SCI INSTITUTE
TECHNICAL REPORT

PHASE: Progressive Hardware Assisted IsoSurface Extraction Framework

Yarden Livnat, Xavier Cavin, Charles Hansen

UUSCI-2002-001

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

Enter date here: 2002

Abstract:

Isosurface extraction is an important technique for visualizing three-dimensional scalar fields. During recent years, researchers have created many acceleration methods for isosurface extraction, including the span space representation and view-dependent methods. In this paper, we introduce a progressive view-dependent isosurface extraction method that exhibits a rapid convergence rate to the exact isosurface and is well suited for remote visualization. The proposed method takes advantage of rendering hardware to resolve visibility tests. In contrast to previous view-dependent isosurface extraction methods, our method (PHASE) can quickly augment the current partial extracted isosurface based on a new point of view without the need for a full view-dependent extraction pass.

THE UNIVERSITY OF UTAH

PHASE: **P**rogressive **H**ardware **A**ssisted Iso**S**urface **E**xtraction Framework

Yarden Livnat, Xavier Cavin, Charles Hansen

Abstract— **Isosurface extraction is an important technique for visualizing three-dimensional scalar fields. During recent years, researchers have created many acceleration methods for isosurface extraction, including the span space representation and view-dependent methods. In this paper, we introduce a progressive view-dependent isosurface extraction method that exhibits a rapid convergence rate to the exact isosurface and is well suited for remote visualization. The proposed method takes advantage of rendering hardware to resolve visibility tests. In contrast to previous view-dependent isosurface extraction methods, our method (PHASE) can quickly augment the current partial extracted isosurface based on a new point of view without the need for a full view-dependent extraction pass.**

Keywords— **Isosurface Extraction, View-Dependent, Progressive Extraction, Hardware Accelerated, Remote Visualization, Output Sensitive**

I. Introduction

Scientists and engineers often rely upon knowledge obtained from experiments and simulations that produce large scale discrete samplings of three-dimensional scalar fields. Isosurface extraction is an important technique for visualizing three-dimensional scalar fields. By exposing contours of constant value, isosurfaces provide a mechanism for understanding the structure of the scalar field. These contours isolate surfaces of interest, focusing attention on important features in the data such as material boundaries and shock waves, while suppressing extraneous information. Several disciplines, including medicine [1], [2], computational fluid dynamics (CFD) [3], [4], and molecular dynamics [5], [6], effectively use this method. However, for very large datasets, the enormous data size overwhelms the interactive extraction and rendering times for isosurfaces. Without techniques to address the large-scale data size, the practicality of these methods would be limited.

The availability of inexpensive yet powerful desktop computers and parallel supercomputers in few location leads to the development of remote visualization techniques. The fundamental drive for this paradigm enables the scientist to perform very large simulations on a remote supercomputer while visualizing and investigating the results on the local desktop. Remote visualization of large datasets poses an even greater challenge for isosurface extraction due to the limited bandwidth of the intermediate network.

In respond to this challenge, current research efforts aim to simplify the geometry of the isosurface *after* extracting the isosurface and *before* it renders or transmits over a network [7], [8]. In effect, the aim of such methods are to reduce the complexity of rendering an isosurface to a sublinear complexity with respect to the size of the original isosurface. However, such methods do not address the initial challenge of extracting and constructing the isosurface.

In this paper, we present a new view-dependent extraction framework that progressively and rapidly extracts the visible portion of an isosurface. The proposed framework uses a novel visibility propagation scheme that eliminates the need for software rendering of extracted triangles and reduces the number of required expensive visibility tests. Moreover, the few required visibility tests are performed using the graphics acceleration hardware. PHASE is particularly attractive for interactive investigation of large data volumes where only small sections of the data are visible at any particular moment, such as navigating through a colonoscopy data set. We begin with a review of earlier work on isosurface extraction in Section II. The PHASE framework is discussed in Section III, followed, in Section IV, by a detailed description of the theory underlying the visibility traversal pipeline and the visibility maps on which the PHASE algorithm is based. Hardware acceleration of PHASE is presented in Section V. We present test results in Section VI and conclude with future directions in Section VII.

II. Previous Work

The time complexity of isosurface extraction algorithms depends on the size of the dataset, n, and the size of the isosurface, k. Early extraction methods [9], [10], [11], [12], [13], [14], [15], [16] exhibit worst case time complexity of $O(n)$. With the introduction of the Span Space, the NOISE algorithm [17], [18] achieved a worst case complexity of $O(\sqrt{n}+k)$ while Cignoni *et al.* [19] achieved $O(\log n+k)$ albeit, with a higher memory cost. As we minimize the

Scientific Computing and Imaging Institute, University of Utah, E-mail:yarden@cs.utah.edu

size dependency of the original dataset, the size of the extracted isosurface, k, becomes the dominate factor. Interactively rendering a very large isosurface presents a great challenge even on high-end graphics workstations.

Livnat and Hansen proposed [20], [21] to surpass the $O(k)$ barrier based upon the observation that the isosurfaces extracted from very large datasets often exhibit high depth complexity. There are two reasons for this depth complexity. First, since the datasets are very large, the projection of individual cells tend to be sub-pixel. This leads to a large number of polygons, possibly non-overlapping, that project onto individual pixels. Second, for some datasets, the depth complexity is high since large sections of an isosurface are internal and thus, occluded by other sections of the isosurface, as illustrated in Figure 1.

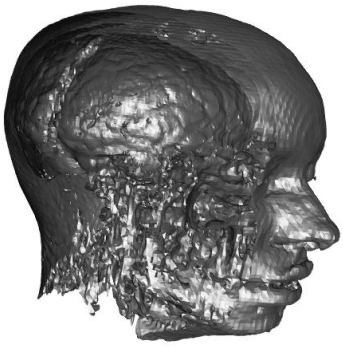


Fig. 1
A SLICE THROUGH AN ISOSURFACE REVEALS THE INTERNAL SECTIONS WHICH CAN NOT CONTRIBUTE TO THE FINAL IMAGE.

These internal sections cannot be seen from any direction unless the external isosurface is peeled away or cut off. Therefore, if one can extract just the visible portions of the isosurface, the number of rendered polygons will be reduced resulting in a faster algorithm and lower bandwidth requirement for remote visualization. Figure 2 depicts a two-dimensional scenario. In a view-dependent method, only the solid lines are extracted whereas in non view-dependent isocontours, both solid and dotted lines, are extracted.

A. View-Dependent Framework

The computer graphics literature contains many view-dependent rendering and traversal algorithms. These algorithms generally rely on a pre-process stage in which the geometry primitives are organized for an efficient access and visibility determination for any given view direction.

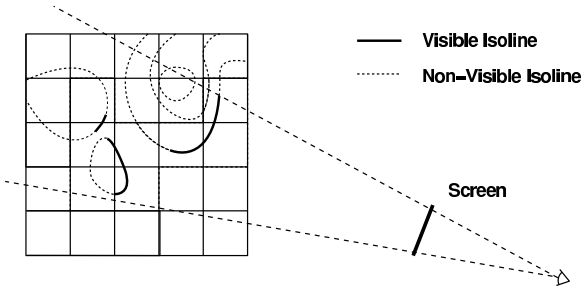


Fig. 2
A TWO-DIMENSIONAL SCENARIO.

