

2 Accelerated Isosurface Extraction Approaches

YARDEN LIVNAT*

2.1 Introduction

The *marching cubes* [7,15] method demonstrated that isosurface extraction can be reduced, using a divide-and-conquer approach, to solving a local triangulation problem. In addition, the marching cubes method proposed a simple and efficient local triangulation using a lookup table. However, the marching cubes did not address the *divide* portion of the approach, i.e., how to efficiently search a large dataset for these small local triangulations. To this end, the marching cubes method checks each and every cell of the dataset.

In this chapter, we introduce the three main approaches to accelerate isosurface extraction, (Section 2.2) and present two specific methods. Section 2.3 introduces the *Span Space* metaphor and uses it to devise a near-optimal search method in Section 2.4. Finally, Section 2.5 examines the view-dependent extraction approach and presents a particular implementation.

2.2 Isosurface Extraction Approaches

The various approaches to the acceleration of isosurface extraction fall into three main categories. Each approach is characterized based on the space in which it operates, namely: geometric, value, or image space decomposition. While some methods can be applied to structured and unstructured datasets, others lend themselves to only one, usually structured, grid.

2.2.1 Geometric Space Decomposition

Originally, only structured grids were available as an underlying geometry. Structured grids impose an order on the given cell set. By utilizing this order, methods based on the geometry of the data set could take advantage of the coherence between adjacent cells.

2.2.1.1 Marching Cubes

Perhaps the most well known isosurface extraction method to achieve high resolution results is the *marching cubes* method introduced by Lorensen and Cline [7]. The marching cubes method concentrated on the approximation of the isosurface inside the cells rather than on efficient location of the involved cells. Toward this end, the marching cube method scans the *entire* cell set, one cell at a time. The novelty of the method is the way in which it decides for each cell *whether* the isosurface intersects that cell and, if so, *how* to approximate it.

2.2.1.2 Octrees

The marching cubes method did not attempt to optimize the time needed to search for the cells that actually intersect the isosurface. This issue was later addressed by Wilhelms and Van Gelder [14], who employed an octree, effectively creating a 3D hierarchical decomposition of the cell set. Each node in the tree was tagged with the minimum and maximum values of the cells it represents. These tags, and the hierarchical

*Text and images taken with permission from the book “The Visualization Toolkit An Object-Oriented Approach to 3D Graphics” published by Kitware, Inc. <http://www.kitware.com/products/vtktextbook.html>.

nature of the octree, enable one to trim off sections of the tree during the search and thus to restrict the search to only a portion of the original geometric space. Wilhelms and Van Gelder did not analyze the time complexity of the search phase of their algorithm. However, octree decompositions are known to be sensitive to the underlying data. If the underlying data contain some fluctuations or noise, most of the octree will have to be traversed. Livnat et al. [6] present an analysis of the octree algorithm and show that the algorithm has a worst-case complexity of $O(k + k \log n/k)$, where n is the number of cells in the dataset and k is the size of the extracted isosurface [6]. Finally, octrees have been applied primarily to structured grids, and they are not easily adapted to handle unstructured grids.

2.2.2 Value Space Decomposition

Decomposing the value space, rather than the geometric space, has two advantages. First, the underlying geometric structure is of no importance, so this decomposition works well with unstructured grids. Second, for a scalar field in 3D, the dimensionality of the search is reduced from three to two.

In general values space decomposition methods exhibits worst-case complexity of $O(n)$. In Section 2.3 we introduce the span space metaphor and present, in Section 2.4 a near optimal isosurface extraction method based on the span space with a worst-case complexity of $O(k + \sqrt{n})$.

2.2.3 Image Space Decomposition

Today's large datasets pose new challenges. Datasets of several gigabytes can be found in many fields e.g., medicine, flow simulation and geo-sciences. The size of isosurfaces extracted from these datasets can reach several million polygons, many of which are less than one pixel in size. Two problems emerge due to the large number of polygons. First, due to the sheer number of cells containing an isosurface, the

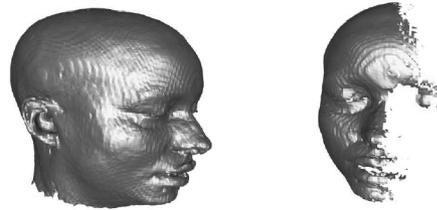


Figure 2.1 Left: The user view. Right: The same isosurface from a 90-degree angle to the user view, illustrating the incomplete reconstruction.

computation of all the local triangulation can be very time-consuming, even if fast acceleration methods are used. Second, the huge number of polygons, in the extracted isosurface, can easily overwhelm even the most powerful graphics accelerators, leading to poor interaction. In other words, not only the size, n , of the datasets is very large, but also the size k of the extracted isosurface becomes a problem.

One current approach to the large number of polygons problem is mesh reduction techniques [13,8]. The mesh reduction is applied to isosurface either as a post-process after the extraction phase or during the extracting phase itself [11]. However, mesh reduction is expensive and requires extracting the entire isosurface for examination. Furthermore, a change in the isovalue requires the *full* extraction of a new isosurface and the reapplication of the mesh reduction step.

Another approach is to employ ray-tracing techniques, which do not create an intermediate polygonal representation. Ray-tracing, nevertheless, does not take advantage of graphics hardware and requires a large number of CPUs to achieve interactivity [9].

View-dependent isosurface extraction [5], on the other hand, aims to reduce the search, construction, and rendering times, all at once. The key to this approach is accessing only cells that contain the *visible* portion of the isosurface from a given view point, i.e., based on the image space. The approach is based on a hierarchical front-to-back traversal of the dataset while skipping the non-visible

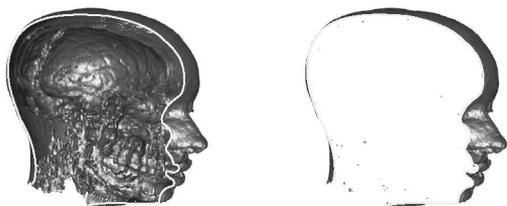


Figure 2.2 Extracted isosurface. A cut plane through the full and view-dependent isosurfaces extracted from the same view point as in Fig. 2.1. Note the large internal structures that are part of the full isosurface but not part of the view-dependent isosurface.

sections of the dataset from the current view point. Fig. 2.2 shows the potential saving of such an approach. Note the large section of the isosurface, which represents the internal organs in the head, yet is not part of the view-dependent isosurface.

