EUROGRAPHICS '98 / N. Ferreira and M. Göbel
(Guest Editors)

Volume 17, (1998), Number 3

Multiresolution Isosurface Extraction with Adaptive Skeleton Climbing

Tim Poston† tim@ciemed.nus.edu.sg

Tien-Tsin Wong‡ ttwong@acm.org

Pheng-Ann Heng‡ pheng@cse.cuhk.edu.hk

†Center for Information-Enhanced Medicine, National University of Singapore
‡Dept. of Computer Science & Eng., The Chinese University of Hong Kong

Abstract

An isosurface extraction algorithm which can directly generate multiresolution isosurfaces from volume data is introduced. It generates low resolution isosurfaces, with 4 to 25 times fewer triangles than that generated by marching cubes algorithm, in comparable running times. By climbing from vertices (0-skeleton) to edges (1-skeleton) to faces (2-skeleton), the algorithm constructs boxes which adapt to the geometry of the true isosurface. Unlike previous adaptive marching cubes algorithms, the algorithm does not suffer from the gap-filling problem. Although the triangles in the meshes may not be optimally reduced, it is much faster than postprocessing triangle reduction algorithms. Hence the coarse meshes it produces can be used as the initial starts for the mesh optimization, if mesh optimality is the main concern.

1. Introduction

Standard isosurface extraction algorithms [1] generate an unwieldy number of triangles (half a million is common for a brain surface), making graphics and interactions unwieldy. This is inevitable with approaches which create triangles lying within voxel-sized cubes, even where a surface is smooth enough to be well approximated by much larger facets. Mesh reduction algorithms [2, 3, 4, 5] can greatly reduce the triangle count and preserve the geometrical details of the isosurfaces. Unfortunately, these postprocessing algorithms are usually time consuming. Hence they are only good for creating economical surfaces for later use. They are less useful when fast creation and display of isosurfaces are required, and when the exact threshold value is not certain. For instance, a surgeon may have to try different threshold values to explore the tumor surfaces. Rapid creation of accurate, economical isosurfaces is vital to many forms of volume data exploration, from neurosurgery to the planning of a gold mine.

We describe here a direct construction of isosurfaces with between 4 and 25 times fewer triangles than marching cubes algorithms [1, 6] (depending on the complexity of the volume), in comparable running times. Hence more complexity can be handled at interactive speed. The proposed algorithm is named as *adaptive skeleton climbing*. Since we construct the isosurfaces by first finding iso-points on grid edges (1-skeleton), then iso-lines on faces (2-skeleton) and finally isosurfaces within boxes (3-skeleton), this approach is known in topology as *skeleton climbing*. Moreover, the size of the constructed boxes will adapt to the geometry of the isosurface (*e.g.* larger boxes for smoother regions), hence it is *adaptive*.

Our approach is quite different from the previous adaptive marching cubes algorithms [7, 8]. We do not need a crack-patching step because we build compatibility (described shortly) into the faces where cells meet before generating triangles.

The proposed algorithm generates isosurfaces in multiple resolutions directly. The coarseness of the generated meshes is controlled by a single parameter. The triangle reduction is done on the fly as the isosurfaces are generated without going through a separate postprocess. The proposed on-the-fly triangle reduction approach can generate more accurate meshes because it directly make use of the voxel values in the volume. On the other hand, the postprocessing triangle reduction approaches [2, 3, 4, 5] usually use the indirect geometrical information from the approximated meshes.