However, isosurface extraction poses a special problem as there is no geometry to pre-process prior to extraction, still worse, the extracted geometry changes for every given isovalue. As such, view-dependent isosurface extraction can rely only on the underlying structure of the data volume.

The view-dependent framework presented by Livnat and Hansen [20] is based on a hierarchical traversal of the data and a Marching Cubes triangulation. This framework exploits coherency in the object, value, and image spaces, as well as balancing work between the hardware and the software. The algorithm used a three step approach: first, the data traversal used in Wilhelms' and Van Gelder's algorithm [13] is augmented to follow a front-to-back order in addition to pruning empty sub-trees based on the min-max values stored at the octree nodes. The second step employs coarse software visibility tests for each meta-cell visited during the traversal. The aim of these tests is to determine whether previously extracted sections of the isosurface obscure a meta-cell from view (thus the requirement for a front-to-back traversal). Finally, the triangulation of the visible cells are forwarded to the graphics accelerator for rendering by the hardware, where the final and exact [partial-] visibility of the triangles is resolved.

B. View-Dependent Implementations

In [20], Livnat and Hansen also presented the WISE (Warped IsoSurface Extraction) algorithm, which provided a particular implementation of the view-dependent approach. The algorithm took advantage of a shear-warp factorization of the viewing transformation, as well as the Greene *et al.* [22] occlusion culling technique. The WISE algorithm demonstrated the potential benefits of such an approach. The ratio of triangle intersections per screen cell, and the fill rate of screen tiles hierarchy pose the two most prominent weaknesses for this method.

Recently, Livnat *et al.* introduced a new view-dependent algorithm, termed SAGE [23], [21], that addressed those

weaknesses. In order to alleviate the problem of projecting many small triangles down the hierarchical tile structure, SAGE employs a bottom-up approach. This approach is based on the observation that the contribution of a small triangle is limited to only a small neighborhood in the hierarchy, *i.e.*, few tiles at the lowest level. Thus, the triangles are projected at the highest resolution and the changes are forwarded up the hierarchy. However, the visibility checks are still performed in a top-down fashion in order to quickly determine if a cell is visible. Replacing Greene's occlusion culling technique with scan-conversion of groups of triangles, reduced the number of triangle intersections per screen cell. The SAGE method exhibits an acceleration factor between 5 and 10 over the previous WISE method.

In 2001, Gao and Shen [24] introduced a parallel multipass view-dependent algorithm, based on similar ideas. A master host maintains a virtual framebuffer for visibility queries. It then distribute the cells that require triangulation among the other hosts. Each such host creates, after the triangulation, a local visibility map based on the triangles it created. The collection of these local visibility maps are then combined and integrated into the master virtual framebuffer and the procedure is repeated. The master framebuffer use 2-bit per pixel as oppose to the 1-bit per pixel both WISE and SAGE use. This extra bit allows the algorithm to project meta-cells which may be hidden by other meta-cells in front of it. As opposed to WISE and SAGE which do not alter the state of the framebuffer during the visibility query, Gao and Shen modify the framebuffer and use this extra bit to determine if a meta-cell is occluded only by previously extracted triangles or by meta-cells as well. While the algorithm shows a very good load balancing it has a several key shortfalls. First, all the rendering is done in software which is slow especially due to the need to maintain the complex 2-bit per pixel state. Second, the performance of the algorithm is poor with regard to execution time.

Recently, Liu *et al.* [25] introduced a progressive view-dependent isosurface propagation algorithm. This algorithm casts rays from the view point through the data to find visible non empty cells, and uses these cells as seeds to propagate the polygonal isosurface. This algorithm allows one to quickly render a good approximation, but takes a long time to converge to the final image due to the large number of rays required.

Zhang *et al.* [26] presented a parallel out of core view dependent extraction method. In this approach, the sections of the data are distributed to the various processors. For a given isovalue, each processor uses ray casting to generate an occlusion map. The maps are then merged and redistributed to all the processors. Using this global occlusion map, each processor extract its visible portion of the isosurface. They noted that updating the occlusion map during the traversal is expensive even when using hierarchical occlusion map, and thus opt not to update the occlusion maps and rely on the first occlusion map approximation.

An alternate approach is to generate an *adaptive* reconstruction of the isosurface. These methods extract an approximation of the isosurface based on some user specified criteria. Westermann *et al.* [27] allowed the user to specify a point of interest inside the data around which they extract a refined isosurface. Further away from that center of attention, they extract the isosurface using a coarse representation of the data. The view-dependent *refinement* approach of Gregorski *et al.* [28] used the user view point as well as user specified error tolerance in screen space. They recursively refined the isosurface until these criteria where met. Their method used a novel representation of the underlying data, and was able to achieve a remarkable interactivity on very large isosurfaces. However, this approach is not suitable for the purpose of remote visualization, as the same section of the isosurface can have different representations from different view points.

C. Visibility Framebuffer

A key requirement in most view-dependent extraction methods is to maintain the current visibility state against which the extraction process can perform visibility queries. To this end, every extracted triangle is projected onto a virtual framebuffer. However, rendering to the virtual framebuffer and performing visibility tests against it must be executed quickly and accurately.

Both WISE and SAGE used 1-bit per pixel visibility masks to maintain the state of the screen pixel (*i.e.* covered or not), while Gao and Shen [24] used 2 bits per-pixel (visible, covered by triangle and covered by a meta-cell). There are four major drawbacks to these approaches. First, because the framebuffer maintains only a minimum state, the first two algorithms cannot simultaneously perform visibility checks on two meta-cells whose projections on the screen may overlap. Second, the rendering of the visibility masks and visibility queries are performed in software that is inherently slow. Third, the algorithms do not take advantage of the coherency between consecutive view points and thus must begin the extraction process anew for each new view point. Fourth, visibility tests are performed on each and every meta-cell encountered during the traversal, though many are trivially visible, *e.g.*, cells on the boundary of the dataset.

III. Progressive View-Dependent Isosurface Extraction

In this section we address two key shortcomings of previous view-dependent approaches (*a priori* knowledge and view coherency). As part of this discussion we introduce the notions of strict *vs* loose traversal order and delayed visibility queries. These two are the key to the PHASE framework that we present at the end of this section. In the next section we provide a more detail discussion of the theory behind PHASE while implementation issues are presented in the following sections.

The proposed framework, PHASE, is based on two observations. First, many of the meta-cells may be trivially categorized as either visible or occluded without performing an expensive visibility query. Second, most of the visible/occluded cells from one view point will remain so when viewed from a nearby view point.

A. Full Z-Buffer Visibility Tests

In order to take advantage of view coherency, one needs to replace the one or two bits per pixel visibility framebuffer with a full z-buffer. Using a z-buffer removes the need to perform the visibility checks in a *strict* front-to-back order. One must still perform most of the visibility tests in a hierarchical front-to-back manner in order to achieve efficient pruning during the data traversal. In addition, the full z-buffer allows the extraction process to use visible triangles from a previous view point, as *potential* occluders [29][Note: `add more citations`] during the extraction from a new view point. To this end, at the beginning of each view-dependent extraction cycle the visibility framebuffer is initialized with the projection of the currently visible triangles. Due to the view coherency, only a few previously hidden meta-cells and triangles will become visible. In essence, most of the construction of the visibility framebuffer is achieved in the first projection pass.

