Interactive Point-Based Isosurface Extraction

Yarden Livnat* Xavier Tricoche†

Scientific Computing and Imaging Institute,
University of Utah

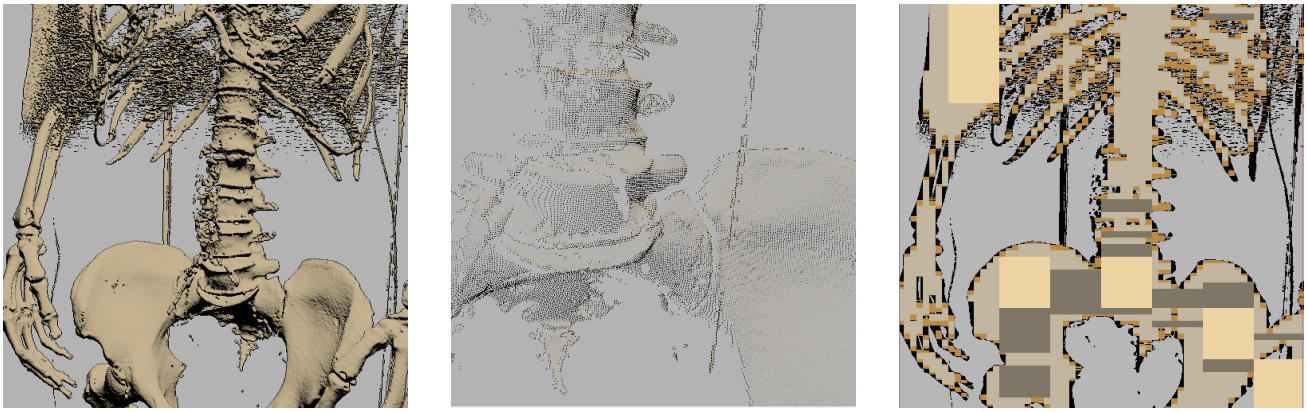


Figure 1: *Left:* A section of the visible female skeleton. *Middle:* A closeup view of the extracted points. *Right:* The final visibility mask. The color represent different levels of the mask hierarchy

ABSTRACT

We propose a novel point-based approach to view dependent isosurface extraction. We introduce a fast visibility query system for the view dependent traversal, which exhibits moderate memory requirements. This technique allows for an interactive interrogation of the full visible woman dataset (1GB) at four to fifteen frames per second on a desktop computer. The point-based approach is built on an extraction scheme that classifies different sections of the isosurface into four categories, depending on the size of the geometry when projected onto the screen. In particular, we use points to represent small and sub-pixel triangles, as well as larger sections of the isosurface whose projection has sub-pixel size. To assign consistent and robust normals to individual points representing such regions, we propose to compute them during post processing of the extracted isosurface and provide the corresponding hardware implementation.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface Algorithms;

Keywords: Isosurface, point-based, view-dependent, large datasets, interactive

1 INTRODUCTION

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 scalar data. These contours isolate surfaces of interest, focusing the analysis on important features in the data, such as material boundaries, while suppressing extraneous information.

Isosurface extraction poses a unique challenge in that no geometry exists before the user provides an isovalue. Furthermore, the user may change the isovalue often, and any geometry extracted based on the previous isovalue should be discarded. Consequently, only a limited amount of meta data can be generated and stored for use during an interactive session. In the case of large datasets this problem is emphasized because the size of the scalar information leaves little memory for additional data structures. In fact, the size of very large datasets can overwhelm the extraction and the rendering systems. For example, the isosurface corresponding to the skeleton in the *Visible Woman* dataset (1GB) contains over eleven million triangles, which clearly over strains what todays graphics cards can handle interactively.

To address this issue, Livnat and Hansen [21] proposed an output sensitive approach based on view-dependent extraction of the isosurface. Others have since proposed various acceleration techniques using massive parallel machines or the graphics hardware. Yet, none of these approaches address the need for an interactive extraction of large datasets using a single desktop computer. This paper presents a view-dependent point-based isosurface extraction algorithm (PISA) that permits to extract and render the isosurfaces of data sets, fitting in the memory of a desktop computer, at interactive frame rates.

This paper is structured as follows. Section 2 reviews earlier work on isosurface extraction, including view-dependent and point-based techniques. In section 3, we discuss issues that arise due to the size of the data and the way it is viewed, and we outline our general approach. Section 4 describes the point-based technique in detail, and section 5 presents the algorithms. Results and performances are shown in section 6 followed by conclusion and future work.

*e-mail: yarden@sci.utah.edu
†e-mail: tricoche@sci.utah.edu

2 PREVIOUS WORK

Early work [23, 11, 27, 8, 28, 14, 26, 15] on isosurface extraction focused on a complete isosurface computation and exhibits a worst case time complexity of $O(n)$, where n is the size of the data. Cignoni *et al.* [5] presented an optimal isosurface extraction method based on the *span space* introduced by Livnat *et al.* [22]. As the size of the datasets grows, however, new challenges arise. First, the size of the datasets may be larger than the available memory. Second, the size of the extracted isosurfaces can overwhelm even the most advanced graphics system. The problems induced by the size of the data are addressed by out of core isosurface extraction methods [3, 4, 2], which aim at maintaining the data on disk, and loading only relevant sections. Another research effort, view dependent isosurface extraction, targets the size of the extracted isosurface by extracting only the visible portion of the isosurface.