The algorithm also exhibits a nice feature that coarser iso-

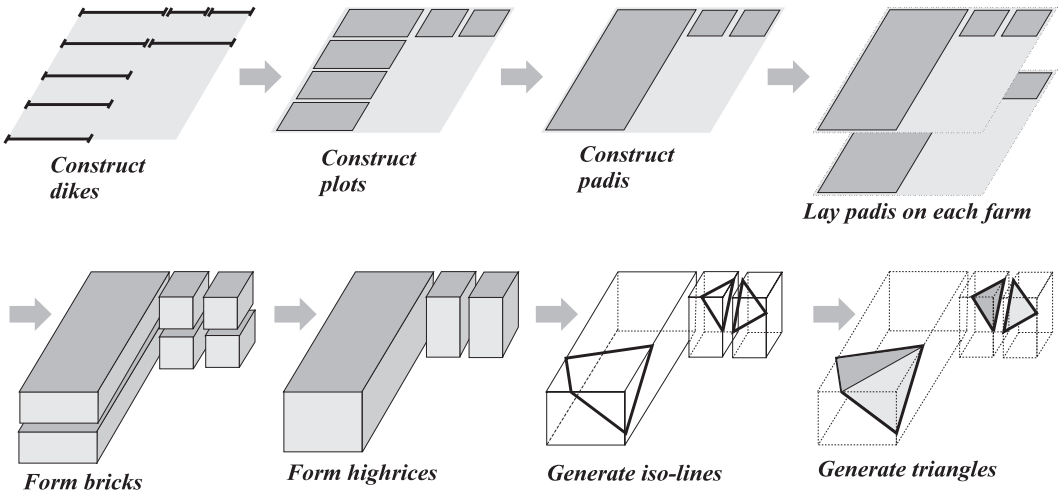


Figure 1: *Overview of adaptive skeleton climbing.*

surfaces require smaller amount of time to generate. This is opposed to the case of mesh optimization approach [3] which requires longer time to generate coarser meshes. Hence our method allows the user to first generate a low resolution mesh as a preview before deciding to generate the detailed high resolution mesh.

1.1. Overview

The algorithm can be intuitively subdivided into four steps:

1. Volume analysis.
2. Construction of simple boxes.
3. Sharing information between adjacent boxes.
4. Isosurface extraction.

In order to fit large triangles to smooth regions, the content inside the volume must be first analysed. The volume is implicitly analysed through the manipulation of basic 1D and 2D data structures in step 1. In step 2, the data structures built allow us to construct the 3D *simple* boxes (described in section 3) whose sizes are closely related to the geometry complexity of the enclosed isosurface. Information is then shared between adjacent boxes to prevent existence of gap in step 3. And finally in step 4, the triangular mesh is generated. Figure 1 shows the processes of adaptive skeleton climbing graphically. The basic idea is to group voxels first in 1D (segments), then in 2D (rectangles) and finally in 3D (boxes).

Section 2 describes the step of implicit volume analysis in detail. Section 3 discusses the construction of simple boxes. Information sharing step is described in section 4. Some details of triangular mesh generation are discussed in section 5. Section 6 discusses how the algorithm is used to generate multiresolution isosurfaces on the fly. Section 7 discusses the practical implementation, shows the results and compares them with marching cubes. Finally, section 8 gives our conclusions and future directions.

2. Volume Analysis

The first step of the algorithm is to analyse the volume through manipulating the basic 1D and 2D data structures. We start the analysis in 1D, *i.e.* consider a linear sequence of voxel samples. Try to find out the length-maximal subsequences of voxels with *simple* structure (described shortly). Then we go on to the 2D data structure and find out the size-maximal rectangular regions of voxel samples with *simple* structure.

2.1. 1D Data Structures and Manipulation

It helps to think of the volume data as giving sampled values at points (dots in Fig. 2), rather than voxel values filling cubes. For the sake of discussion, let's define the 1D terminologies and data structures. A line of 2^n+1 sample points is called *lign* (Fig. 2(a)) where n is an integer ≥ 0. A *dike* (Fig. 2(b)) is a segment of lign which covers voxel samples in the interval $[a2^m, (a+1)2^m]$, where $0 \leq m \leq n$ and $0 \leq a < 2^{n-m}$, both a and m are integers. That is, all dikes are organized in a binary tree (Fig. 3). The reason to use binary tree on 1D data instead of octree on 3D data [8] is that binary tree provides more flexibility in grouping voxels.