There is, however, a cost of using this approach. While new triangles can be added in each cycle, none are removed even if they are hidden from the new view point. More importantly, the projection of the triangles and the visibility queries are much more complex and time consuming than those performed on a one bit per pixel visibility buffer. Rendering requires computing and comparing the z value for each rendered pixel. Visibility checks of meta-cells can not rely on a single bit for large areas in the visibility framebuffer rather the z values of all the pixels that are covered by a meta-cell must be examined. The visibility queries are also more complex and expensive even with sophisticated data structures such as the hierarchical z-buffer [30].

One way to accelerate the full z-value rendering is to replace the virtual visibility framebuffer with the graphics hardware framebuffer. This approach resolves the rendering issue, but on the other hand, introduces a major hurdle because reading back from the framebuffer is relatively slow. This is a problem especially in view-dependent algorithms that must perform large number of visibility checks.

B. Loose versus Strict Traversal Order

PHASE addresses the slow read back problem by deferring the read as long as possible and performing many visibility checks via a single query of the hardware buffer. Referring to Figure 3, we first note that octant B, C and D do not overlap and thus, can be traversed in parallel (once octant A is fully traversed), *i.e.*, their visibility queries can be performed in one read from the hardware framebuffer. Yet, this parallelism is limited as none of the sub-octants inside each of them can be traversed before a visibility query is performed on their parent. In addition, this approach is limited only to non-overlapping regions.

The key to the PHASE approach is that a full z-buffer removes the need for a *strict* front-to-back traversal of the data. A strict front-to-back traversal requires that all the cells in octant A be checked before octant B, C and D can be traversed. However, even though some portions of these octants may be occluded by triangles in octant A, other portions are, nevertheless, trivially visible. As such, a hierarchical traversal can descend these three octants even before a complete traversal of octant A completes. Furthermore, sub-octants $B_{A,B,D,E}$ are also trivially visible as they form part of the outside boundary.

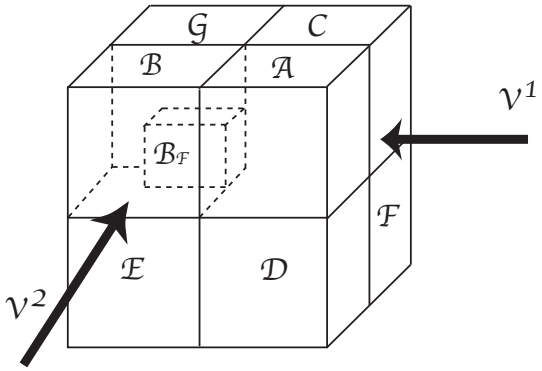


Fig. 3
Octants, sub-octants and various view directions.

In summary, we can conclude that octants A, B, C, D, E, F and G are trivially visible without the need to perform any

visibility queries against the framebuffer. Some of the sub-octants are also trivially visible and can be traversed out of order. However, not all octants can be traversed out of the strict front-to-back order. It is not clear, for example, that octant B_F can be traversed before octant A is fully traversed.

Consider two view points, V^1 and V^2, as shown in Figure 3 and assume that sub-octant B_D is empty (the iso-surface does not go though it). From view point V^1, sub-octant B_F should not be visited before sub-octant A_H. However, from view point V^2, no such constraint exists. Furthermore, the fact that B_D is visible and empty implies that B_F may be visible from many other view points between V^1 and V^2 (as well as many other view points). Note that we concluded (for view point V^2) that sub-octant B_F is visible, and can thus be traversed at anytime, purely on adjacency information, current view point, and data values. All of this information is readily available without the need to perform expensive visibility queries.

C. Delayed Visibility Queries

The notion of out of order traversal allows us to continue the hierarchy traversal even when a strict front-to-back traversal order stalls waiting for a visibility query. In essence, we can queue requests and continue the traversal out of order. Note that we should triangulate any non-empty leaf cell that the traversal does reach. After we traverse all possible out-of-order traversal paths we are left with a set of triangles and collection of cells for which visibility queries are required. We then render these triangles into the framebuffer in a single batch and execute a single visibility query against the hardware framebuffer (see section V). The algorithm is now repeated using the meta-cells we determine are visible.

It is important to note that the delayed visibility queries may refer to meta-cells that overlap or even hide each other. Due to the use of a full z-buffer for the rendering and visibility we can ensure the correct visibility results are returned (see section V).

D. The PHASE Framework

For the sake of clarity we refer to both cells and meta-cells as *nodes*. In addition, *visiting* a node will mean triangulating a cell or traversing a meta-cell.

The key steps in the progressive hardware accelerated extraction framework are:

We can take advantage of view coherency when just the view point changes between consecutive extraction passes. In essence, we can treat this case similar to another refinement loop. First, we initialize the set of potentially visible nodes with all the non-visible nodes (not just the root) from the previous view point, sorted front-to-back, relative to the current view point. We then clear the framebuffer, and project all the triangles we extracted for the previous view point. This time however, we project using the new view point. In other words, we use the extracted triangles as potential occluders. Due to the view coherency, very few previously non-visible cells will become visible and the loop will end quickly. We can also determine which of the previously extracted triangles are not visible from the new view point, and mark their cells as non-visible. However, this will require a special step that will not affect the final image, only the number of triangles rendered. As such, the cost of removing the hidden triangles may be higher than the cost of rendering them.

Algorithm 1
PHASE FRAMEWORK

```
potentialNodes ← rootNode
repeat
  traverse data (Section IV-B) and determine:
  · visibleNodes ← visible non-empty leaf nodes
  · possibleNodes ← partial occluded nodes
  Triangulate visibleNodes
  Render visibleNodes to framebuffer
  Query hardware framebuffer (Section V):
    (one query for all possibleNodes)
  · potentialNodes ← visible nodes
until potentialNodes is empty
```

The following sections explore, in detail, the notions of out of order traversal and visibility queries against the hardware framebuffer.

IV. THE TRAVERSAL PIPELINE

With the PHASE framework concept in mind, we now formalize the theory of visibility propagation using the notions of loose traversal order, and delayed visibility queries. We also present our particular implementation approach and prove its correctness.

In section III we established that some of the nodes can be determined to be visible *a priori* and can thus be traversed out of order. However, we need a mechanism to serialize this process so that we can establish some of its properties and devise a method to execute it. This will allow us to determine which node can be visited first and which nodes traversal must be delayed until some prerequisites are met.

We first note that a strict front-to-back traversal of the data can be implemented as a stack. At each stage, the front node is examined and classified as either: *empty*, *vis-*

ible or *non-visible*. An empty node (*i.e.* one that does not intersect the isosurface) is skipped. A non-visible node is discarded or added to the *non-visible* set for use in the next iteration. A visible node, on the other hand, is either triangulated (a cell) or is traversed (meta-cell) and its eight children are push into the top of the stack in back-to-front order.

Consider again the scenario in Figure 3 and the states of the traversal stack (Figure 4). In step 1, the stack is initialized with the root node *R*. The stack top node, *R*, is then popped and after we determine it is visible, we push, in step 2, octants *A* through *H* onto the stack. Again, the traversal must stall as we need to determine if the node at the top of the stack (A_A) is visible. In fact, we have to stall the traversal and perform a visibility query for each node we pop from the traversal stack.