2.3 The Span Space

Let $\varphi: G \rightarrow V$ be a given field and let \mathcal{D} be a sample set over φ , such that,

$$\mathcal{D} = \{d_i\} \quad d_i \in D = G \times V$$

where $G \subseteq \mathbb{R}^p$ is a geometric space and $V \subseteq \mathbb{R}$, for some $p \in \mathbb{Z}$, is the associated value space. Also, let $d = |\mathcal{D}|$ be the size of the data set.

Definition 2–1 (Isosurface Extraction) *Given a set of samples \mathcal{D} over a field $\varphi: G \rightarrow V$, and given a single value $v \in V$, find,*

$$S = \{g_i\} \quad g_i \in G \quad \text{such that,} \quad (1) \\ \varphi(g_i) = v.$$

Note that S , the isosurface, need not be topologically simple.

Approximating an isosurface, S , as a global solution to Equation 2.1 can be a difficult task because of the sheer size, d , of a typical science or engineering data set.

Data are often generated from 3D images or as solutions to numerical approximation techniques, such as from finite difference or finite element methods. These methods naturally de-

compose the geometric space, G , into a set of polyhedral cells, C , where the data points define the vertices. While $n = |C|$, the number of cells, is typically an order of magnitude larger than d , the approximation of the isosurface over C becomes a manageable task. Rather than finding a global solution, one can seek a local approximation within each cell. Hence, isosurface extraction becomes a two-stage process: Locating the cells that intersect the isosurface and then, locally, approximating the isosurface inside each such cell. We focus our attention on the problem of finding those cells that intersect an isosurface of a specified iso-value.

On structured grids, the position of a cell can be represented in the geometric space G . Because this representation does not require explicit adjacency information between cells, isosurface extraction methods on structured grids conduct searches over the geometric space, G . The problem as stated by these methods is defined as follows:

Approach 2–1 (Geometric Search) *Given a point $v \in V$ and given a set C of cells in G space where each cell is associated with a set of values $\{u_j\} \in V$ space, find the subset of C which an isosurface, of value v , intersects.*

Efficient isosurface extraction for unstructured grids is more difficult, as no explicit order, i.e., position and shape, is imposed on the cells, only an implicit one that is difficult to utilize. Methods designed to work in this domain have to use additional explicit information or revert to a search over the value space, V . The advantage of the latter approach is that one needs only to examine the minimum and maximum values of a cell to determine whether an isosurface intersects that cell. Hence, the dimensionality of the problem reduces to two for scalar fields.

Many methods for isosurface extraction over unstructured grids, as well as some for structured grids, view the isosurface extraction problem in the following way:

Approach 2–2 (Interval Search) Given a point $v \in V$ and given a set of cells represented as intervals,

$$I = \{[a_i, b_i]\} \text{ such that, } a_i, b_i \in V$$

find the subset I_s such that,

$$I_s \subseteq I \text{ and, } a_i \leq v \leq b_i \quad \forall (a_i, b_i) \in I_s,$$

where a norm should be used when the dimensionality of V is greater than one.

Posing the search problem over intervals introduces some difficulties. If the intervals are of the same length or are mutually exclusive, they can be organized in an efficient way suitable for quick queries. However, it is much less obvious how to organize an arbitrary set of intervals. Indeed, what distinguishes these methods from one another is the way they *organize* the intervals rather than the way they perform searches.

Our approach is not to view the problem as a search over intervals in V but rather as a search over points in V^2 . We start with an augmented definition of the search space.

Definition 2–2 (The Span Space) Let C be a given set of cells, define a set of points $P = \{p_i\}$ over V^2 such that,

$$\forall c_i \in C \text{ associate, } p_i = (a_i, b_i)$$

where,

$$a_i = \min_j \{v_j\}_i \text{ and } b_i = \max_j \{v_j\}_i$$

and $\{v_j\}_i$ are the values of the vertices of cell i .

Though conceptually not much different from the interval space, the span space, nevertheless, leads to a simple and near optimal search algorithm. In addition, the span space enables us to clarify the differences and commonalities between previous interval approaches as shown in [6].

One key concept is that points in 2D exhibit no explicit relations between themselves, while intervals tend to be viewed as stacked on top of each other, so that overlapping intervals exhibit merely coincidental links. Points do not exhibit

such arbitrary ties and in this respect lend themselves to many different organizations. However, as we shall show later, previous methods grouped these points in very similar ways because they looked at them from an interval perspective.

Using our augmented definition, the isosurface extraction problem can be stated as

Approach 2–3 (The Span Search) Given a set of cells, C , and its associated set of points, P , in the span space, and given a value $v \in V$, find the subset $P_s \subseteq P$, such that

$$\forall (x_i, y_i) \in P_s \quad x_i < v \quad y_i > v.$$

We note that $\forall (x_i, y_i) \in P_s, x_i < y_i$ and thus the associated points will lie above the line $y_i = x_i$. A geometric perspective of the span search is given in Fig. 2.3.

2.4 Near Optimal Isosurface Extraction (NOISE)

In the following, we present an acceleration method that is based on the span space decomposition. Using the span space as our underlying domain, we employ a kd-tree as a means for simultaneously ordering the cells according to their maximum and minimum values.

2.4.1 Kd-Trees

Kd-trees were designed by Bentley [1] in 1975 as a data structure for efficient associative searching. In essence, kd-trees are a multi-dimensional version of binary search trees. Each node in the tree holds one of the data values and has two sub-trees as children. The sub-trees are constructed so that all of the nodes in one sub-tree, the *left* one for example, hold values that are less than the parent node's value, while the values in the *right* sub-tree are greater than the parent node's value.