Fig. 3(b) shows the binary tree organization of 15 dikes which covers 9 voxels. The voxels covered by each dike are shown graphically in Fig. 3(c). The nodes are labeled in breadth-first-search order, with the root node as 1. With this dike-labeling scheme, we can store two length-$(2^{n+1}-1)$ arrays of dike information, *occupancy* and *simple dike*, for a

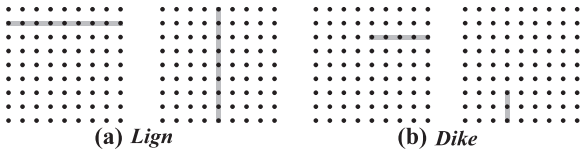


Figure 2: *Basic 1D data structures.*

lign of $2^n + 1$ samples. For simplicity, let $N = 2^n$ for short hand.

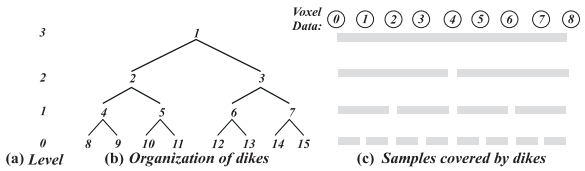


Figure 3: *Binary tree organization of 1D voxel data.*

The *occupancy array* of a lign describes the presence of iso-points (1D analogy of 3D isosurface) on its dikes. With this occupancy array, we can accurately locate the position of iso-point and how the isosurface crosses the lign. Now, let us denote the voxel sample with value above or equal to the threshold (τ) as $\bullet$, and sample with value below τ as $\circ$. Then the binary value of the i^{th} entry in the occupancy array means:

$$\texttt{occ[}i\texttt{]} = \begin{cases} 00_2 & \text{all samples in dike } i \text{ are on the same side of } \tau. \\ 01_2 & \text{if dike } i \text{ is crossed by isosurface once, upward } \circ \rightarrow \bullet. \\ 10_2 & \text{if dike } i \text{ is crossed by isosurface once, downward } \bullet \rightarrow \circ. \\ 11_2 & \text{if dike } i \text{ is crossed by isosurface more than once.} \end{cases}$$

Note the binary values symbolize the crossing conditions. For instance, if the isosurface crosses the dike once and the voxels within the dike change from $\circ$ (0) on the left to $\bullet$ (1) on the right, then the value in `occ[]` is 01_2 ($\circ \rightarrow \bullet$). Once the entries of unit dikes (leaf nodes of the binary tree) are initialized directly from volume data, the entries of the non-unit dikes (upper interior nodes) can be found by applying a recursive bitwise OR operations on the leaf nodes. The values in `occ[]` are specially designed.

```
occ[i] := (occ[2i]) OR (occ[2i + 1])
```

Another array is *simple dike array*. It tells us the length-maximal *simple* dikes inside the lign. A dike i is *simple* if $\texttt{occ}[i] < 11_2$; that is, the dike is crossed at most once by the isosurface. The entry `simple[`i`]` holds the index of the length-maximal simple dike with the same left end as dike i.

Intuitively speaking, simple dike array tells us which voxels can be grouped together without violating the binary boundary due to the tree organization (*binary edge* for short) and the simplicity constraints. The length-maximal simple step following dike i is the dike `simple[`i`+1]`. By performing the following pseudocode fragment, we can walk through the lign in steps of length-maximal dikes in an efficient way.

```
current := simple[1]
while current ≠ "end of walk" mark
    current := simple[current+1]
```

Fig. 4(a) illustrates that a lign is subdivided into length-maximal dikes (shown in black in Fig. 4(b)). In this 9-voxel lign example, the lign is subdivided into 4 dikes. The first two dikes are unit dikes, since the isosurface crosses both of them. Although the isosurface crosses the rest of the segment only once, it is still subdivided into two dikes due to the binary edge constraint imposed by the binary tree organization. Hence the subdivision may not be always minimal. But this restriction simplifies the merging process in the 2D adaptive skeleton climbing discussed in next section.