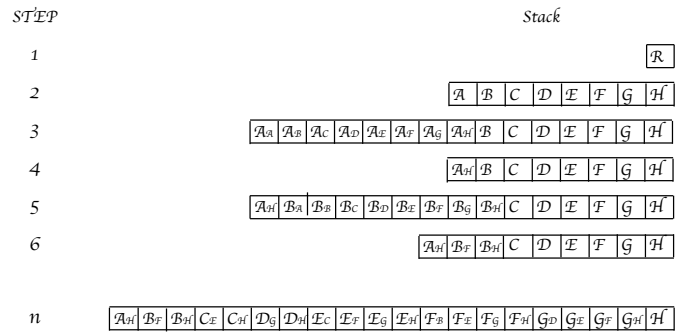


Fig. 4
OUT OF ORDER VIEW-DEPENDENT TRAVERSAL.

Using the notion of out of order traversal we can liken the traversal stack to a CPU pipeline. When a simple pipeline reaches a read from memory it must stall until the data is retrieved. An advanced compiler can anticipate this stall and schedule other operations, which do not depend on the result of the read, ahead of operations which do. Similar if we do not use a strict view-dependent traversal order, we can continue to visit the next nodes in the traversal pipeline that do not depend on the visibility results of the node at the head of the pipeline.

Consider again, scenario in Figure 3, with respect to our view point. Even before we start to traverse the data, we can determine that the root node, as well as nodes *A* though *G* (but not *H*) are all visible. Furthermore, we can determine that certain additional nodes must also be visible because they are on the visible boundary of the root node. Using this information, we can see (Figure 4) that the traversal does not need to stall in step 1, nor in step 2 as we know both the root node and node *A* are visible. Better yet, we also know that nodes A_A through A_G are also visible and thus we can traverse them without the need to perform a visibility query against the framebuffer. It does not matter if nodes A_A through A_H are empty or contain triangles. After we traverse the visible A_A though A_G nodes we can visit node A_H. However, in the case of node A_H, we must perform a visibility check as it may be hidden by triangles we extracted from the previously traversed nodes A_A though A_G. Yet, we do not need to stall the traversal pipeline, even at that step. Taking advantage of the notion of out-of-order traversal, we note that the next node in the pipeline, node *B*, is already known to be visible. As such we can traverse node *B* and replace it with a list of its children, sorted based upon the current view point (step 5). Note that we must maintain the general view dependent order and keep node A_H in front of all the children of *B*. Once again, we know from the initial check that nodes B_i are all visible except B_F and B_H. We can thus traverse these nodes, leaving node A_H at the front of the pipeline (step 6).

Continuing the above procedure, we can traverse 37 nodes before the traversal pipeline finally does stall. At that step we are left with 19 sub-octants and one full octant (*H*) in the pipeline. Assuming we did find triangles during our traversal, we update the visibility framebuffer by rendering the triangles we extracted, and then perform a single visibility query to determine which of the 20 nodes in the stalled pipeline are visible (see Section V).

The traversal pipeline can then resume by traversing the visible nodes while maintaining the front-to-back order in the pipeline. We stop the traversal once there are no more visible nodes in the pipeline (after a visibility query).

A. The Theory of Visibility Propagation

Now that we established *how* we can traverse the nodes out of order, we need to establish *when* are we allowed to. In other words, how do we establish that a node is visible without performing an explicit visibility query against the framebuffer. In the previous scenario, we were able to skip over 19 nodes during the traversal due to our initial examination of the scene. In effect, we were able to peel off the visible portion of the data volume's outer shell. We can follow the same procedure after every visibility query.

Let us again consider the initial examination of the scene and, this time, assume some of the nodes are empty. For example, assume node *E* is empty. We can then ask whether node *H* has became visible from our view point and, if so, what can be stated about any of its children H_i. It is clear that from view point V^1, we gain no new information about *H* in this case. However, from view point V^2, node *H* is visible as well as nodes H_A, H_B, H_D and H_E. On the other hand, the visibility of node H_E from some

particular view point somewhere between V^1 and V^2 may depend on the triangles we extract in node D.

Now, we must ask whether we can formulate a set of simple rules that will enable us to tell how the visibility of one node *may* affects the visibility of another node. We seek rules that answer this without resorting to a visibility query against the framebuffer. We could ask for a *certain* answer but this will require us to consider the exact coverage of the previous extract triangles. Furthermore, it will require us to stall the traversal pipeline until all the nodes, which may cover the node in question, are processed. On the other hand if we allow some leniency, such that a node may be classified as visible when it is not (but not the other way around) then we may be able avoid stalling the traversal pipeline. In this case, we may overshoot and traverse non-visible nodes, though we are guaranteed not to miss any visible triangles.

In other words, knowing the visibility status of the three front faces of a cube, what can we infer about the visibility of the inside and the back faces of that cube?

A.1 Basic Visibility Propagation Rules

We start with a few basic rules. These rules are somewhat trivial but nevertheless always true.

Consider the two-dimensional scenario depicted in Figure 5. As long as the point of view remains in the shaded area, *i.e.*, a corner view, only the *front* and *left* (from the eye view point) sides of the node are visible. Furthermore, if *both* sides are not visible, *e.g.*, obscure by objects closer to the eye point, then the back facing sides (*right* and *back*) are not visible as well.

We can deduce from the above case the following two basic rules:

Propagation rule 1: If all the viewer facing sides of a cell are occluded then so is the cell itself, and its back-facing sides.

Propagation rule 2: If any viewer facing side of cell is visible then the cell itself is visible.

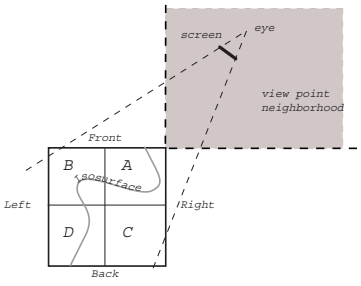


Fig. 5
A TWO-DIMENSIONAL VISIBILITY SCENARIO.

Consider now two different view points both of which are inside the shaded area as seen in Figure 6. The visibility of cell D from view point 1 cannot be determined before cells A and B are triangulated. If we disregard, for a moment, that cell D may be visible through cell C, we note that it is not possible to determine the *a priori* visibility of a cell for which some of the cells in front of it contain triangles (unless the there is another path through which the cell is determined to be visible). We formulate this with respect to the obstructing cell as:

Propagation rule 3: if a visible cell intersects the iso-surface then it is not possible to determine the visibility of the cell back facing faces *a priori*, *i.e.*, before the triangles are extracted, and we must perform an exact visibility check.

In this final case, the back-faces can only be marked as *Possible.*

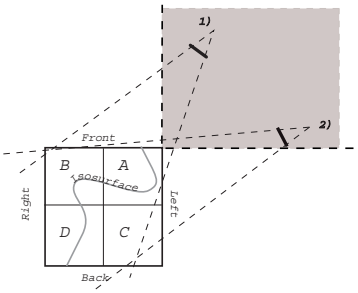


Fig. 6
TWO VIEW POINTS FROM THE SAME REGION.

A.2 General Visibility Propagation Rules

Two of the three basic rules specify when visibility propagation cannot continue while the third can help determine that a cell interior is visible. None of these rules enable us to propagate visibility information from the front faces to the back ones. The reason is that we can not make any such statements about the visibility propagation which will be true from *any* direction.