Binary trees partition data according to only one dimension. Kd-trees, on the other hand, utilize multidimensional data and partition the data by alternating between each of the dimensions of the data at each level of the tree.

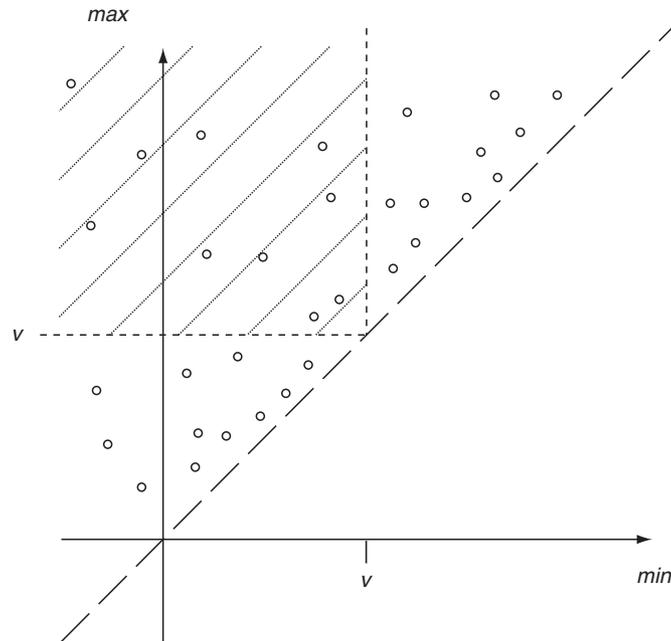


Figure 2.3 Search over the span space.

2.4.2 Search over the Span Space Using Kd-Tree

Given a data set, a kd-tree that contains pointers to the data cells is constructed. Using this kd-tree as an index to the data set, the algorithm can now rapidly answer isosurface queries. Fig. 2.4 depicts a typical decomposition of a span space by a kd-tree.

2.4.2.1 Construction

The construction of the kd-trees can be carried out recursively in optimal time $O(n \log n)$. The approach is to find the median of the data values along one dimension and store it at the root node. The data are then partitioned according to the median and recursively stored in the two sub-trees. The partition at each level alternates between the min and max coordinates.

An efficient way to achieve $O(n \log n)$ time is to recursively find the median in $O(n)$, using the

method described by Blum et al. [3], and to partition the data within the same time bound.

A simpler approach is to sort the data into two lists according to the maximum and minimum coordinates, respectively, in order $O(n \log n)$. The first partition accesses the median of the first list, the *min* coordinate, in constant time, and marks all of the data points with values less than the median. We then use these marks to construct the two subgroups, in $O(n)$, and continue recursively.

Although the above methods have complexity of $O(n \log n)$, they do have weaknesses. Finding the median in optimal time of $O(n)$ is theoretically possible yet difficult to program. The second algorithm requires sorting two lists and maintaining a total of four lists of pointers. Although it is still linear with respect to its memory requirement, it nevertheless poses a problem for very large data sets.

A simple and elegant solution is to use a Quicksort-based selection [12]. While this

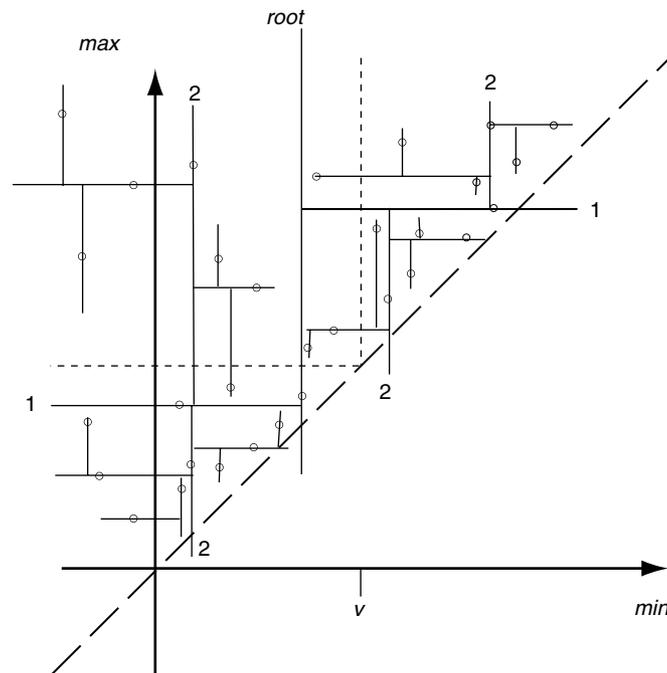


Figure 2.4 Kd Tree.

method has a *worst case* of $O(n^2)$, the *average case* is only $O(n)$. Furthermore, this selection algorithm requires no additional memory and operates directly on the tree.

It is clear that the kd-tree has one node per cell, or span point, and thus that the memory requirement of the kd-tree is $O(n)$.

2.4.2.2 Query

Given an iso-value, v , we seek to locate all of the points in Fig. 2.3 that are to the *left* of the vertical line at v and are *above* the horizontal line at v . We note that we do not need to locate points that are *on* these horizontal or vertical lines.

The kd-tree is traversed recursively when the iso-value is compared to the value stored at the current node alternating between the minimum and maximum values at each level. If the node is to the left (above) of the iso-value line, then only the left (right) sub-tree should be traversed. Otherwise, *both* sub-trees should be

traversed recursively. For efficiency, we define two search routines, *search-min-max* and *search-max-min*. The dimension we currently check is the first named, and the dimension that we still need to search is named second. The importance of naming the second dimension will be evident in the next section, when we consider optimizing the algorithm.