2.1 View-dependent Isosurface Extraction

View-dependent isosurface extraction was first introduced by Livnat and Hansen [21]. Their approach is based on a front-to-back traversal of the data, while maintaining a virtual frame-buffer of all extracted triangles. The virtual frame-buffer is used during the traversal to cull sections of the dataset, which are hidden from the given viewpoint by closer parts of the isosurface. The authors also proposed a particular implementation for the virtual frame-buffer, and for performing visibility queries against this frame-buffer. Their method, termed WISE, is based on shear-wrap factorization [17] and Greene's coverage maps [12]. Gao and Shen [9] presented a distributed parallel view-dependent approach that uses multi-pass occlusion culling with an emphasis on load balancing. Gregorski *et al.* [13] proposed a recursive tetrahedral mesh refinement scheme based on longest edge bisection. Their method requires a rather long preprocessing step, special data layout on disk, and a 1 to 4 ratio between the size of the data and the meta-data.

Recently, Gao and Shen proposed [10] a fast hardware-assisted view-dependent approach that takes advantage of NVidia accelerated occlusion culling queries, and of a new spherical partition scheme they introduced. Although the paper features fast extraction examples, the datasets used are fairly small (*e.g.*, a 40MB subsampled version of the *Visible Woman*'s legs section) and the provided information refers only to a single view distance. In addition, the paper does not discuss the memory requirements of the spherical tree, or effects of the various tree parameters on the size of the tree and the overall performance of the algorithm.

A ray casting approach was presented by Parker *et al.* [25, 24]. The Real Time Ray Tracer (RTRT) approach lends to large shared memory machines due to the parallel nature of ray casting. An additional benefit of the ray casting approach is the ability to generate global illumination effects such as shadows. Liu *et al.* [19] also used ray casting but instead of calculating the intersection of each ray with the isosurface, their rays are used to identify active cells. These active cells are then used as seeds in the more traditional isosurface propagation method.

Zhang *et al.* [29] presented a parallel out-of-core view dependent extraction method. In this approach, sections of data are distributed to several 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 extracts its visible portion of the isosurface. These authors noted that updating the occlusion map during the traversal is expensive, even when using hierarchical occlusion map. They opted to rely on the first occlusion map approximation, and not update the occlusion maps.

2.2 Point-Based Methods

Points have received an increasing interest in the computer graphics community in recent years as an alternative type of display primitive. They prove especially appealing for the rendering of complex geometric models of large size. The point-based approach was pioneered by Levoy and Whitted [18] who advocated the use of points as a universal meta-primitive for modeling and rendering. The first point-based method for isosurface extraction goes back to an early work, the *Dividing Cubes*, by Cline et al. [6]. Using a view-dependent approach, grid cells are subdivided until they reach a sub-pixel size in screen space and can be rendered as individual points. More recently Ji et al. [16] proposed to use points to achieve interactive non-photorealistic rendering of isosurfaces for remote visualization of large data sets. However this method assumes the computation of the triangulated isosurface at interactive frame rates by a server with high computation power before the rendering step. Closer to our approach is the work presented by Co et al. [7]. Their method, termed *Iso-splatting*, combines a sparse volume sampling (a single point per voxel), a local correction step to project each point on the isosurface, and a point splatting technique [30] to permit fast isosurface extraction. A somehow similar technique was used by Baerentzen and Christensen [1] who render the vertices of the *Marching Cubes* triangulation. Our algorithm improves on both methods by embedding a view-dependent approach in the sampling phase which significantly speeds up the extraction while preserving moderate memory requirements.

3 OVERVIEW

The aim of this work is to achieve *interactive* isosurface *interrogation* of *large datasets* using a *single desktop computer*. In the following, we briefly define each of the above terms within the scope of our work, along with the issues they raise and the choices we made to address them.

Interactive interrogation We define interactive interrogation as the ability to dynamically change the isovalue, and to view the extracted isosurface from various directions at a rate of several frames per second. In the context of our view-dependent approach, we allow for viewing of a partial extraction (which we term *Incomplete Reconstruction*) from other view points, as seen in Figure 1(Middle).

Large datasets We strive to work with the largest possible datasets as long as the data and the required data structure fit into the memory of the computer. A 2GB memory computer, for example, can easily handle a 1GB dataset with a 250MB additional overhead. Several issues arise when dealing with such datasets: the memory footprint of the data, the number of triangles of the extracted isosurface, and the dimension of the individual triangles.

- *Size of data*: Accelerating the search phase requires additional meta data, such as a hierarchical minmax tree. However, the Octree used in previous methods, while suitable for small datasets, requires too much memory, even for the pointerless version (BON-Tree), and a lowest level of $2 \times 2 \times 2$ cells [27]. The problem with the Octree stems from its low branching factor (two in each direction). To address this issue, we use a shallow tree in the form of nested grids. Each node in the tree represents a sub grid, and all grids at level i have the same $N_i \times M_i \times P_i$ cells.

- *Number of triangles*: Isosurfaces extracted from large datasets can contain a large number of triangles. The skeleton of the visible woman, for example, contains about 11 million triangles. These massive isosurfaces pose problems in the extraction time (interactivity), their memory footprint (bandwidth

to the graphics hardware), and their rendering (graphics hardware capabilities). We employ an output sensitive approach, namely view-dependent isosurface extraction. In addition to extracting only the visible portion of the isosurface, we allow the user to view the partial extraction from other viewpoints (at $\geq$100 frames per second). Thus, both extraction and rendering time can be reduced dramatically, since the number of geometric primitives is decreased by orders of magnitude.