We can, however, consider approximation rules. Such rules can state the visibility status of the back faces in *most* cases. We should be able to use such rules as long as they are conservative, *i.e.*, they may define a face as visible when it is not but not the other way. But there is a cost associated with each such rule, it is the amount we pay for over reaching and traversing or triangulating cells which are hidden.

Returning to the second example, it is clear that in both cases cells A, B and C are all visible. Yet, while cell D may not be visible from view point 1, it is visible from view

point 2 due to the fact that cell C is empty. In general we can state that if an empty cell is [partially-] visible, then each of the back-facing sides may be visible from some view points in the shaded area. An approximation rule for this case can be stated as:

Propagation rule 4: If an empty cell is [partially-] visible, then each of the back-facing sides is visible.

The cost associated with this rule maybe too high in some cases. Consider, the application of rule 4 to a visible boundary cell. If the top of the cell is visible then according to this rule, so it the bottom. However, this means that the cell bellow it is also visible (its top is visible) and thus that cell's bottom is visible. It follows that if the entire column is made out of empty cells then all of them will be defined as visible. However, most of this column may in fact be hidden from view except its top, *i.e.*, this is clearly an over estimate in most cases.

A more conservative rule may state:

Propagation rule 5: If a face of an empty cell is [partially-] visible, then the two adjacent back faces are also visible while the visibility of the opposite face can not be determined without an explicit visibility check (*i.e. Possible*).

However, when we view a cell face on then the above rule maybe far from appropriate. In this case it is the opposite face which should be designated as visible. The other faces are likely to have a small footprint on the screen and thus it would be more cost effective to designate them as *Possible*:

Propagation rule 6: If we view an empty cell face on and that face is visible, then the opposite back face is visible. The other faces should be consider *Possible*.

One can devise more such rules, each with its own cost and pre-conditions. Determining an optimal set of such rule is an open question for future research.

B. Visibility Maps

It is obviously impractical to maintain the visibility state of all six faces for each and every node. When we consider the initial state, before the traversal starts, we note the need to establish the visibility state only for the boundary nodes. This is true in the general case as well. Better yet, the number of cells on the boundary cannot grow during the traversal. Based on this observation we can devise a way to implement the idea of visibility propagation using only two-dimensional structures we term visibility maps.

B.1 Definition and Validity

For simplicity, assume first that we deal only with cells (no meta-cells). We create three visibility maps, one for each front facing side of the volume, see Figure 7. Each map has the same number of cells as are on the volume side facing it.

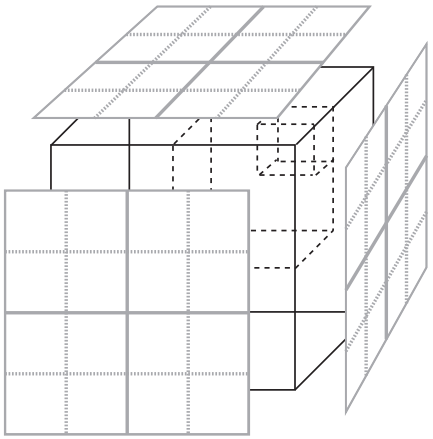


Fig. 7
A 3D VOLUME THE THREE SIMPLE 2D VISIBILITY MAPS.

Definition: At every stage of the traversal, the visibility state of each of the faces of the next node in the pipeline is reflected in the corresponding entry in the visibility map associated with that face.

Construction: We start by initializing all the visibility maps to *Occluded*. We then consider the visibility of the root node and all the cells on the its front facing faces. For each visible boundary node we set the appropriate entry corresponding visibility map to *Visible*.

Theorem 1: It is sufficient to maintain visibility state only in the entries of the three visibility maps. In other words, the visibility maps reflect the correct state of the next cell in the traversal pipeline.

To prove this theorem we first need the following two lemmas:

Lemma 1: If a node is the next to be visited in the pipeline then it must be on the current boundary.

Proof: Assume, in negative, that the node is not on the current boundary. It follows that none of its adjacent nodes have been visited. But this contradicts the requirement that we traverse the nodes in a front-to-back order. Remember that we consider a node out of order only if we know it is visible from at least one view point yet, we cannot know this for the current node as we did not visit any of its neighbors. ■

Recall that we visit a node only if it is visible. A node can be marked as visible only if it was determined to be visible based on a visibility query or if any of its three neighbors in front of it were both visible and empty.

Lemma 2: The visibility state of the next cell to be traversed is *Visible* if and only if that cell reside on the outer most boundary in the direction of the map which mark that cell as *Visible*.

Proof:

`If:` if the cell reside on the outer most boundary then there is no other cell between it and the map. Based on the map construction, if the map state is *visible* then the cell must be visible as well.

`Only if:` Assume, in negative, that the map state for the cell is *visible* but the cell does not reside on the outermost boundary with respect to that map. It follows that there is another cell on the boundary and is closer to the map. However, according to the view dependent traversal order, we should have visited that other cell before the current one and thus, remove it. The only case in which we would not removed a cell is if that cell is not visible, but this contradicts the assumption. If follows that no other cell may exist closer to the map than the current cell. ■

Proof: [Theorem 1]

The trivial case: Based on their construction, the entries in the three maps correspond one-to-one to the cells on the boundary of the volume. Consequently, setting the visibility state of a map entry to the state of the corresponding volume boundary cell satisfies the theorem.

The general case: Consider the general case where the next node in the traversal pipeline is node N_i. According to lemma 1, cell N_i must be on the boundary, and according lemma 2, its status can be *visible* only if the corresponding status in the visibility map is *visible.* It follows then that the visibility maps are sufficient to determine if the next cell in the pipeline is visible. ■

We can now generalize the visibility maps for any kind of a node. In this case, we construct and maintain a separate set of visibility maps for each level in the volume hierarchy. In other words, the octree structure of the volume can be reflected in the visibility maps as a quadtree.

B.2 Manipulation Rules

In the previous section, we defined and showed how to construct the three visibility maps. We argued that the visibility maps can be represented as quadtree. We now consider how to maintain these maps consistent throughout the pipeline traversal.

Consider a face visibility map of a three dimensional volume as depicted in Figure 8.

Visibility Map rule 1: A node is *occluded* if all of its four children are occluded.

Visibility Map rule 2: A node is *visible* if any of its four children is visible.

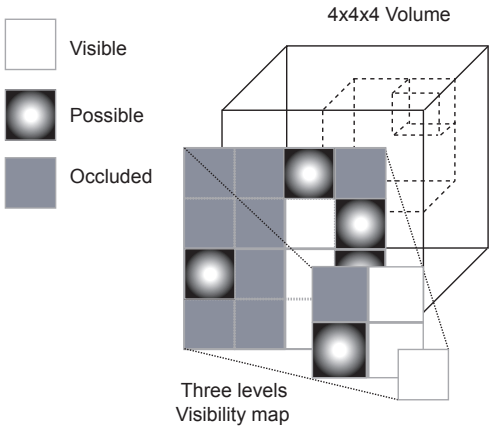


Fig. 8
A 3D VOLUME AND ONE OF ITS 2D QUADTREE VISIBILITY MAP.

Last, we need to address the cases of a visible cell that contains triangles and the visibility status of the cells behind it.

Visibility Map rule 3: If a *visible* cell contains triangles then the back facing sides are marked *possible*.

