconds with about 0.8% rays cast (point B). To obtain 100% pixels, it took 16.2 seconds. The bone extraction case has similar curves but it took 2.2 seconds to obtain 85% of the isosurface and 4.3 seconds to extract 99.5% of the isosurface. The total extraction took 18.4 seconds. This is because the isosurface of the bone has about 60% more triangles to render than the skin.

The last 0.5% of pixels took much longer time to extract in both cases, but they make very little difference on the screen.



The graph on the left shows the breakdown of the computation time. The definitions of the stages are as follows:

*Intersection time* is the time spent on ray-dataset intersection and finding visible seeds.

*Extraction time* is the time spent on using marching cubes algorithm to compute the isosurface triangulation inside active cells.

*Propagation time* is the time spent on finding the active neighbors of a cell in the propagation queue and testing whether they satisfy the propagation criteria.

*Rendering time* is the time spent on rendering all the triangles generated.

Note that among these four time components, only intersection time is proportional to the number of rays cast. The other three are proportional to the number of triangles generated. When the full computation ends, the intersection calculation is the most time-consuming stage. However, most of the isosurface has been generated by the time the intersection stage becomes significant. The dashed line shows point B in Graph 1. Before point B, the time spent in intersection is negligible. At point B, the representation is 99.5% correct. After point B, the intersection calculation becomes dominant. In this experiment, because of the high resolution of the display (1600×1200), all the triangles are generated at the finest data resolution. This implies that our algorithm is suitable for large-scale displays. For a fixed dataset, when the resolution of the display increases, only the intersection time will increase, which has very little influence on the position of point B on the time axis.

## 4. Comparisons with previous 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 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 [9], we ran our algorithm using the same size display (512×512 pixels) as they used in recent experiments [10] with the same Visible Woman dataset. The results of their experiments indicate that running on a SGI Onyx 2 they extract 344,628 triangles in 35.8 seconds, and then render this surface in 0.6 seconds. In contrast, our method, running on a Pentium III 933MHz PC with GeForce graphics card, extracts and renders 1,289,904 triangles in 4.3 seconds. 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 [10], a view dependent algorithm that improves on the performance of WISE (and given the relative hardware difference probably exceeds our performance as well), with a reported extraction time of 4.4 seconds and rendering time of 0.3 seconds for the same dataset. Because 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: 85% and 99.5% correctness were achieved at 0.9s and 2.2s respectively on the 512×512 display. Finally, comparing these numbers with those in Table 1 (1.1s, 2.4s), we show that while the number of the pixels increases by more than a factor of 7 (262,144 to 1,920,000), the points A and B were only delayed 20% and 10% respectively. This suggests that the progressive aspect of our algorithm scales well for very high-resolution displays.

## 5. Conclusions and Future Work

This paper describes a new isosurface extraction algorithm based on ray casting and propagation. We have shown that the new approach is progressive and efficient.

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 [7]. 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.

Our algorithm is suitable for large datasets. Currently the entire dataset resides in memory, which limits the size of dataset we can visualize. Because surface propagation has 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 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] C. L. Bajaj, M. van Kreveld, R. van Oostrum, V. Pascucci, and D. R. Schikore. Contour Trees and Small Seed Sets for Isosurface Traversal. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pages 212-219, ACM Press, Nice, France, 1997.

[2] Yi-Jen Chiang, Cláudio T. Silva and William J. Schroeder. Interactive Out-Of-Core Isosurface Extraction. In *Proceedings of IEEE 1998 Conference on Visualization*, 1998, Pages 167 – 174.

[3] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2): 158-170, 1997.

[4] Satyan Coorg and Seth Teller; Temporally Coherent Conservative Visibility (extended abstract). In *Proceedings Of The Twelfth Annual Symposium On Computational Geometry*, 1996, Pages 78 – 87.

[5] Takayuki Itoh and Koji Koyamada. Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists. *IEEE Transactions On Visualization And Computer Graphics*, 1(4): 319 -327 December 1995.

[6] Takayuki Itoh, Yasushi Yamaguchi and Koji Koyamada. Volume thinning for automatic isosurface propagation. In *Proceedings Of IEEE 1996 Conference On Visualization*, 1996, Page 303-310.

[7] Kai Li, Han Chen, Yuqun Chen, Douglas W. Clark, Perry Cook, Stefanos Damianakis, Georg Essl, Adam Finkelstein, Thomas Funkhouser, Allison Klein, Zhiyan Liu, Emil Praun, Rudrajit Samanta, Ben Shedd, Jaswinder Pal Singh, George Tzanetakis and Jiannan Zheng. Early Experiences and Challenges in Building and Using A Scalable Display Wall System. *IEEE Computer Graphics and Applications, vol 20(4), pp 671-680, 2000.*

[8] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space; *IEEE transactions on Visualization and Computer Graphics*, 2(1): 73-84, March 1996.

[9] Yarden Livnat and Charles Hansen. View Dependent Isosurface Extraction. In *Proceedings of IEEE 1998 Conference On Visualization*, 1998, Pages 175 – 180.