Following is a short pseudo-code for the min-max routine.

```
search-min-max( iso-value, node )
{
  if ( node.min < iso-value ) {
    if ( node.max > iso-value )
      construct a polygon(s) from node
      search-max-min ( iso-value,
        node.right );
  }
  search-max-min ( iso-value,
    node.left );
}
```

Q1

Estimating the complexity of the query is not straightforward. Indeed, the analysis of the worst case was developed by Lee and Wong [4] only several years after Bentley introduced kd-trees. Clearly, the query time is proportional to the number of nodes visited. Lee and Wong analyzed the worst case by constructing a situation where all the visited nodes are not part of the final result. Their analysis showed that the worst-case time complexity is $O(\sqrt{n} + k)$. The average case analysis of a region query is still an open problem, though observations suggest that it is much faster than $O(\sqrt{n} + k)$ [12, 2]. In almost all typical applications $\hat{k} \sim n^{2/3} > \sqrt{n}$, which suggests a complexity of only $O(k)$. On the other hand, the complexity of the isosurface extraction problem is $\Omega(k)$, because it is bound from below by the size of the output. Hence, the proposed algorithm, NOISE, is optimal, $\theta(k)$, for almost all cases and is near optimal in the general case.

2.4.3 Optimization

The algorithm presented in the previous section is not optimal with regard to both the

memory requirement and search time. We now present several strategies to optimize the algorithm.

2.4.3.1 Pointerless Kd-Tree

A kd-tree node, as presented previously, must maintain links to its two sub-trees. This introduces a high cost in terms of memory requirements. To overcome this, we note that, in our case, the kd-tree is completely balanced. At each level, one data point is stored at the node, and the rest are equally divided between the two sub-trees. We can, therefore, represent a pointerless kd-tree as a one-dimensional array of the nodes. The root node is placed at the middle of the array, while the first $n/2$ nodes represent the left sub-tree and the last $(n - 1)/2$ nodes the right sub-tree, as shown in Fig. 2.5.

When we use a pointerless kd-tree, the memory requirements for our kd-tree, per node, reduce to two real numbers, for minimum and maximum values, and one pointer back to the original cell for later usage. Considering that each cell, for a 3D application with tetrahedral cells, has pointers to four vertices, the kd-tree

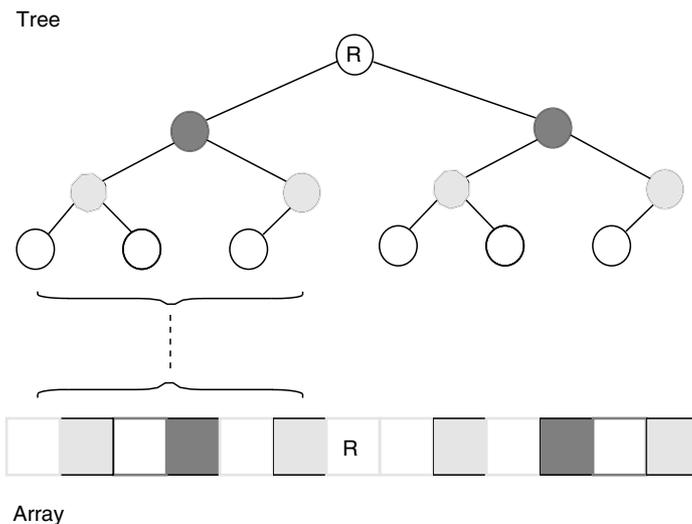


Figure 2.5 Pointerless Kd-tree

memory overhead is even smaller than the size of the set of cells.

The use of a pointerless kd-tree enables one to compute the tree as an offline preprocess and load the tree using a single read in time complexity of only $O(n)$. Data acquisition via CT/MRI scans or scientific simulations is generally very time consuming. The ability to build the kd-tree as a separate preprocess allows one to shift the cost of computing the tree to the data-acquisition stage. Hence, the impact of the initialization stage on the extraction of isosurfaces for large data sets is reduced.

2.4.3.2 Optimized Search

The search algorithm can be further enhanced. Let us consider, again, the min-max (max-min) routine. In the original algorithm, if the iso-value is less than the minimum value of the node, then we know we can trim the right sub-tree. Consider the case where the iso-value is greater than the node's minimum coordinate. In this case, we need to traverse *both* sub-trees. We have no new information with respect to the search in the right sub-tree, but for the search in the left sub-tree we *know* that the minimum condition is satisfied. We can take advantage of this fact by skipping over the odd levels from that point onward. To achieve this, we define two new routines: *search-min* and *search-max*. Adhering to our previous notation, the name *search-min* indicates that we are only looking for a minimum value.

Examining the search-min routine, we note that the maximum requirement is already satisfied. We do not gain new information if the iso-value is less than the current node's minimum and again only trim off the right sub-tree. If the iso-value is greater than the node's minimum, we recursively traverse the right sub-tree, but with regard to the left sub-tree we now know that all of its points are in the query's domain. We therefore need only to *collect* them. Using the notion of pointerless kd-tree as proposed in Section 2.4.3.1, any sub-tree is represented as a *contiguous* block of the tree's nodes. Collecting

all of the nodes of a sub-tree requires only sequentially traversing this contiguous block.

A pseudo-code of the optimized search for the odd levels of the tree, i.e., searching for minima, is presented in Fig. 2.6. The code for even levels, searching for maxima, is essentially the same and uses the same collect routine.

2.4.3.3 Count Mode

Extracting isosurfaces is an important goal, yet in a particular application one may wish only to know how many cells intersect a particular isosurface. Knowing the number of cells that intersect the isosurface can help one to make a rough estimate of the surface area of the isosurface on a structured grid and on a "well-behaved" unstructured grid. The volume encompassed by the isosurface can also be estimated if one knows the number of cells that lie inside the isosurface as well as the number of cells that intersect it.

The above algorithm can accommodate the need for such particular knowledge in a simple way. The number of cells intersecting the isosurface can be found by incrementing a counter rather than constructing polygons from a node and by replacing collection with a single increment of the counter with the size of the sub-tree, which is known without the need to traverse the tree. To count the number of cells that lie inside the isosurface, one need only look for the cells that have a maximum value below the iso-value.