Visibility Map rule 4: If none of the front facing sides is *visible* and at least one is *possible* then the cell is also *possible*.

Using the binary values: *Occluded* = 00, *Possible* = 01 and *Visible* = 11 the cell visibility is a *bitwise OR* of the visibility of the front faces.

B.3 Initialization

In the simple case where the bounding box of the data is visible, all the visibility maps entries are initialized to *Visible.* However, if portions of the data bounding box are not visible then the initialization step is slightly more involved.

For each visibility map, we first determine which portion is visible by projecting the screen on the map plane and clipping it to the map area. Alternatively, the map's four coordinates can be projected and clipped to the screen and then back-projected to the map. Once the visibility portion is determined we scan convert it onto the map's lowest level, *i.e.*, the highest resolution. The map is then updated in a recursive fashion, up until the root entry is reached.

Alternatively, the visibility of root node can be determined using the procedure outlined in section IV-B.6. In this case the *Possible* list is initialized with only the root node.

B.4 Updates

When the visibility of a meta-cell or a leaf cell is set, each visibility map is updated according to the spatial position of the cell on each map and its position in the hierarchy. Each map entry then propagates the change to its parent. The parent entries are called recursively and are ordered to recompute their values based on the values of their children. This climbing continues until reaching the root of the map or finding an ancestor for which the value does not change. The value of a parent entry is set to an OR of the values of its children.

B.5 Rebuilding the Visibility Maps

Theorem 1 asserts that the visibility maps are correct for the next node in the pipeline. This assertion stands as long as we keep the front-to-back traversal order. Once the pipeline is empty, if there are still nodes in the *Possible* list, then we need to perform a visibility query on all of these nodes and repeat the traversal. Care must be taken as the visibility maps from the previous iteration can not be use as is. The maps reflect the correct visibility status only as long we continue to traverse in a front-to-back order but the next iteration will start with a node which, by definition, is in front of the last node we examined. We must, therefore, reinitialize the visibility maps. The question is what should we initialize them to?

The answer is two-fold. First, the visibility maps are initialized to *Occluded*. Second, before processing the next node in the pipeline we check if we have already performed a visibility query on it. If we did, then rather then *referring* to visibility maps, we use the result of the visibility query and *update* the visibility maps. Note that this accrues only for nodes from the previous iteration. New nodes that are encounter during the current traversal are processed as usual and their visibility status is determine by consulting the visibility maps. Caution must be taken not to update the visibility maps with the visibility information of a node, *before* the node reaches the front of the pipeline, *e.g.*, initializing the visibility maps with all of the results from the visibility query at the start of the next traversal iteration.

B.6 Visibility Query

In section V, we discuss how to perform a visibility query using the graphics hardware. Here, we are interested with *what* should this query determine, not *how*. The simplest query need only determine if *any* portion of a node is visible. A more refined query will determine which of the three front faces is visible and update only the appropriate visibility maps. In both cases, however, the entry in the map's quadtree is updated as well as its entire subtree. Instead, our visibility queries determine the visibility status of each of the *cells* on the front faces of the node. We can then update the visibility maps entries with more accurate information.

Recall that this is similar to the situation we had at the start when we had only the root node. We can, therefore, use this procedure to initialize the visibility maps at the start as well. All we need is to initialize the maps to *Occluded*, initialize the *Possible* list with the root node, perform this visibility query and start the loop.

C. Other Viewing Regions

Until now, we assumed that three of the exterior faces are facing toward the user. In this case, only three visibility maps are needed, one for each pair: left/right, top/bottom and front/back. However, when the view point is such that only one or two faces are visible then we must use both maps of the two occluded faces. As the algorithm descends down the data hierarchy, the traversal order in each of the two portions of the data (see Figure 9) will be different, thus a different visibility map must be maintained for each case. This does not pose a problem as the traversal direction determines which of the two visibility maps should be used for each cell. Furthermore, the two appropriate maps are initialized to *Occluded* as no portion of them is visible at the start. Note, that as the algorithm discover *Visible* or *Possible* cells, some of the entries in these maps will change.

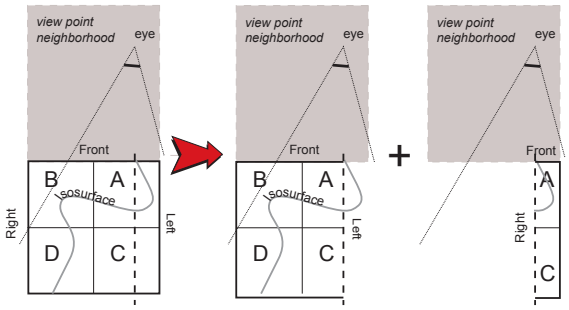


Fig. 9
VIEWING THE DATA FROM THE SIDE.

A similar situation occurs when the view point is inside the volume. Consider a two-dimensional case depicted in Figure 10. The view point is inside the data and, from the start, the only known information is that the cells just in front of it in the viewing direction are visible. This case translates to an orthogonal projection of the image onto the facing sides of the volume's bounding box. Again, if

the image plane is perpendicular to a face, then the face is reset to *Occluded*. In many cases, the image's absolute size inside of the data will shrink to less than the size of a single entry at the highest resolution of the map. In this case, we will mark a single entry as *Visible*.

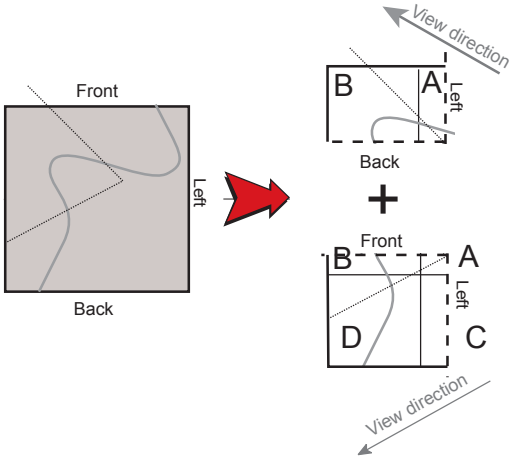


Fig. 10
VIEW POINT INSIDE THE DATA VOLUME.

V. HARDWARE ACCELERATED VISIBILITY QUERIES

After the initial pass, where most of visible cells the iso-surface intersects are identified, we are left with a collection of nodes that are marked as possibly visible. These nodes may overlap and even completely overshadow each other. However, we do know their relative front to back order.

In order to quickly determine which of these *Possible* nodes are visible from the current direction, PHASE uses the graphics hardware accelerator. The nodes which were classified as visible can be interrogated using the same visibility propagation method used in the initial approximation step. The process then continues recursively by clearing the framebuffer, projecting all the triangles and checking which of the *Possible* nodes becomes visible. Note that a *Possible* node that did not pass the visibility test in one iteration, may become visible in a later iteration if it was hidden by another node which, at one time, was determined to be visible.

Procedure **??** takes advantage of the rendering speed as well as the z-buffer provided by the graphics accelerator. The order in which the triangles, as well as the meta-cells, are projected is not important. There are, however, several drawbacks to this approach. First, it uses the actual screen window the user sees. Second, we project each of the previously extracted triangles in each iteration. Third, it is time consuming to read the entire framebuffer in each pass.