- *Size of triangles*: Large isosurfaces typically exhibit sub-pixel triangles. In the case of a full isosurface extraction, sub-pixel triangles are a waste of time and memory resources. In the case of incomplete reconstruction, the result can be bad aliases, and sections of the isosurface disappearing. Our solution consists in using points to represent small triangles and meta cells (sub volumes). Points have advantages over triangles, such as reduced aliases, and points are visually persistent.

Desktop computers Until recently, the Utah Real Time Ray Tracer (RTRT) using ray casting [25] on massive parallel supercomputers was the only option for interactive extraction and visualization of large isosurfaces. We want to provide scientists with the ability to interactively extract and visualize isosurfaces from large datasets on their desktop computer. Modern desktop machines can have a large amount of memory (4GB), along with two or more processors and state-of-the-art graphics hardware. Furthermore, modern processors contain vector unit(s) that can be taken advantage of for visualization applications.
The highly parallel nature of modern graphics hardware and its somewhat limited programmability, suggest to use it for visibility queries in a view-dependent approach. Until recently, this approach was not feasible due to the long latency in receiving replies from the graphics hardware. Livnat *et al.* [20] and Gao and Shen [10] proposed methods to overcome the latency issue, but neither method was shown to work on large datasets.
Vector units provide hardware acceleration for parallel operations such as matrix vector multiplication used in the screen projection. Thus, we opted for a highly optimized software-based visibility query while taking advantage of the two processors and their vector units. We still leverage the graphics hardware for the rendering phase and screen-based computation of the normals, described in section 5.3. The post-processing does not suffer from the latency issue because the computed normals can be left on the graphics hardware and reused for shading.

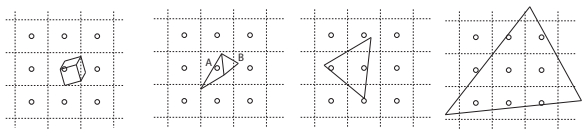
4 Point-Based Approach to View Dependent Extraction

Traditionally, isosurfaces were represented as a large collection of triangles. For large isosurfaces, however, the average projection of the triangles has sub-pixel size. Furthermore, for large datasets, the same holds true for whole sections of the volume. In the following, we propose to use point primitives to represent small triangles and even replace the triangular representation.

4.1 Classification

We consider the following four cases. Note that in all cases we can delay the normal computation to a post-processing step as described in section 5.3. However, when considering large triangles (close views) we can take on the corresponding normals for the rendering stage.

Far view From a far enough viewpoint, a sub-volume of data projects to a single pixel, see Figure 2(a). If the isosurface passes through the sub-volume the corresponding section will be seen as a single pixel and can be represented by a point.



(a) Far (b) Mid (c) Near (d) Closeup

Figure 2: View Classification

Mid view Closer to the isosurface, a data cell may project onto more then one pixel, but the extracted triangles may still have sub-pixel size as shown in Figure 2(b). From the point of view of memory utilization and available bandwidth to graphics hardware, it is clear that a single point is cheaper than a full triangle.
There is in this case another important issue to consider, namely, incomplete reconstruction. This corresponds to the ability to change the viewpoint without recomputing the isosurface thus allowing for fast ($\geq$100 frames per second) manipulation of large and complex isosurfaces. Recall that in view-dependent isosurface extraction only the visible portion is extracted. Only those triangles that cover the center of a pixel are included. Consider the case when the view point changes slightly but the isosurface is not recomputed. The sub-pixel triangle A that covered the center of the pixel is now off center, and is no longer displayed. Meanwhile, triangle B which should now cover the center of the pixel, was not included in the original extracted isosurface because it did not previously cover the center of any pixel. In this case, triangles lead to strong popping artifacts where portions of the isosurface disappear, and others reappear. We address this issue by replacing sub-pixel triangles that cover the center of a pixel with a point, using the fact that point primitives are persistent in OpenGL.

Near view Closer to the isosurface, triangles may cover more than one pixel. The aliases and artifacts discussed above still persist, though to a lesser degree. Small triangles do not disappear and reappear as they cover more than one pixel at a time, however, the area they cover can change significantly as they move since each single pixel makes up a large portion of their area. In addition, the local normal associated with such a triangle is sensitive to high frequencies in the surface geometry, and can cause artifacts when seen in a more global view.

Closeup view From a close viewpoint, surface triangles cover many pixels and may provide better performance than a large collection of triangles. In such cases, however, it may be more advantageous and accurate to replace the traditional triangle representation with a trilinear interpolation of the cell vertices, as was done by Parker *et al.* [25]. Representing the interpolated surface using points is the simplest way to achieve this.

5 The Algorithm

The proposed point-based view-dependent isosurface extraction relies on a hierarchical front-to-back traversal of the data using value and visibility-based pruning, as seen in Listing 1.

5.1 Value-Based Pruning

Value-based pruning is essentially the same as the Octree method used by Wilhelms and Van Gelder [27], although we use a GridTree (nested grids). Each of the tree nodes stores the minimum and maximum values of its children, which permits us to ignore the corresponding sub-tree if the isovalue is outside this range.