The worst-case complexity of the count mode is only $O(\sqrt{n})$. A complete analysis is presented in Livnat et al. [6]. It is important to note that the count mode does not depend on the size of the isosurface. The count mode thus enables an application to quickly count the cells that intersect the isosurface and to allocate and prepare the appropriate resources *before* a full search begins.

2.4.4 Triangulation

Once a cell is identified as intersecting the isosurface, we need to approximate the isosurface

```

search_min_max( iso_value, node )
{
  if ( node.min < iso_value ) {
    if ( node.max > iso_value )
      construct polygon(s) from node;
    search_max_min( iso_value, node.right );
    search_max( iso_value, node.left );
  } else
    search_max_min( iso_value, node.left );
}
search_min( iso_value, node )
{
  if ( node.min < iso_value ) {
    construct polygon(s) from node;
    search_skip_min( iso_value, node.right );
    collect( node.left );
  } else
    search_skip_min( iso_value, node.left );
}
search_skip_min( iso_value, skip_node )
{
  if ( skip_node.min < iso_value )
    construct polygon(s) from skip_node;
  search_min( iso_value, skip_node.right );
  search_min( iso_value, skip_node.left );
}
collect( sub_tree )
{
  sequentially construct polygons for all nodes
  in this sub_tree
}

```

Figure 2.6 Optimized Search

inside that cell. Toward this goal, the marching cubes algorithm checks each of the cell's vertices and marks them as either *above* or *below* the isosurface. Using this information and a lookup table, the algorithm identifies the particular way the isosurface intersects the cell. The marching cubes method and its many variants are designed for structured grids, although they can be applied to unstructured grids as well.

In Livnat et al.[6], we proposed an algorithm for unstructured grids of tetrahedral cells. We first note that if an isosurface intersects *inside* a

cell, then the vertex with the maximum value *must* be above the isosurface, and the vertex with the minimum value *must* be below it.

To take advantage of this fact, we reorder the vertices of a cell according to their ascending values, say v_1 to v_4 , *a priori*, in the initialization stage. When the cell is determined to intersect the isosurface, we need only to compare the iso-value against, *at most*, the two middle vertices. There are only three possible cases: only v_1 is *below* the isosurface, only v_4 is *above* the isosurface, or $\{v_1, v_2\}$ are below and $\{v_3, v_4\}$ are

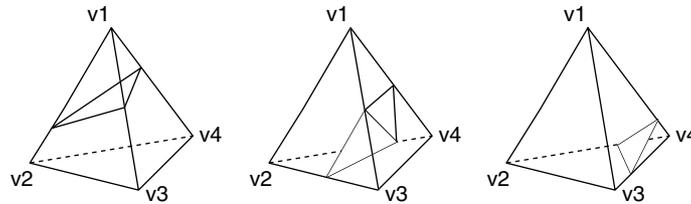


Figure 2.7 Triangulation. The vertices are numbered according to ascending values.

above (Fig. 2.7). Moreover, the order of the vertices of the approximating triangle(s), such that the triangle(s) will be oriented correctly with respect to the isosurface, is known in advance at no cost. We can take further advantage of the fact that there are only four possible triangles for each cell and compute their normals *a priori*. This option can improve the triangulation time dramatically, yet it comes with a high memory price tag.

2.5 View-Dependent Isosurface Extraction

The view-dependent extraction approach is based on the observation that isosurfaces extracted from very large datasets tend to exhibit high depth complexity for two reasons. First, since the datasets are very large, the projection

of individual cells tends to be subpixel. This leads to a large number of polygons, possibly nonoverlapping, projecting onto individual pixels. Second, for some datasets, large sections of an isosurface are internal and thus are occluded by other sections of the isosurface, as illustrated in Fig. 2.2. These internal sections, common in medical datasets, 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. Fig. 2.8 depicts a two-dimensional scenario. In a view-dependent method only the solid lines are extracted, whereas in non-view-dependent isocontouring solid and dotted are extracted.

Our view-dependent approach is based on a hierarchical traversal of the data and a marching cubes triangulation. We exploit coherency in

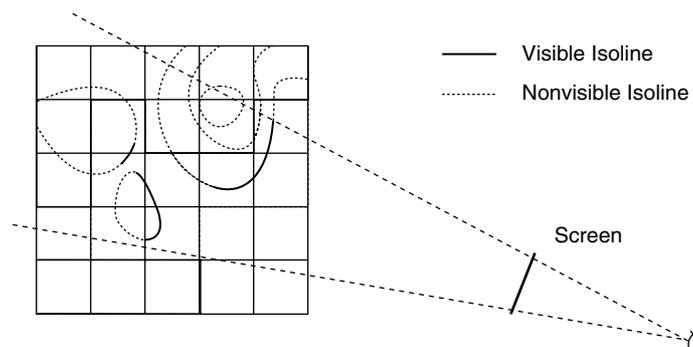


Figure 2.8 A two-dimensional scenario.

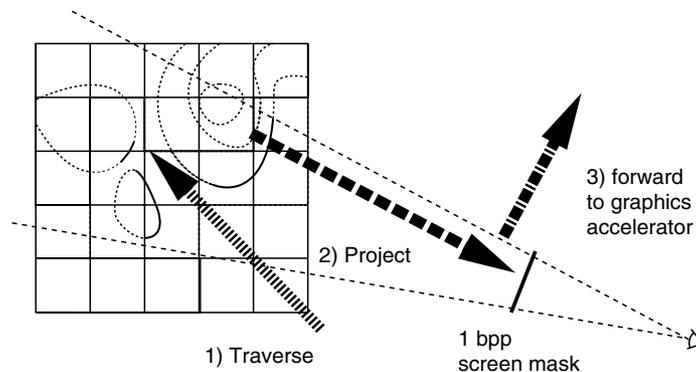


Figure 2.9 The three-step algorithm.

the object, value, and image spaces, as well as balancing the work between the hardware and the software. The three-step approach is depicted in Fig. 2.9.