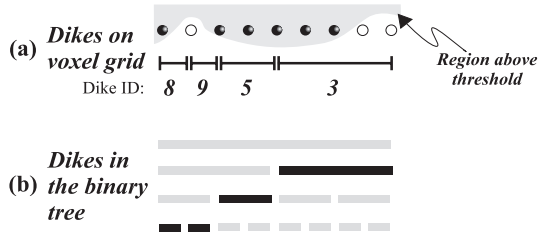


Figure 4: *(a): The lign is subdivided into length-maximal dikes. (b): The dikes visited when walking through the lign.*

2.2. 2D Adaptive Skeleton Climbing

2.2.1. Data Structures

The 1D data structures allow us to group voxels into length-maximal simple segments (dikes). Similarly, in the 2D, we want to group voxels to form size-maximal *simple* rectangles. Consider an $(N+1)\times(N+1)$ *farm* of voxel samples, with $N+1$ horizontal and $N+1$ vertical ligns, each with its own occupancy and simple dike arrays. First, let's define the 2D terminologies and data structures. A *strip* (Fig. 5(a)) consists of two consecutive ligns. A *plot* (Fig. 5(b)) is analogous to the dike which consists of two consecutive dikes.

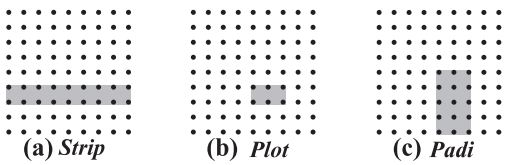


Figure 5: *The 2D data structures.*

Plots are also organized by a binary tree. Similarly a plot is *simple* if and only if its two dikes are also simple. Hence, we can define a *simple plot array* which is similar to the simple dike array. Since the shorter dike has a larger dike ID, the length-maximal simple plots can be easily found by

performing a MAX operation on each pair of elements in the simple dike arrays of the two consecutive ligns.

```
strip[j].simple[i] := MAX(lign[j].simple[i],
                          lign[j + 1].simple[i])
```

Fig. 6 shows one such operation graphically. The calculated plots are overlaid with the voxel samples in Fig. 6(b). Note that each plot is crossed at most twice by the isosurface.

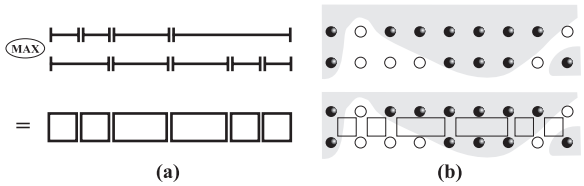


Figure 6: *Length-maximal plots from consecutive ligns.*

2.2.2. Merging Plots to Form Padis

A rectangle with dikes as sides is called *padi* (Fig. 5(c)). A padi is *simple* if all plots inside it and its four side dikes are simple. Our goal is to subdivide the 2D farm of voxels into size-maximal padis. To do so, neighboring simple plots are merged to form simple padis (Fig. 7), as large as possible. Note there is no unique way to merge plots. Different merging strategy gives different sets of padis. Fig. 7 shows two alternatives when merging the two consecutive strips. Even an optimal merging is found for 2D, it may not yield an optimal merging in 3D (discussed in next section). Moreover, a fast algorithm is crucially required since it will be frequently executed. A slow optimistic algorithm is useless in this case. Hence we do not use any optimistic algorithm to search for the optimal merging. A heuristic bottom-up merging (ASC2D, Fig. 8) is used due to its efficiency and simplicity.

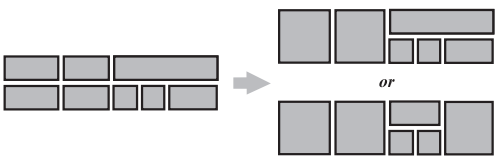


Figure 7: *Merging plots to form padis.*

The algorithm ASC2D accepts $2N$ initialized simple plot arrays as input. There are N arrays for horizontal strips and N arrays for vertical strips. The array can be initialized by the MAX operations discussed previously. The basic idea of the algorithm ASC2D in Fig. 8 is as follows. Let us denote the horizontal direction from left to right as direction x and vertical direction from bottom to top as direction y. For each length-maximal plot on each horizontal strip (x-strip), expand it in y direction by merging it with consecutive plots having the same length (Fig. 7). With the binary edge constraint, it is more likely to find neighbor plots with same