Listing 1: The Algorithm

```
view_isosurface( iso, extract )
{
  if ( extract ) {
    stack.push(root);
    while ( !stack.empty() ) {
      node = stack.pop();

      // prune based on value
      if ( iso ≤ node.min || node.max ≤ iso ) continue;

      if ( node.is_leaf() ) {
        // extract
        geom = extract(node);
        scan geom onto frame−buffer;
        isosurface += geom;
      }
      else {
        // prune based on visibility
        bbox = node.bounding_box();
        if ( !visible( bbox) ) continue;

        // push children in reverse order
        for (children in back to front order)
        stack.push(node.child());
      }
    }

    // see section 5.3
    compute_normals();
  }

  render();
}
```

5.2 Visibility-Based Pruning

Visibility pruning is achieved by traversing the GridTree in a front-to-back order with respect to the view point and traversing the children in a depth first order. When a leaf node is reached, the isosurface geometry is extracted and scanned onto a frame-buffer. During the traversal we perform a visibility query on each node that passes the value pruning test. The visibility query is done by projecting the bounding box of the node onto the frame-buffer and checking if it includes any part of the background. If this is not the case then the entire node is hidden by previously extracted geometry and thus can be pruned.

The visibility query is a key step in the visibility pruning process. On one hand, its overhead must be small to justify its use. On the other hand, it should be able to identify non visible cells efficiently. Note that if a node is visible then all the time spent in the visibility query is an overhead since we will need to perform visibility queries on its children as well. However, if the node is not visible then pruning the node can potentially save a lot of time and thus justify a longer query time. Furthermore, if a *non-visible* node is erroneously labeled as *visible*, the algorithm can correct itself later when checking the children. On the contrary, if a *visible* node is labeled as *non-visible* then the result of the algorithm will be wrong. In essence, the emphasis of the query should be on discovering if a node is potentially visible quickly while possibly overestimating a node's visibility.

5.2.1 Visibility frame-buffer

Visibility queries can be done in software or using the graphics hardware. For reasons described in section 3, we use a software-based visibility query. To determine if a node is visible, all pixels covered by its bounding box must be checked, *i.e.*, for each pixel, we only need to know if it was covered by previously extracted geometry. To that end, only a one bit per pixel visibility frame-buffer is needed. Figure 3 shows two examples of isosurfaces with their visibility masks.

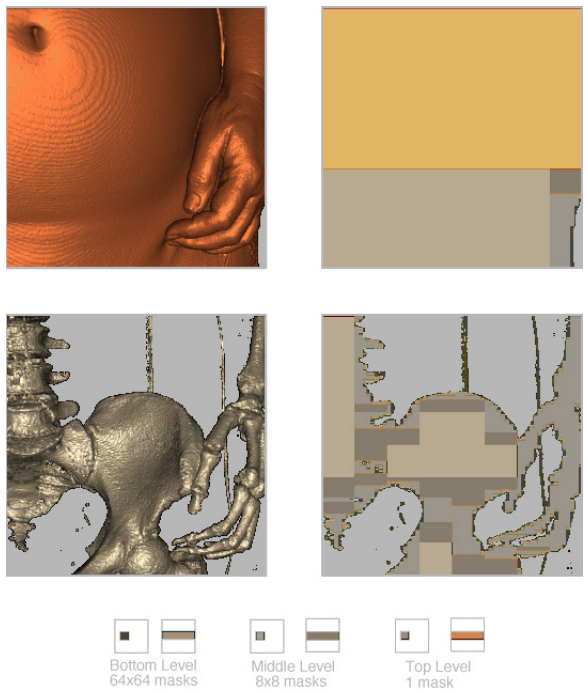


Figure 3: Isosurfaces and their corresponding visibility masks. The colors show where a single bit or a whole row (8 bits) of that level are covered.

5.2.2 Top-Down queries

To accelerate the visibility query, a hierarchical frame-buffer is used. In our implementation, each bit in one level represents a square of 8×8 bits in the next level (a single 64 bit word). Using this approach, a 512×512 frame-buffer can be represented by only 3 levels. It is important to note that, although the 8×8 representation makes it more difficult to check or update a specific bit, it reduces query time because each bit enables an acceleration in the x and y directions.

In general, the projection of a three-dimensional box onto the frame-buffer plane is not axis-aligned, which makes it harder to check all the pixels it covers. Rather, we check the visibility of the axis aligned 2D bounding box of this projection. Clearly, we overestimate the coverage of the node. However, based on the analysis in section 5.2 the overall investment is repaid.

A visibility query needs only to report whether the node is visible. It is not important to know, at this stage, which part is visible. As such, we can interrogate the hierarchical frame-buffer from top to bottom, and stop the search when a single *visible* bit is discovered. In order to accelerate the search, we first check all bits *completely inside* in the bounding box. If any of these bits are marked *visible*, the node is visible. If all these bits or marked *non-visible*, the bits on the boundary must be checked. For each boundary bit

marked *visible*, we must descend to the next level because we ignore which part of the 8×8 area in the next level is visible.

The visibility queries rely on continuous updates of the visibility frame-buffer, with newly extracted geometry. It is important to reduce the overhead associated with this update phase. Previous work on view-dependent extraction used a strategy similar to the query mechanism, *i.e.*, top-bottom traversal, to update the visibility frame-buffer. This approach is suitable for large triangles, but in the case where most extracted triangles are few pixels or less, the overhead of a top-bottom approach is high.

We propose to use a bottom-up approach where the extracted geometry is scan converted directly into the lowest level of the visibility frame-buffer at screen resolution. After scanning a row whose index ends in 7 (i.e., $mod(row, 8) = 7$), we examine all 8×8 tiles touched. If any tile is completely covered, we update the next level. Since we assume each scanned geometry covers only a few pixels, the level above rarely needs to be updated (only when all of the 64 bits are set), dramatically reducing cost.