First, Wilhelms' and Van Gelder's [14] algorithm is augmented by traversing down a hierarchical tree in a front-to-back order in addition to pruning empty sub-trees based on the min-max values stored at the tree nodes. The second step employs coarse software visibility tests for each [meta-] cell which intersect the isosurface. The aim of these tests is to determine whether the [meta-] cell is hidden from the view point by previously extracted sections of the isosurface (thus the requirement for a front-to-back traversal). Finally, the triangulations of the visible cells are forwarded to the graphics accelerator for rendering by the hardware. It is at this stage that the final and exact [partial-] visibility of the triangles is resolved. A dataflow diagram is depicted in Fig. 2.10.

2.5.1 The Min/Max Tree

Wilhelms and Van Gelder [14] used an octree for their hierarchical representation of the underlying dataset. Each node of the octree contained the minimum and maximum values of its subtree. In order to reduce the memory footprint, the octree leaves were one level higher than the data cells. In other words, each leaf node represented the min/max values of $8 (2 \times 2 \times 2)$ data

cells. Wilhelms and Van Gelder also introduced a new octree variant (BON tree) for handling datasets with sizes that are not a power of two.

The BON tree was adequate for relatively small datasets of total size less than 2^{28} or 256 MB. In addition, the use of 32 bits pointers for each node proves to be expensive when the data are 8 bits per node. In this case, the min/max values in each node require only 2 bytes but the pointer to the next node requires 4 bytes. The alignment of this pointer further increases the size of each node by 2 bytes, resulting in 8 bytes per node instead of only 2 bytes. As a result, the BON tree can consume as much memory as the dataset itself, and sometimes even more. For large datasets, this is too high a price to pay.

In the case of view-dependent extraction, where each node in the hierarchical tree has to be culled against the virtual framebuffer, each level in the hierarchy increases the cost of the traversal. The trade-off is that deep hierarchy provides better culling, *i.e.*, pruning of non-visible or empty sections but with an increase cost both in terms of memory and processing time of each node.

To address these issues, we have implemented a shallow hierarchy [9]. Each level of the hierarchy can have a different number of nodes. The depth of the hierarchy can thus be adapted on a per-dataset case for optimum memory-vs.-time configuration.

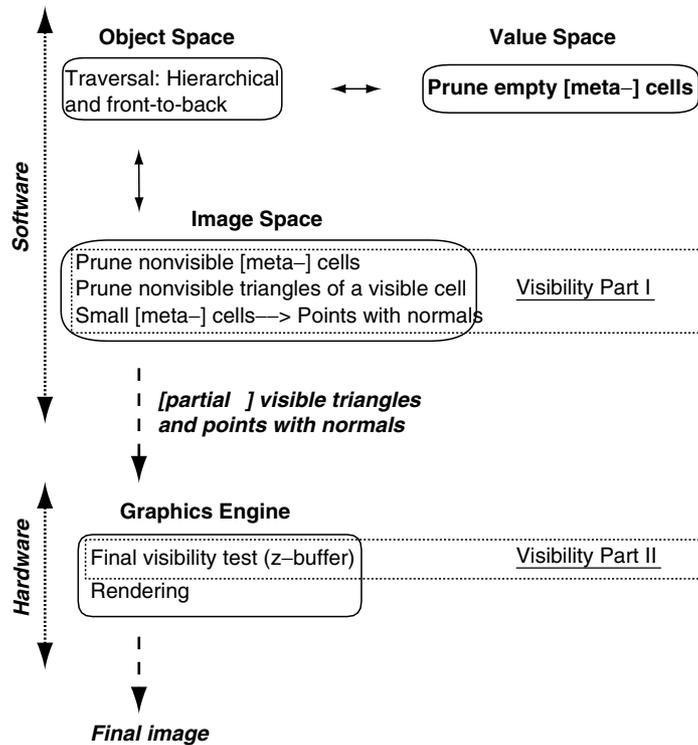


Figure 2.10 The algorithm data flow.

2.5.2 Visibility

Determining whether a meta-cell is hidden and thus can be skipped is fundamental to this algorithm. Toward this end, we create a virtual screen with only one bit per pixel. During the front-to-back traversal of the data we update this virtual screen with the projection of the triangles we extract. In effect, the virtual screen represents a dynamic visibility mask. At each stage of the traversal, the mask state represents the areas that are still visible from the user view point.

Determining whether a meta-cell is visible is accomplished by projecting the meta-cell on to the virtual screen and checking if any part of it is visible, i.e., if any of the pixels it covers is not set. If the entire projection of the meta-cell is not visible, none of its children can be visible.

2.5.3 Hierarchical Framebuffer

It is important to quickly and efficiently classify a cell as visible. A hidden cell and all of its children will not be traversed further, and thus the potential savings can justify the time and effort invested in the classification. A visible cell, on the other hand, does not gain any benefit from this test, and the cost of the visibility test is added to the total cost of extracting the isosurface. As such, the cell-visibility test should not depend heavily on the projected screen area; otherwise, the cost would prohibit the use of the test for meta-cells at high levels of the octree—exactly those meta-cells that potentially can save the most.

To achieve fast classification we employ a hierarchical framebuffer. In particular, each node in the hierarchy represents 64 (8×8)

children. The branch factor of 64 was chosen such that it can be represented in one word and so that comparison against the projection of a meta-cell can be done efficiently.

2.5.3.1 Top-Down Visibility Queries

The main purpose of the hierarchical framebuffer is to accelerate the classification of a meta-cell as *visible*. As such, it is important to know for each node of the framebuffer hierarchy if any part of it might be visible. Therefore, a node in the hierarchy is marked as opaque if and only if *all* of its children are opaque.

Determining whether a meta-cell is visible can now be done in a top-down fashion. The meta-cell is first projected onto the framebuffer, and an axis align bounding box is computed. This bounding box is then compared against the hierarchical framebuffer starting at the root node. The top-bottom ap-

proach accelerates the classification of the meta-cell as it can determine, at early stages, that some portion of the bounding box is visible.