Input: $2N$ initialized simple plot or layout arrays (N for horizontal strips & N for vertical strips).
Output: A set of size-maximal padis (and iso-lines).
Algorithm:
Initialize an empty candidate list of padis.
For each x-strip (horizontal strip)
  /* Expand the plots to form padis */
  For each length-maximal simple plot a
    Let rectangle $r := a$
    While $\exists$ neighbor simple plot b on the adjacent strip
      $r := r \cup b$ (Fig. 7)
    Subdivide r in y-direction into pieces according to the binary edge restriction and give k padis $r_1, r_2, \ldots r_k$ (Fig. 9)
    For each generated padi r_i
      For each padi p_j inside the candidate list
        If p_j encloses r_i
          Delete r_i
        If r_i encloses p_j
          Remove p_j from the candidate list
        If p_j partially overlaps with r_i
          Clip r_i
      If r_i is not removed
        Add r_i to the candidate list
/* Optional Iso-line Generation */
For each padi p_j in the candidate list
  Generate iso-line for p_j by looking up the table(Fig.12).

Figure 8: *Algorithm* ASC2D.

length and align to each other. A candidate rectangle is then formed. Since the binary edge constraint is also applied to the vertical direction, this rectangle is subdivided to form size-maximal padis (Fig. 9).

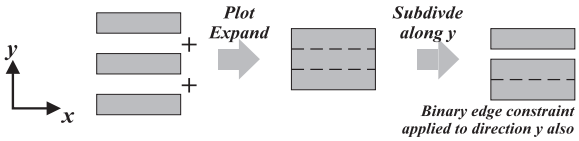


Figure 9: *Plots are first merged to form rectangle. The rectangle is then subdivided along y direction to satisfy the binary edge constraint applied to the y direction.*

During the execution of the algorithm ASC2D, many padis will be generated. They may overlap with each other or one may enclose another. All padis which are enclosed by any other padi will be removed. Those overlapping padis will be clipped with each other. Fig. 10 shows an example result of running the algorithm ASC2D. The generated padis are shown as rectangles among the voxel samples.

A layout of padis is generated as the result of ASC2D. This

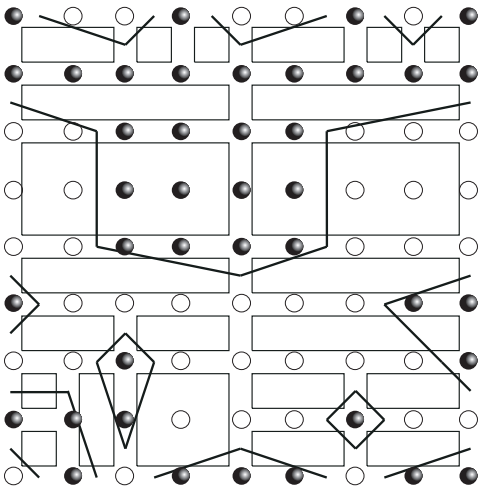


Figure 10: *Example result of running algorithm* ASC2D.

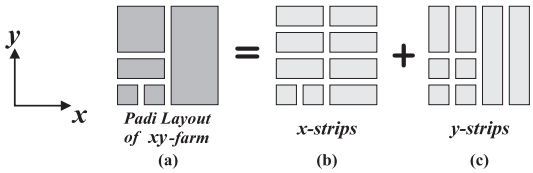


Figure 11: *Storing the padi layout in layout arrays.*

layout information is stored implicitly in the *layout arrays*. Layout array is very similar to the simple plot array but with the constraint that no plot may cross the boundary of any generated padi on the layout. For a farm of $(N+1) \times (N+1)$ voxels, $2N$ layout arrays are defined, N x-strips and N y-strips. The i^{th} entry in x-strip (y-strip) stores the index of the length-maximal plot that fits into the padi layout and shares its left (bottom) end with plot i. Fig. 11 shows the padi layout of a 5×5 xy-farm, which is represented by x-strips (Fig. 11(b)) and y-strips (Fig. 11(c)). The reason to store the layout in this way is to simplify the simple box construction discussed in section 3.