Small meta cells Nodes in the GridTree with projection the size of a pixel or less are represented in our framework by a single point. We must ensure that the projected node covers only a single point. Considering Figure 4, we note that, while in cases A and B the pixel center is covered, this is not true for case C. The simplest solution is to require $\lfloor (left + 0.5) = \lfloor (right - 0.5)$, and similarly for the top and $bottom$ boundaries. Setting the corresponding bit in

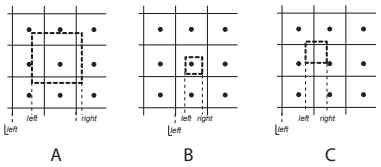


Figure 4: Small meta cells

the visibility frame-buffer is relatively cheap. We must, however, be careful when creating the 3D point that will be sent to the graphics hardware for rendering. We must ensure the point will project on the same pixel covered in the visibility frame-buffer. To this end, we back-project the pixel center into world coordinates using the inverse of the projection matrix.

Triangle scanning When the GridTree traversal reaches a single data cell a geometry needs to be generated to represent the isosurface in that cell. In our implementation, we selected the Marching Cubes triangulation with the exception of replacing small triangles with points, as discussed in section 4.1.
There are many approaches and methods for scan conversion of triangles. We use a barycentric coordinates approach because it requires the least amount of initial setup. Since we have to scan convert a large number of small triangles, which on average cover a few pixels, savings in the setup stage are crucial. Our algorithm computes the bounding box of the triangle in screen coordinates, then computes the (α, β, γ) barycentric coordinates for each pixel center within this bounding box. For each pixel whose center satisfies $\alpha \geq 0 \quad \beta \geq 0 \quad$ and $\quad \gamma \geq 0$ and is marked *visible* in the visibility frame-buffer, a 3D point is created by interpolating the three vertices of the triangle in 3D using the same (α, β, γ) coordinates.

5.3 Surface Normals

In the case of *near* and *mid* range views, see section 4.1, the triangle normal can be associated with the point(s) representing the triangle. However, there is no well-defined surface associated with the points generated for small nodes in the *far* view case.

In a ray casting approach such as RTRT [25], when a node containing the isosurface reduces to a pixel, the intersecting ray corresponds to a random sampling. This creates aliasing problems for the normals and jittered sampling must be applied to avoid a noisy image. While this anti-aliasing can be effective, it may require many samples per pixel to achieve good results. Hence the performance of the algorithm, which depends linearly on the number of rays used, drops significantly.

We propose to defer the surface normal computation to a post-processing step. Once the entire visible isosurface is extracted, the surface normals can be computed at the required points, based on their local properties. More specifically we derive the neighborhood connectivity of the points from their projection on the screen. To avoid computing normals across disconnected components of the isosurface we apply a threshold on the z-coordinates. In essence, we apply a specialized low pass filter on the extracted geometry. Computing surface normals using screen space connectivity can be done in software or using the graphics hardware. Pros and cons are discussed next.

5.3.1 Software

Flexibility in the design and implementation of the filter is the main advantage of performing the surface normal computation on the CPU. There are no practical limitations to the size of the filter, i.e., the extent of the processed neighborhood, or to the types of mathematical operations and the number of iterations it requires. On the flip side, the process is inherently slow. At most, two processors can be used on a typical high-end desktop. Furthermore, a correspondence needs to be maintained between the 3D points and their projection on the screen. If mixing points and triangles (for *close-up* views) are allowed in the same scene, we must compute in software the *z* coordinate for each visible point of the triangles.

5.3.2 Graphics hardware

Modern graphics hardware offers means for computing surface normals using vertex and fragment programs. The major advantage of the graphics hardware is the number of dedicated fragment and vertex units, which take advantage of the highly parallel nature of the type of filters required. The main disadvantage of todays graphics hardware is the limited programmability, especially of fragment units.

In order to use the graphics hardware, we need to provide it with the world and screen coordinates of each point. We also need to provide a method for accessing the computed normals during later rendering phases. This can be done using off screen p-buffers that we can both render and use as textures for a later lookup. Furthermore, modern graphics hardware supports *float* p-buffers where both world coordinates and computed normals can be stored accurately. We use the same projection (modelview, perspective and viewport) for the extraction, the following normal computation steps, and the final rendering. This guarantees that in each of the following steps, the fragment programs can easily find the required information for that fragment (pixel). This information includes the world coordinates of the point being projected onto this pixel along with the point's screen neighbors (*i.e.*, its neighbors with respect to the current view).

Position Map The first step is to store on the graphics hardware the world coordinates of the points and their location on the screen. To this end, we create a *position map* by projecting all geometry, points and triangles, onto a float p-buffer, saving the vertices world coordinates the output.

Normals Map Recall that for points derived from triangles, triangles normals can be used. If these normals are used, they are stored in the *normal map*, in addition to the computed normals.

First, we select a new p-buffer of floats, and project the geometry for which the normal is *known* along with its normals. In the fragment program, the incoming normals are saved as the output color, while setting the α channel to 1 (see below).
Next, we bind the *position map* from the previous step as texture, and project the geometry to compute the normals, *i.e.,* either there was *no* known normals as in the case of very far sub volumes, or we chose not to compute them from data. For each pixel, the fragment program computes the normal based on the world position of the point being projected and its screen neighbors (using texture lookup from the *position map*). The normals are then stored as output color. The fragment program can use the α value to differentiate between real neighbors and the background.
The *normal map* now contains a normal of each of the isosurface points, and normals for the pixels each triangle covers.