Using the displayed window can be avoided by using either the back-buffer or a separate off-screen buffer, *e.g.*, pbuffer. One can also avoid the need to project all the triangles by turning off the write to (but not the compare against) the depth buffer before projecting the nodes. This preserves the z-buffer and eliminates the need to re-project the triangles. The meta-cells will still be drawn correctly with respect to the triangles. However, as the write to the depth buffer was turned off, the meta-cells may not be drawn correctly with respect to each other. The solution is to project the nodes in a *back-to-front* order. Note that the nodes are extracted in a front to back order and thus, there is no need to sort the cells prior to the projection, only to traverse the node list in reverse order. Using this approach, closer nodes will overwrite more distant cells and the final image will be correct.

There are two ways to address the issue of slow z-buffer readback. Some graphics accelerators support the histogram extension that returns the number of time each pixel value is repeated in the framebuffer (or a sub region of it). The problem with this approach is that it is not widely supported and even for the hardware that support it, it may take a long time to compute a histogram for a large area of the framebuffer. Our approach is to compute, see Listing **??**, a three-dimensional bounding box of all the meta-cells that were projected. We then project this box onto the screen and find the bounding box (in screen space) of its projection. We can then read back only that region of the framebuffer and compute the histogram in software.

A. Visualization of the Progressive Approximation

If the user is not interested in intermediate approximation then PHASE can use the back buffer for the visibility tests during the progressive extraction. Alternatively, if intermediate approximations are preferred, an auxiliary buffer can be used to accelerate the progressive process as depicted in Figure 12.

The first pass, described in Section IV-A, does not involve any projection of triangles and thus the auxiliary buffer is not used. The initial approximation A_0 is then projected on the back buffer and subsequently is shown to the user after the buffers swap.

The second iteration begins with the projection of the first approximation A_0 onto the auxiliary buffer followed by visibility check of all the unclassified nodes. Based on this visibility test more triangles are generated, D_1, to augment the first approximation. As the back buffer does not contain any useful information, it is cleared and the

```
procedure fast_hw_visibility_test()
{
   // set up the depth buffer
  disable shading

  disable color buffer write
  draw( new_triangles )
  enable color buffer write;

  // project possible nodes back to front
  reset bbox3d
  clear color buffer to 0
  disable depth buffer write;
  for ( node in possible_list ) {
     color = node.index+1;
     draw( node.boundingBox )
     bbox3d.extend( node.boundingBox )
  }
  enable depth buffer write;

  // read back visibility results
  bbox2d = screen bounding box( bbox3d )
  pixels = read framebuffer( bbox2d )
  hist = histogram( pixels )

  // visibility status
  for ( index=1; index<hist.size(); index++ )
     if (hist[index] > 1)
        nodes[index-1].visible = true
}
```

Fig. 11
HARDWARE ASSISTED VISIBILITY TEST

next approximation $A_1 = A_0 + D_1$ is projected. The front and back buffers are swapped again to present the user with the next approximation.

From the third iteration on, we can take advantage of the information already in the auxiliary and back buffers. The auxiliary buffer already contains the projection of the initial approximation, and thus only the triangles found in the previous iteration need to be projected. The back-buffer already contains the second approximation A_1, there is no need to re-project all the triangles, only the newly extracted triangles D_2.

B. Progressive Extraction While the View Point Changes

Assume the user has changed the point of view slightly, *e.g.*, rotated the data slightly during the time PHASE extracted the initial approximation. Due to view coherency, the initial approximation of the isosurface for the new view point should be very similar to the one just extracted. Consequently, it should be sufficient to re-project this old approximation using the new view point as though it was the initial extraction from the new view point, and then continue with the next pass. Care must be taken, though, to reorder the nodes based on the new view point.

Step	Front	Back	Aux.	Operation
1	n/a	A_0	n/a	
				swap Front/Back
	A_0	n/a		
2			A_0	init Aux.
				compute D_1
				render to Back
		$A_0 + D_1$		
		A_1		swap Front/Back
	A_1	A_0		
3				render D_1 to Aux.
			$+D_1$	
			A_1	compute D_2
				update Back
		$+D_1 + D_2$		
		A_2		swap Front/Back
	A_2	A_1		
n	A_{n-1}	A_{n-2}	A_{n-2}	render D_{n-2} to Aux.
			$+D_{n-2}$	
			A_{n-1}	compute D_{n-1}
				update Back
		$+D_{n-2} + D_{n-1}$		
		A_n		swap Front/Back
	A_n	A_{n-1}		

Fig. 12
PROGRESSIVE ISOSURFACE EXTRACTION.

Moreover, if the view point did not change much then we can approximate the visibility status by using the back depth buffer as an approximation of the projection of the triangles based on the new view point. Instead of the back buffer, we use it as is and continue with the visibility test of the cells in the *Possible* list. Note that the *Possible* nodes are projected according to the new viewpoint onto a depth buffer which is correct for the previous viewpoint, *i.e.*, a two view position back of the new one. As long as the viewpoint does not changedrastically this approximation works well.

VI. RESULTS

We have implemented the PHASE framework on a graphics PC running Linux. For the sake of comparison, we also have implemented within a common application the standard octree-based algorithm and the SAGE

method. For the experimentations, we used a 1.9 GHz AMD Athlon processor with 1.5 GB of DDR DIMM memory and a GeForce4 Ti 4600 graphics board.

We applied these methods to three different sections of the Visible Woman CT data: one 512x512x209 2-byte values dataset from the head section, one 512x512x857 2-byte values dataset from the body section and one 512x512x617 2-byte values dataset from the legs section. The extracted isosurfaces were rendered to a 512x512 window using hardware accelerated OpenGL.

A. Fixed Point of View

First, we must consider the classical isosurface extraction for two different values ($v = 600.5$ for the skin and $v = 1224.5$ for the bones) and with a fixed point of view showing the whole isosurface to the user from outside the dataset. This is the typical case when the user sets a new isovalue to explore the dataset.

As a reference, Table I show the timings for a full isosurface extraction using the octree-based algorithm and for a view-dependent isosurface extraction and rendering using the SAGE method.

During a progressive isosurface extraction with the PHASE method, the first iteration takes about 1 s both for the skin (for about $500,000$ triangles) and the bones (for about $400,000$ triangles). After this first iteration, about 85% of the final image has already been computed. The second iteration takes about 0.6 s and leads to more than 97% of the final image. Finally, a few more iterations, taking about 0.3 s each are required to get to full convergence.

For an isovalue change, the view dependent SAGE method can be up to two times faster than the full isosurface extraction. The slowest is PHASE since it needs at least two iterations to get a good image quality.

B. Moving the Point of View

The full isosurface extraction leads to a huge number of triangles than can overwhelm the graphics accelerator capacity. The framerate when moving the point of view is limited to 5fps for the skin and 3fps for the bones.

The SAGE method needs to recompute everything for every new point of view. The framerate is limited to 1fps.

The progressive approach of PHASE allows to only extract new triangles (with iterations of 300ms). We can then get 2fps.

This dataset is obviously no more a good example!

C. Closeup views

When we are inside or very close to the dataset, PHASE reacts very fast while SAGE needs more time.

VII. Conclusions

Acknowledgments

References

[1] W. E. Lorensen, "Marching through the visible man," in *Proceedings of Visualization 1995*, Oct. 1995, pp. 368–373.

[2] U. Tiede, T. Schiemann, and K. H. Höhne, "High quality rendering of attributed volume data," in *Proceedings of Visualization 1998*, Oct. 1998, pp. 255–262.