2.2.3. Iso-line Generation

Once the size-maximal padis are found, we can generate 2D iso-lines which separate • voxels from those ◦ voxels. Although we will not generate any iso-lines until the 3D boxes have been constructed (discussed in next section). For the sake of presentation, it is more convenient to discuss it here.

The iso-line can be efficiently generated by looking up a 2D padi configuration table in Fig. 12, instead of a 3D voxel cube configuration table as in marching cubes algorithms [1, 6]. Fig. 12 shows all possible padi configurations and their corresponding iso-lines. Note the padi needs not be a square. Similar to the 3D voxel configurations, ambiguity also exists on 2D padi configurations (the two lower left configurations in Fig. 12).

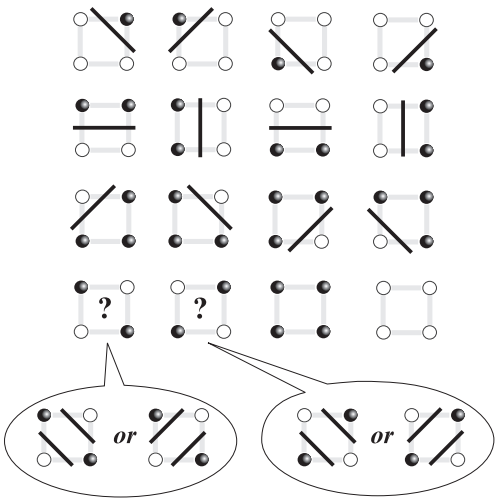


Figure 12: *Generate iso-lines by a 16-entry table. Two ambiguous cases lead to subsampling.*

The ambiguity with two diagonally opposite • corners can sometimes be resolved by subsampling at the center of the padi. However, wrong iso-lines will still be generated in some cases (Fig. 13). Where connectivity is crucial, software should warn the user of ambiguous cases and offer finer, more CPU-costly tools for local investigation. In many cases the warning is as useful to the surgeon, geologist or other user as any silently-attempted best guess by the software.

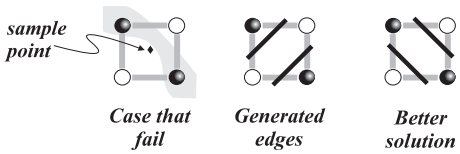


Figure 13: *Bad ambiguity resolution by subsampling.*

The generated padis (shown as rectangles) and iso-lines (shown as thick lines) are overlaid on the 2D voxel grid in Fig. 10. The algorithm isolates ◦ from • voxels, with 30 edges on 23 adaptive padis rather than the 46 edges on 64 unit squares. The feature is the key how the algorithm reduces triangles.

3. Construction of Simple Boxes

By manipulating these 1D and 2D data structures, enough information is provided for us to construct 3D *simple* boxes. The information is implicitly stored as the 2D padis. Using this information, we go on to construct simple boxes by stacking simple padis.

3.1. 3D Adaptive Skeleton Climbing

3.1.1. 3D Data Structures

Consider a $(N+1) \times (N+1) \times (N+1)$ voxel sample grid. For the sake of discussions, we now define the 3D terminologies and data structures. All terminologies are graphically illustrated by a $3 \times 3 \times 3$ volume in the Fig. 14(a). A *farm* contains $(N+1) \times (N+1)$ voxels on a 2D grid. A *slab*, analogous to a strip containing two consecutive ligns, consists of two consecutive farms. A *brick* in the slab has two matching padis in two consecutive farms as faces. A *highrice* is a rectangular box composed of stacked bricks.