Rendering In the rendering phase, the *normal map* is used as a texture where each fragment can lookup its normal and perform correct shading. Care must be taken in the case of incomplete reconstruction, for example, when the user is allowed to change the viewpoint without recomputing the isosurface. In this case, the geometry will be projected onto a different location, and the fragment lookup will not work correctly.
We address this issue by saving the modelview and projection matrix used during the creation of the *normal map*. During the rendering phase, this matrix is provided to the vertex program, along with other shading parameters, such as the position of the light. The vertex program uses the given matrix, computes the fragment position at the time the *normal map* was created, and passes this index to the fragment shader.

5.4 Parallel Implementation

An increasing number of commodity computers feature a second processor thus allowing for parallel computing. Our view dependent extraction can be parallelized using sort first approach, i.e., based on screen projection. We construct two visibility framebuffers, each representing half the screen. To better balance the computation load, in each processor, a different thread traverses the dataset using its own visibility frame buffer. Since each frame buffer represents a different section of the screen, each thread effectively ignores large sections of the dataset. An additional optimization can be achieved by splitting, not the entire screen, but the projection of the bounding box of the dataset.

6 RESULTS

Setup Our setup includes a Dual Processor Power Mac 2GHz G5 with 2GB memory and ATI 9800 Pro graphics card. We chose the *Visible Woman* dataset for our benchmarks because it is a relatively large dataset (512x512x1734 shorts = 867MB) that exhibits both smooth and noisy complex isosurfaces. The later (mainly the skeleton) provides challenging test cases for view dependent isosurface extraction algorithms.

GridTree memory requirement The GridTree structure is determined by its depth, which can be set by the user. We allow the user to impose a particular branch factor for the lowest level. Table 1 shows depth versus memory requirement of the GridTree and an Octree (a Branch On Need version to conserve memory) for the *Visible Woman* dataset. The original *Visible Woman* dataset consists of seven sections, all of which have the same cross section of 512 by 512 pixels but different world space sizes. In addition, the number of slices in each section varies dramatically (209, 18, 22, 857, 4, 7, 617). PISA can work with these 7 sections by traversing them in a view dependent order. However, this is not a typical situation and thus we chose to resample the datasets and create a single dataset of the same size ($512 \times 512 \times 1734$) that is based on the spatial characteristics of the torso section. We note that the performance using the seven sections is only slightly slower than when using the single dataset. The RTRT is optimal when used with a very shallow hierarchy of only 3 level which allow for an extremely low memory requirement.

Depth	Base Branch	Size (MB)	Relative %
Octree	2	744	86
RTRT	12	3.5	0.4
PISA			
5	2	227	26
5	5	18	2
9	2	353	40

Table 1: Memory requirements of the Octree, RTRT and PISA relative to their depth and the branch factor of the base level. The dataset size is 867MB.

Performance We compare the performance of the proposed algorithm to the classic Octree [27] and to the Real Time Ray Tracer (RTRT) [25, 24], using either one or two of the G5 CPUs. In addition we reimplemented three routines using the *Altivec*, the G5 vector unit, which is similar to the Pentium SSE/SSE2. The three routines that benefit the most from using the Altivec units are the bounding box projection, triangulation using the marching cubes, and finally the scan-conversion routine that generates the final points.

Two isovalues were used, one for the skin and one for the skeleton. The skin isosurface exhibits large continuous sections which allow early termination in a view dependent extraction, Figure 5[c]. The skeleton isosurface constitutes a challenging case for any view dependent approach as it comprises a very large number of small sections (bones) and exhibits many see-through holes which prevent early termination, Figure 5[a]. The torso section is especially hard since it is very noisy which leads to very large number of triangles/points.

Table 2 and Table 3 present statistics for the two isosurfaces and from two viewpoints. The viewpoints where picked based on the portion of the dataset that was visible. Other viewpoints from the same areas exhibit the same characteristics. The torso as seen in Figure 5[a] is slower in all the cases but statistics were dropped out for lack of space. The tables show the performance using the classical Octree method (using a BonTree to conserve memory), the Real Time Ray Tracer (RTRT), and three different configurations of the GridTree. For each case we show the time in *msec* to perform a single extraction (Cull), the frame rate, and a relative speedup. The frame rate is shown for both a view-dependent cull + normal computation on the GPU + a single draw, and for the case when the current isosurface is redrawn from a new viewpoint, i.e. an incomplete reconstruction. The speedup is measured for each GridTree configuration when using one or two CPUs and when using the Altivec vector unit. In addition we compare the relative speed up of the 2 CPUs (+Altivecs) to the RTRT performance using 2 CPUs.

It is interesting to note that using the Altivec unit with a single CPU is almost equivalent to using a second CPU. In most cases this contribution is about 50%. Using both the Altivec and a second CPU leads to an improvement of about 250% across the board. It is clear that a full extraction of the isosurface (Octree) is not a feasible solution for large datasets. Comparing the results of the various GridTree, we note that the branch factor at the base level is a key for the acceleration but that this acceleration comes at a price of a higher memory footprint. The branch factors of the 5(5) tree is $3 \times 3 \times 3 \times 4 \times 5$ in X and Y, and $4 \times 4 \times 5 \times 5 \times 5$ in the Z direction. The 5(2) tree exhibits $4 \times 4 \times 4 \times 4 \times 2$ in X and Y, and $5 \times 5 \times$