[3] J. M. Favre, "Towards efficient visualization support for single-block and multi-block datasets," in *Proceedings of Visualization 1997*, Oct. 1997, pp. 423–428.

[4] P. D. Heermann, "Production visualization for the asci one teraflops machine," in *Proceedings of Visualization 1998*, Oct. 1998, pp. 459–462.

[5] M. Lanzagorta, M. V. Kral, J.E. Swan II, G. Spanos, R. Rosenberg, and E. Kuo, "Three-dimensional visualization of microstructures," in *Proceedings of Visualization 1998*, Oct. 1998, pp. 487–490.

[6] C. R. F. Monks, P. J. Crossno, G. Davidson, C. Pavlakos, A. Kupfer, C. Silva, and B. Wylie, "Three dimensional visualization of proteins in cellular interactions," in *Proceedings of Visualization 1996*, Oct. 1996, pp. 363–366.

[7] Martin Bertram, Mark A. Duchaineau, Bernd Hamann, and Kenneth I. Joy, "Bicubic subdivision-surface wavelets for large-scale isosurface representation and visualization," in *Proceedings of Visualization 2000*, Oct. 2000, pp. 389–396.

[8] Martin Bertram, Daniel E. Laney, Mark A. Duchaineau, Charles D. Hansen, Bernd Hamann, and Kenneth I. Joy, "Wavelet representation of contour sets," in *Proceedings of Visualization 2001*, Oct. 2001, pp. 303–310.

[9] W.E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163–169, July 1987.

[10] M. Giles and R. Haimes, "Advanced interactive visualization for CFD," *Computing Systems in Engineering*, vol. 1, no. 1, pp. 51–62, 1990.

[11] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," *Computer Graphics*, vol. 24, no. 5, pp. 57–62, November 1990.

[12] R. S. Gallagher, "Span filter: An optimization scheme for volume visualization of large finite element models," in *Proceedings of Visualization '91*. 1991, pp. 68–75, IEEE Computer Society Press, Los Alamitos, CA.

[13] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," *ACM Transactions on Graphics*, vol. 11, no. 3, pp. 201–227, July 1992.

[14] T. Itoh and K. Koyamada, "Isosurface generation by using extrema graphs," in *Visualization '94*. 1994, pp. 77–83, IEEE Computer Society Press, Los Alamitos, CA.

[15] H. Shen and C. R. Johnson, "Sweeping simplicies: A fast isosurface extraction algorithm for unstructured grids," *Proceedings of Visualization '95*, pp. 143–150, 1995.

[16] T. Itoh, Y. Yamaguchi, and K. Koyyamada, "Volume thining for automatic isosurface propagation," in *Visualization '96*. 1996, pp. 303–310, IEEE Computer Society Press, Los Alamitos, CA.

[17] Y Livnat, H. Shen, and C. R. Johnson, "A near optimal isosurface extraction algorithm using the span space," *IEEE Trans. Vis. Comp. Graphics*, vol. 2, no. 1, pp. 73–84, 1996.

[18] J.S. Painter, P. Bunge, and Y. Livnat, "Case study: Mantle convection visualization on the cray t3d," in *Visualization '96*. 1996, pp. 409–412, IEEE Computer Society Press, Los Alamitos, CA.

Dataset	Method	Triangles	Extract	Render	Image quality
Head (skin)	Octree	1,430,824	1.3s	0.2s	100%
	SAGE	200,000	1.0s (both)		100%
	PHASE (1 it.)	520,000	0.2s (both)		91%
	PHASE (2 it.)	610,000	0.3s (both)		98%
	PHASE (10 it.)	640,000	0.7s (both)		100%
Head (bones)	Octree	2,207,592	2.0s	0.3s	
	SAGE	150,000	1.0s (both)		
	PHASE		s (%) + s (%)		
Body (skin)	Octree	12,105,624	10.8s	1.7s	
	SAGE	120,000	2.0s (both)		
Body (bones)	Octree	8,221,990	7.3s	1.1s	
	SAGE	70,000	3.0s (both)		
Legs (skin)	Octree	3,873,622	3.5s	0.5s	
	SAGE	170,000	2.0s (both)		
Legs (bones)	Octree	2,328,940	2.0s	0.3s	
	SAGE	75,000	1.2s (both)		

TABLE I

ISOSURFACE TIMINGS FOR THE OCTREE-BASED METHOD.

[19] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno, "Optimal isosurface extraction from irregular volume data," in *Proceedings of IEEE 1996 Symposium on Volume Visualization*. 1996, ACM Press.

[20] Y. Livnat and C. Hansen, "View dependent isosurface extraction," in *Visualization '98*. October 1998, pp. 175–180, ACM Press.

[21] Y. Livnat, C. D. Hansen, and C. R. Johnson, "Isosurface extraction for large-scale datasets," in *In Proceedings of Scientific Visualization - Dagstuhl 2000*, ed. Frits Post, Ed., 2001.

[22] Ned Greene, "Hierarchical polygon tiling with coverage masks," in *Computer Graphics*, August 1996, Annual Conference Series, pp. 65–74.

[23] Y. Livnat, *NOISE, WISE and SAGE: Algorithms for Rapid Isosurface Extraction*, Ph.D. thesis, University of Utah, Dec 1999.

[24] Jinzhu Gao and Han-Wei Shen, "Parallel view-dependent isosurface extraction using multi-pass occlusion culling," in *Parallel and Large Data Visualization and Graphics*. Oct 2001, pp. 67–74, IEEE Computer Society Press.

[25] Zhiyan Liu, Adam Finkelstein, and Kai Li, "Progressive view-dependent isosurface propagation," in *Proceedings of Vissym'2001*, 2001.

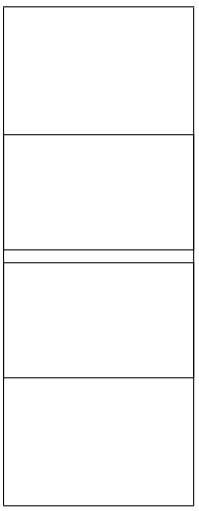
[26] Xiaoyu Zhang, Chandrajit Bajaj, and Vijaya Ramachandran, "Paralel and out-of-core view-dependent isocontour visualization using random data distribution," in *Joint Eurographics — IEEE TCVG Symposium on Visualizatation (VisSym-02)*, D. Ebert, P. Brunet, and I. Navazo, Eds., 2002, pp. 1–10.

[27] R. Westermann, L. Kobbelt, and T. Ertl, "Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces," *The Visual Computer*, vol. 15, no. 2, pp. 100–111, 1999.

[28] Benjamin Gregorski, Mark Duchaineau, Peter Lindstrom, Valerio Pascucci, and Keneth I. Joy, "Interactive view–dependent rendering of large isosurfaces," in *Visualization '02*. 2002, pp. 475–482, IEEE Computer Society Press.

[29] F. Bernardini, J.T. Klosowski, and J. El-Sana, "Directional discretized occluders for accelerated occlusion culling," *Computer Graphics Forum*, vol. 19, no. 3, 2000.

[30] Ned Greene, Michael Kass, and Gavin Miller, "Hierarchical z-buffer visibility," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, pp. 231–238, ACM Press.



Yarden Livnat

Xavier Cavin

Charles Hansen