2.5.3.2 Bottom-Up Updates

In order to keep the hierarchical framebuffer state current, we must update it as new triangles are extracted. However, a top-down update of the hierarchy is not efficient. If the extracted triangles are small, then each update of the hierarchy (i.e., rendering of a triangle) requires a deep traversal of the hierarchy. Such traversals are expensive and generally add only a small incremental change.

To alleviate the problem of projecting many small triangles down the hierarchy, we employ a bottom-up approach. Using this approach, the contribution of a small triangle is limited to only a small neighborhood at the lowest level, and thus only a few updates up the hierarchy will be necessary.

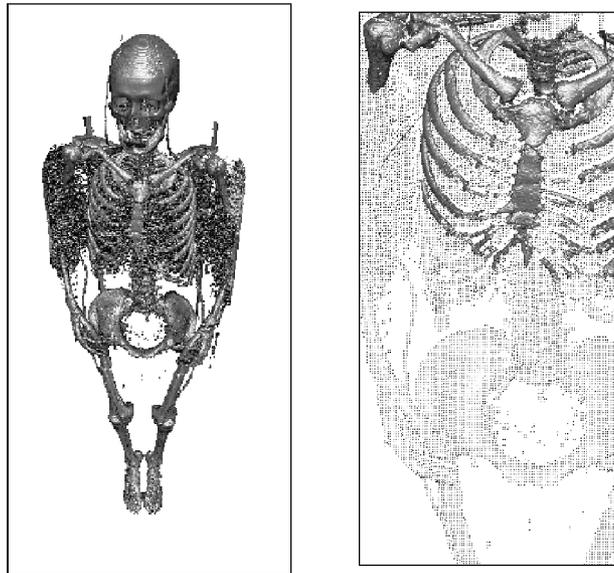


Figure 2.11 Rendering points. The left image was extracted based on the current view point. The right image shows a closeup of the same extracted geometry.

2.5.4 Scan Conversion of Concave Polygons

Once a data cell is determined to both be visible and intersect the isosurface, we use the marching cubes method to triangulate the cell. In most cases, the marching cubes creates more than one triangle, but it is not obvious which one of the triangles is in front of the other. As such, we need to render each one of them onto the framebuffer and forward all of them to the hardware for rendering.

Updating the hierarchical framebuffer one triangle at a time is not efficient, as the triangles from a single cell are likely to affect only a small section of the hierarchy and might even overlap. We thus employ a scan-conversion algorithm, which can simultaneously project a collection of triangles and concave polygons. The use of the scan-conversion algorithm is made particularly simple due to the bottom-up update approach. The projected triangles and polygons are scan-converted at screen resolution at the bottom level of the framebuffer hierarchy before the changes are propagated up the hierarchy. Applying the scan conversion in a top-down fashion would make the algorithm unnecessarily complex.

Additional acceleration can be achieved by eliminating redundant edges, projecting each vertex only once per cell and using triangle strips or fans. To achieve these goals, the marching cubes lookup table is first converted into a triangle fans format. The usual marching cubes lookup table contains a list of the triangles (three vertices) per case.

2.5.5 Rendering Points

Another potential savings is achieved by using points with normals to represent triangles or [meta-] cells that are smaller than a single pixel. The use of points in isosurface visualization was first proposed as the *Dividing Cubes Method* by Lorensen and Cline. Phister et al. [10] also used points to represent surface elements (surfels) for efficient rendering of complex geometry.

In a view-dependent approach, during the traversal of the hierarchy, whenever a nonempty [meta-] cell is determined to have a size less than two pixels and its projection covers the center of a pixel, it is represented by a single point. Note that the size of the bounding box can be almost two pixels wide (high) and still cover only a single pixel. Referring to Fig. 2.12, we require,

```

if (right - left < 2) {
    int L = trunc(left);
    int R = trunc(right)
if (L == R - 1)
    // create a point at R + 0.5
else if (L == R)
    // too small: does not cover the center
    of a pixel
else
    // too large : covers more than two
    pixels
}

```

and similarity for the bounding box height.

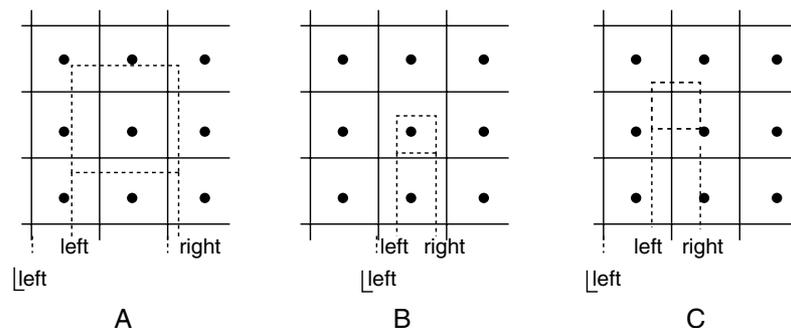


Figure 2.12 Pixel center and the projected bounding box. A point is created for cases A and B but not for C.

Figure 2.11 shows an example in which some of the projected cells are small enough that they can be rendered as points. On the left is the image as seen by the user, while on the right is a close-up view of the same extracted geometry (i.e., the user zoomed in but did not re-extract the geometry based on the new viewpoint). Notice that much of the image on the left is represented as points. Points are useful not only in accelerating the rendering of a large isosurface but also in remote visualization because less geometry needs to be transferred over the network.

2.5.6 Fast Estimates of a Bounding Box of a Projected Cell

The use of the visibility tests adds an overhead to the extraction process that should be minimized. Approximating the screen area covered by a meta-cell, rather than computing it exactly, can accelerate the meta-cell visibility tests. In general, the projection of a meta-cell on the screen has a hexagon shape with non-axis-aligned edges. We reduce the complexity of the visibility test by using the axis-aligned bounding box of the cell projection on the screen, as seen

in Fig. 2.13. This bounding box is an overestimate of the actual coverage and thus will not misclassify a visible meta-cell, though the opposite is possible.

The problem is in how to find this bounding box quickly. The simplest approach is to project each of the eight vertices of each cell onto the screen and compare them. This process involves eight perspective projections and either two sorts (x and y) or 16 to 32 comparisons.