Skin Isosurface						
Test	Type	Cull+Draw speedup RTRT	Cull+Draw speedup Tree	Cull+Draw f/sec	Draw only f/sec	Cull msec
Full extraction (not view dependent)						
Octree	15M tri	n/a	n/a	0.03	0.2	29,700
Far view of the whole body						
RTRT	2 CPUs	-	n/a	2.7	n/a	n/a
PISA					(41K points)	
5 (5)	1 CPU		-	0.6	277	1652
	+ Altivec		x1.2	0.7	277	1428
	2 CPUs		x1.7	1.0	277	943
	+ Altivec	x0.4	x2.0	1.2	277	812
5 (2)	1 CPU		-	3.6	277	269
	+ Altivec		x1.6	5.7	277	163
	2 CPUs		x1.6	5.8	277	159
	+ Altivec	x3.3	x2.5	9.0	277	98
9 (2)	1 CPU		-	3.8	277	253
	+ Altivec		x1.5	5.9	277	160
	2 CPUs		x1.6	6.3	277	146
	+ Altivec	x3.6	x2.5	9.6	277	94
Closeup view of the feet						
RTRT	2 CPUs	-	n/a	0.7	n/a	n/a
PISA					(170K points)	
5 (5)	1 CPU		-	3.3	73	271
	+ Altivec		x1.1	3.8	73	234
	2 CPUs		x1.5	4.9	73	174
	+ Altivec	x7.4	x1.6	5.2	73	152
5 (2)	1 CPU		-	3.9	73	226
	+ Altivec		x1.5	5.7	73	146
	2 CPUs		x1.5	5.8	73	141
	+ Altivec	x11.6	x2.1	8.3	73	91
9 (2)	1 CPU		-	4.3	73	211
	+ Altivec		x1.5	6.4	73	138
	2 CPUs		x1.5	6.4	73	135
	+ Altivec	x11.9	x2.3	9.8	73	86

Table 2: Performance statistics for the isosurface of the skin. Test X (Y) refers to a GridTree of depth X with a base branch factor of Y. Cull+Draw refers to extracting a new isosurface + computing the normals on the GPU + a draw phase.

Skeleton Isosurface						
Test	Type	Cull+Draw speedup RTRT	Cull+Draw speedup Tree	Cull+Draw (f/sec)	Draw only f/sec	Cull (msec)
Full extraction (not view dependent)						
Octree	11M tri	n/a	n/a	0.04	0.3	20,600
Far view of the whole body						
RTRT	2 CPUs	-	n/a	2.3	n/a	n/a
PISA					(25K points)	
5 (5)	1 CPU	-		0.7	398	1469
	+ Altivec		x1.2	0.8	398	1217
	2 CPUs		x1.4	1.0	398	925
	+ Altivec	x0.5	x1.7	1.2	398	772
5 (2)	1 CPU	-		2,8	398	352
	+ Altivec		x1.6	4.6	398	211
	2 CPUs		x1.6	4.6	398	205
	+ Altivec	x3.3	x2.7	7.5	398	123
9 (2)	1 CPU	-		3.5	398	277
	+ Altivec		x1.5	5.4	398	177
	2 CPUs		x1.7	6.0	398	158
	+ Altivec	x3.9	x2.6	9.0	398	101
Closeup view of the feet						
RTRT	2 CPUs	-	n/a	1.1	n/a	n/a
PISA					(99K points)	
5 (5)	1 CPU		-	4.4	130	208
	+ Altivec		x1.2	5.4	130	170
	2 CPUs		x1.7	7.5	130	114
	+ Altivec	x8.1	x2.1	8.9	130	93
5 (2)	1 CPU		-	5.8	130	155
	+ Altivec		x1.5	8.4	130	101
	2 CPUs		x1.7	9.6	130	84
	+ Altivec	x12	x2.3	13.2	130	56
9 (2)	1 CPU		-	6.0	130	150
	+ Altivec		x1.4	8.7	130	97
	2 CPUs		x1.6	9.9	130	80
	+ Altivec	x13.5	x2.5	14.9	130	53

Table 3: Performance statistics for the isosurface of the skeleton. Test X (Y) refers to a GridTree of depth X with a base branch factor of Y. Cull+Draw refers to extracting a new isosurface + computing the normals on the GPU + a draw phase.

$6 \times 6 \times 2$ in Z. The 9(2) has almost a constant branch factor of 2, i.e., almost an Octree though much more compact. While the 9(2) does outperform all the others, it seems that the 5(2) is the optimal configuration. We may conclude that the optimal GridTree is one where the base factor is 2. More tests are needed to find if there exists *a priori* an optimal tree depth based on the size of the data.

The most surprising point is the relative speedup of PISA vs. the Real Time Ray Tracer. We expected PISA to perform well as it takes advantage of coherence in image space, value space, geometric space, vector units and the GPU, yet PISA on a single G5 is able to perform on the same level as RTRT on a 8-32 CPUs machine.

Finally, the notion of incomplete reconstruction as seen in Figure 1 can allow manipulation of the data at 70-400 fps. We intend to explore the idea of partial extraction during a dynamic manipulation of the viewpoint by the user, i.e. allow the extraction to proceed for a limited amount of time and add the results to the current isosurface. Such an approach will require special modification such that the current extracted isosurface can be used to cull away large sections of the data and accelerate the search for the new sections of the isosurface that becomes visible.

7 Conclusions

We have presented a new point-based approach to view-dependent isosurface extraction, where points are selected based on both the visibility and the projected area of sections of the isosurface. We proposed a software-based view dependent implementation along with a fast visibility query system. A post-processing surface normal estimation framework was also introduced to provide points with consistent and robust normals in configurations where a large region of the isosurface projects onto a single pixel. We showed that the combination of these techniques yields a method that, in contrast to previous work, allows for isosurface extraction of large data sets at interactive frame rates on a standard desktop computer.