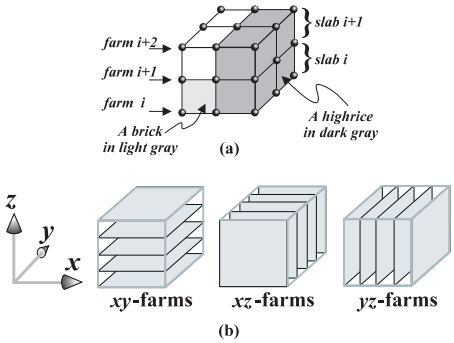


Figure 14: *Data structures for the 3D algorithm.*

It is convenient to denote a farm xy-farm, xz-farm or yz-farm, according to which plane the farm is parallel to (Fig. 14(b)). Similarly, a brick is called xy-brick if it is parallel to the xy plane.

3.1.2. Merging Bricks to Form Highrices

Our goal is to construct 3D *simple* rectangular boxes. We start by finding *simple* bricks. A brick is *simple* if the two padis forming it are also simple. A highrice is *simple* if all its component bricks are simple, and its six faces are *simple* padis.

Firstly, simple xy-bricks are identified. Then these simple xy-bricks will be stacked one by one to construct the simple xy-highrice. Note a highrice can be treated as composing of xy-bricks, xz-bricks or yz-bricks, depends on which dimension the bricks are stacked. We call a highrice the xy-highrice if it is constructed by stacking simple xy-bricks. In our algorithm, we only interest in finding the simple xy-highrices.

Fig. 15 outlines the main algorithm. The first step generates the padi layout on each farm by algorithm ASC2D without the iso-line generation step. Then we identify the simple xy-bricks by performing simple MAX operations for each pair of corresponding entries in the layout arrays on two consecutive farms. Just like the 2D version. An example is shown graphically in Fig. 16.

Input: An $(N+1) \times (N+1) \times (N+1)$ 3D voxel grid.
Output: A set of maximal highrices and isosurface triangular mesh.
Algorithm:
/* Generate the padis on each farm */
For each farm out of (xy-farm, xz-farm and yz-farm)
    Find size-maximal padis by ASC2D.
Set layout arrays based on padi layout on each xy-farm.
For each xy-slab
    Find layout of xy-bricks by MAX operations.(Fig.16)
Initialize an empty candidate list of highrices.
/* Stack the bricks to form highrice */
For each xy-slab
    For each xy-brick a
        Let rectangular box $r := a$
        While $\exists$ neighbor simple xy-brick b on the adjacent xy-slab
            $r := r \cup b$ (Fig. 17)
        Subdivide r into xy-highrices with the binary restriction applied along z and give k xy-highrices $r_1, r_2, \ldots, r_k$ (Fig. 18)
        For each generated xy-highrice r_k
            For each xy-highrice h_l in the candidate list
                If h_l encloses r_k
                    Delete r_k.
                If r_i encloses h_l
                    Remove h_j from the candidate list.
                If h_l partially overlaps with r_i
                    Clip r_l.
            If r_k is not removed
                Add r_k to the candidate list
/* Sharing information among highrices */
For each farm (xy-farm, xz-farm, yz-farm)
    Reinitialize layout array to fit xy-highrice boundaries (Fig. 20)
    Find a padi layout with ASC2D using new layout values
/* Iso-line Generation */
For each xy-highrice in the final candidate list
    For each padi on the surface of the xy-highrice
        Generate iso-lines. (Fig. 12)
    Connect the iso-lines to form loops
    For each edge loop on the surface of xy-highrice
        Triangulate it and emit the triangles.

Figure 15: *Algorithm* ASC3D.

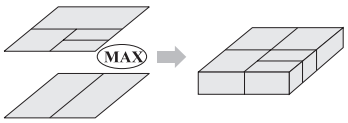


Figure 16: *To find the simple bricks inside the slab,* MAX *operations are done on each pair of the layout arrays of neighboring farms.*

```
xy-slab[k].x-strip[j].layout[i] :=
              max(xy-farm[k].x-strip[j].layout[i],
                  xy-farm[k + 1].x-strip[j].layout[i])
xy-slab[k].y-strip[j].layout[i] :=
              max(xy-farm[k].y-strip[j].layout[i],
                  xy-farm[k + 1].y-strip[j].layout[i])
```

Next, neighbor bricks merge to form highrices in 3D (Fig. 17), analogous to merging plots to form padis in the 2D case (Fig. 7). Again there is no unique merging rule (Fig. 17), and again we prefer a fast heuristic to a search for a suboptimal subdivision. For each size-maximal xy-brick on each slab, we stack the xy-bricks upward along z direction until no more simple bricks available. Then this temporary box is subdivided along z direction to fulfill the binary edge constraint (Fig. 18). During the highrice formation, the generated highrices may overlap with each other or one may be enclosed by another. Any enclosed highrice will be removed. Overlapping highrices are clipped.