Our solution is to approximate the bounding box as follows. Let P be the center of the current meta-cell in object space. Assuming the size of the meta-cell is (dx, dy, dz) , we define the eight vectors:

$$D = \left(\pm \frac{dx}{2}, \pm \frac{dy}{2}, \pm \frac{dz}{2}, 0 \right)$$

The eight corner vertices of the cell can be represented as

$$V = P + D = P + (\pm D_x, \pm D_y, \pm D_z)$$

Applying the viewing matrix M to a vertex V amounts to

$$VM = (P + D)M = PM + DM$$

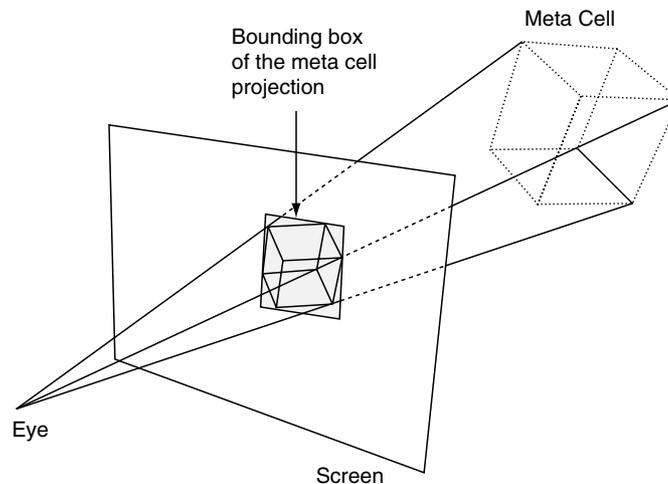


Figure 2.13 Perspective projection of a meta-cell, the covered area, and its bounding box.

After the perspective projection the x screen coordinate of the vertex is

$$\frac{[VM]_x}{[VM]_w} = \frac{[PM]_x + [DM]_x}{[PM]_w + [DM]_w}$$

To find the bounding box of the projected meta-cell, we need to find the minimum and maximum of these projections over the eight vertices in both x and y . Alternatively, we can overestimate these extreme values such that we may classify a nonvisible cell as visible but not the opposite. Overestimating can thus lead to more work but will not introduce errors.

The maximum x screen coordinate can be estimated as follows:

$$\begin{aligned} \max \left(\frac{[VM]_x}{[VM]_w} \right) &\leq \frac{\max([PM]_x + [DM]_x)}{\min([PM]_w + [DM]_w)} \\ &\leq \frac{[PM]_x + \max([DM]_x)}{\min([PM]_w + [DM]_w)} \\ &\leq \frac{[PM]_x + [D^+M^+]_x}{\min([PM]_w + [DM]_w)} \end{aligned}$$

where we define the $+$ operator to mean to use the absolute value of the vector or matrix elements.

Assuming that the meta-cells are always in front of the screen, we have

$$\begin{aligned} V_z > 0 &\Rightarrow P_z - D_z^+ > 0 \\ &\Rightarrow [PM]_z - [D^+M^+]_z > 0 \end{aligned}$$

Thus,

$$\max \frac{[VM]_x}{[VM]_w} = \begin{cases} \frac{[PM]_x + [D^+M^+]_x}{[PM]_w - [D^+M^+]_w} & \text{if numerator} \geq 0 \\ \frac{[PM]_x + [D^+M^+]_x}{[PM]_w + [D^+M^+]_w} & \text{otherwise} \end{cases}$$

Similarly, the minimum x screen coordinate can be overestimated as

$$\min \frac{[VM]_x}{[VM]_w} \leq \begin{cases} \frac{[PM]_x - [D^+M^+]_x}{[PM]_w + [D^+M^+]_w} & \text{if numerator} \geq 0 \\ \frac{[PM]_x - [D^+M^+]_x}{[PM]_w - [D^+M^+]_w} & \text{otherwise} \end{cases}$$

The top and bottom of the bounding box are computed similarly.

2.6 Summary

In Chapter 2, we classified the various approaches to the acceleration of isosurface extraction into three categories, namely geometric-based, value-based and image-based. We also presented two particular acceleration methods. The NOISE method is based on the span space representation of the value space and exhibits a worst-case complexity of $O(k + \sqrt{n})$. The view-dependent method is based on a front-to-back traversal, dynamic pruning (based on a hierarchical visibility framebuffer), and point representation of distant meta-cells.

References

1. J.L. Bentley. Multidimensional binary search trees used for associative search. *Communications of the ACM*, 18(9):509–516, 1975.
2. J.L. Bentley and D.F. Stanat. Analysis of range searches in quad trees. *Info. Proc. Lett.*, 3(6):170–173, 1975.
3. M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *J. of Computer and System Science*, 7:448–461, 1973.
4. D.T. Lee and C.K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(23):23–29, 1977.
5. Y. Livnat and C. Hansen. View-dependent isosurface extraction. In *Visualization '98*, pages 175–180. ACM Press, October 1998.
6. 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.
7. W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.
8. K.M. Oh and K.H. Park. A type-merging algorithm for extracting an isosurface from volumetric data. *The Visual Computer*, 12:406–419, 1996.
9. 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.

10. H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.
11. T. Poston, H.T. Nguyen, P.A. Heng, and T.T. Wong. ‘skeleton climbing’: ast isosurfaces with fewer triangles. In *Pacific Graphics '97*, pages 117–126, Seoul, Korea, October 1997.
12. R. Sedgewick. *Algorithms in C++*. Addison-Wesley, Massachusetts, 1992.
13. R. Shekhar, E. Fayyad, R. Yagel, and J.F. Cornhill. Octree-based decimation of marching cubes surfaces. In *Visualization '96*, pages 335–342. IEEE Computer Society Press, Los Alamitos, CA, 1996.
14. J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.
15. G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2:227–234, 1986.

AUTHOR QUERIES

Q1 Au: would you like to number this as a figure?