[10] Yarden Livnat and Charles Hansen. On View Dependent Isosurface Extraction for Large Scale Data. *Under submission.*

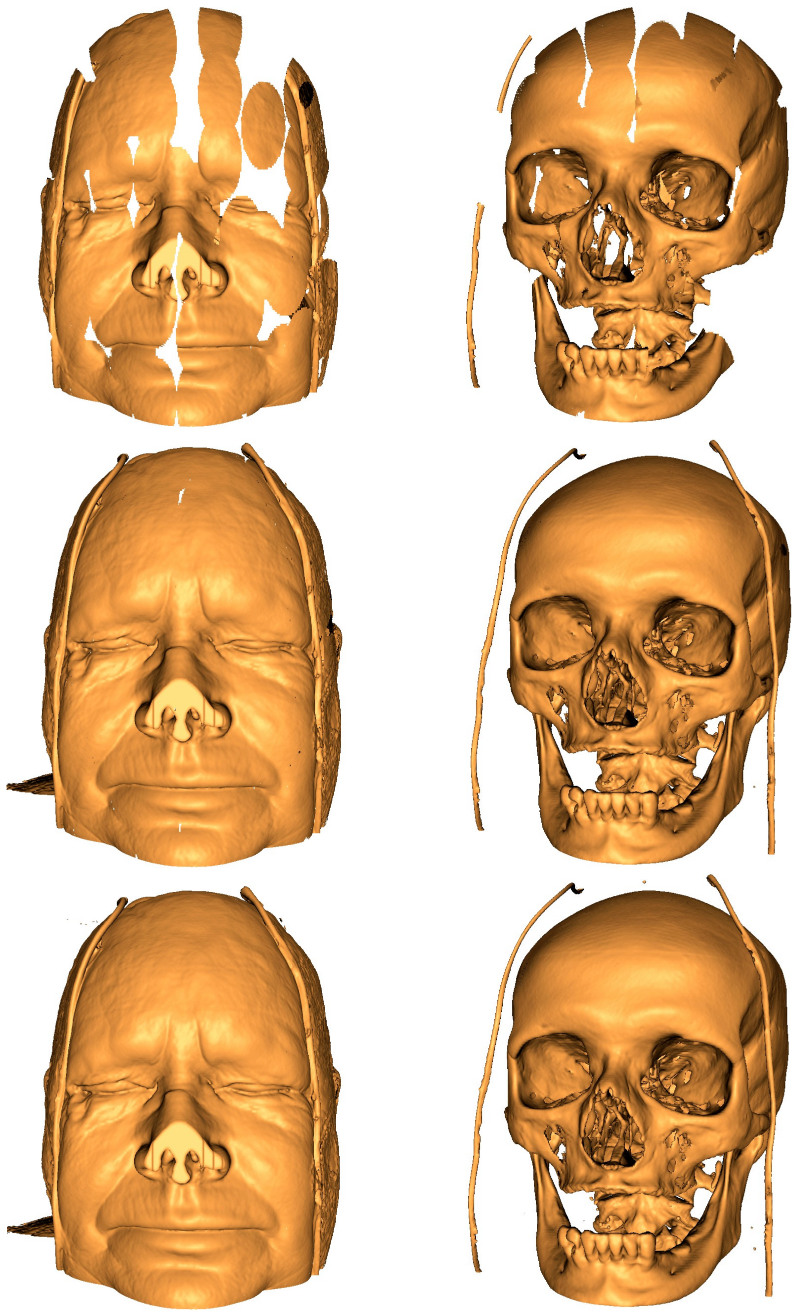
[11] William E. Lorensen and Harvey E. Cline. Marching cubes: A High-Resolution 3D Surface Construction Algorithm. In *Proceedings Of The 14th Annual Conference On Computer Graphics*, 1987, Pages 163 – 169.

[12] Michael Lounsbery, Tony DeRose and Joe Warren; Multiesolution Analysis For Surfaces Of Arbitrary Topological Type. *ACM Transactions on Graphics,* 6(1):34-73, January 1997.

[13] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen and Peter-Pike Sloan. Interactive Ray Tracing For Isosurface Rendering. In *Proceedings Of IEEE 1998 Conference On Visualization*, 1998, Pages 233 – 238.

[14] Han-Wei Shen. Isosurface Extraction In Time-Varying Fields Using A Temporal Hierarchical Index Tree. In *Proceedings Of IEEE 1998 Conference On Visualization*, Pages 159 – 166.

[15] Jane Wilhelms and Allen Van Gelder. Octrees For Faster Isosurface Generation. *ACM Transactions on Graphics.* 11(3): 201-227, July 1992.



**Color plate 1: The left column shows three progressively refined images of the skin surface generated at points A, B and C in Graph 1. The right column shows three images of the bone surface generated at points A, B and C in Graph 2. In the middle row, after less than 1% of the rays have been cast, the resulting images are almost indistinguishable from the final images in the bottom row.**