For future work, we are looking at extending the algorithm to allow out of core extraction of larger datasets that cannot fit into main memory.

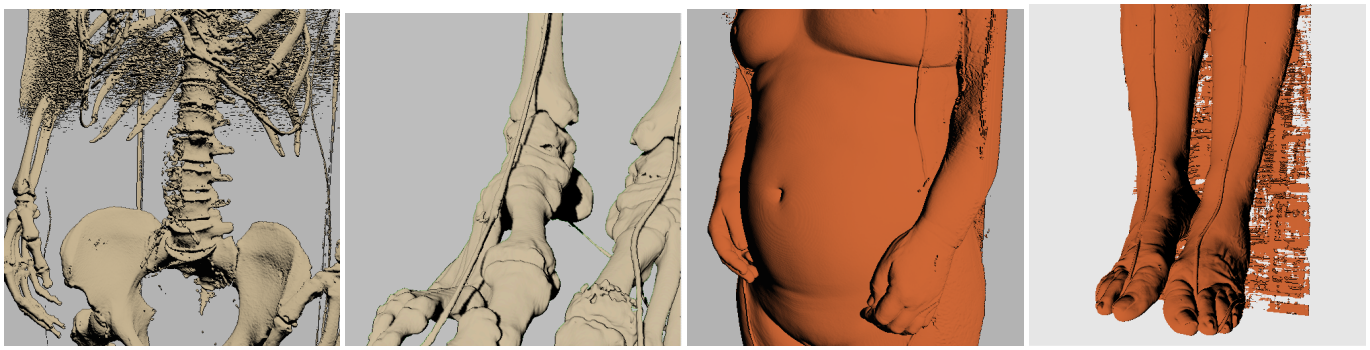
Acknowledgments

This work was supported in part by grants from the DOE ASCI, the DOE AVTC, the NIH NCRR, and by the National Science Foundation. We would like to thank Rachel McNeil and Charles Hansen for their contribution. We also thank Chris Co and the anonymous reviewers for their comments that helped improve the paper.

References

[1] J. A. Baerentzen and N. J. Christensen. Hardware accelerated point rendering of isosurfaces. *Journal of WSCG*, 11(1):41–48, 2003.

[2] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings*



(a) (b) (c) (d)

Figure 5: Several views of the visible woman dataset.

of the 1999 IEEE symposium on Parallel visualization and graphics, pages 97–104. ACM Press, 1999.

[3] Yi-Jen Chiang and Cláudio T. Silva. I/O optimal isosurface extraction. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 293–300, 1997.

[4] Yi-Jen Chiang, Cludio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of the conference on Visualization '98*, pages 167–174. IEEE Computer Society Press, 1998.

[5] 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*. ACM Press, 1996.

[6] H.E. Cline, Lorensen W.E., and Ludke S. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, 1988.

[7] C. S. Co, B. Hamann, and K. I. Joy. Iso-splatting: A point-based alternative to isosurface visualization. In *11th Pacific Conference on Computer Graphics and Applications (PG'03)*, pages 325–334, 2003.

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

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

[10] Jinzhu Gao and Han-Wei Shen. Hardware-assisted view-dependent isosurface extraction using spherical partition. In *Proceedings of the symposium on Data visualisation 2003*, pages 267–276. Eurographics Association, 2003.

[11] M. Giles and R. Haimes. Advanced interactive visualization for CFD. *Computing Systems in Engineering*, 1(1):51–62, 1990.

[12] Ned Greene. Hierarchical polygon tiling with coverage masks. In *Computer Graphics*, Annual Conference Series, pages 65–74, 1996.

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

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

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

[16] G. Ji, H.-W. Shen, , and J. Gao. Interactive exploration of remote isosurfaces with point-based non-photorealistic rendering. In *Joint Eurographics - TCVG Symposium on Visualization*, 2003.

[17] Philippe G. Lacroute. Fast volume rendering using shear-warp factorization of the viewing transformation. Technical report, Stanford University, September 1995.

[18] M. Levoy and T. Whitted. The use of points as display primitives. Technical report, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

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

[20] Y. Livnat, X. Cavin., and C. Hansen. Phase: Progressive hardware assisted isosurface extraction framework. Technical Report USCI-2002-001, SCI Institute, 2002.

[21] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *Visualization '98*, pages 175–180. ACM Press, October 1998.

[22] Y Livnat, H. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Trans. Vis. Comp. Graphics*, 2(1):73–84, 1996.

[23] W.E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[24] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 1999.

[25] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *Visualization 98*, pages 233–238. IEEE Computer Society Press, October 1998.

[26] H. Shen and C. R. Johnson. Sweeping simplicies: A fast iso-surface extraction algorithm for unstructured grids. *Proceedings of Visualization '95*, pages 143–150, 1995.

[27] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *Computer Graphics*, 24(5):57–62, November 1990.

[28] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[29] Xiaoyu Zhang, Chandrajit Bajaj, and Vijaya Ramachandran. Parallel and out-of-core view-dependent isocontour visualization using random data distribution. In D. Ebert, P. Brunet, and I. Navazo, editors, *Joint Eurographics — IEEE TVCG Symposium on Visualization (VisSym-02)*, pages 1–10, 2002.

[30] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In Eugene Fiume, editor, *Proceedings of ACM SIGGRAPH 2001*, pages 371–378. ACM Press/ ACM SIGGRAPH, 2001.