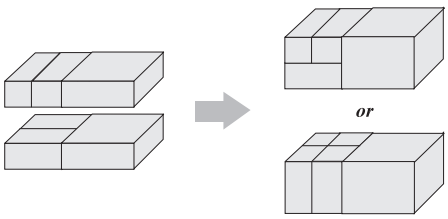


Figure 17: *Merging bricks to form highrices*

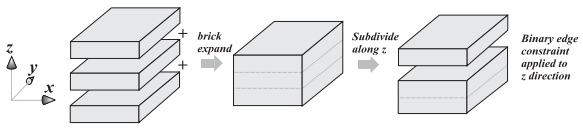


Figure 18: *Bricks are first merged to form box. Then the box is subdivided along z to form highrices in order to fulfill the binary edge constraint.*

4. Sharing Information Between Highrices

At this moment, we can immediately generate triangles inside each xy-highrice with the padi layout on the surface on the xy-highrices. This will yield triangular mesh with crack, just like the cases of Shu *et al.* [7] and Shekhar *et al.* [8], since the boxes may not be unit cubes.

Fig. 19(a) shows a large highrice next to a small one. The isosurface crosses the plane separating the two highrices (Fig. 19(b)). If triangles are emitted for each highrice without the knowledge of their neighbors, gaps will appear in the generated triangular mesh. This is because the linear iso-lines generated on the highrice surfaces may not match each other geometrically (Fig. 19(c)), even though they are topologically correct. To prevent this mismatch, information must be shared between adjacent highrices.

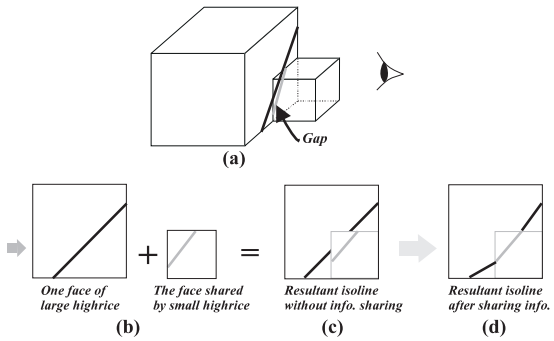


Figure 19: *Sharing information between neighbor highrices.*

In our algorithm, the neighbor information can be shared by manipulating the basic data structures. Recall that we store the padi layout of each farm in layout arrays in section 2.2.2. Layout arrays are variants of simple plot arrays. We can reuse these arrays with the new constraint that no plot may cross the boundary of any face of any generated xy-highrice. That is, we store the 3D highrice layout in these arrays this time. Fig. 20 shows the farm between the two highrices in Fig. 19. The surface boundary of the larger highrice is shown as thick dark gray line in the farm of Fig. 20(b), while that of the smaller highrice is shown as thick light gray line.

Once the layout arrays are reinitialized, algorithm ASC2D is executed on them (instead of simple plot arrays) to give a new set of padis. Since the length-maximal plots represented by the layout arrays are not allowed to cross any boundary, the generated padis will fit inside these boundaries. Fig. 20(c) shows the generated padis for the previous example. After generating iso-lines on each padi, three segments of iso-lines will be generated in the example (Fig. 20(d)), therefore no gap will exist.

5. Isosurface Extraction

Instead of thinking the padis are laid on the farm, they can also be regarded as padis laid on the six faces of each xy-highrice. Each face of the xy-highrice may contain more than one padis as in Fig. 21.

To generate the isosurface, we first generate iso-lines

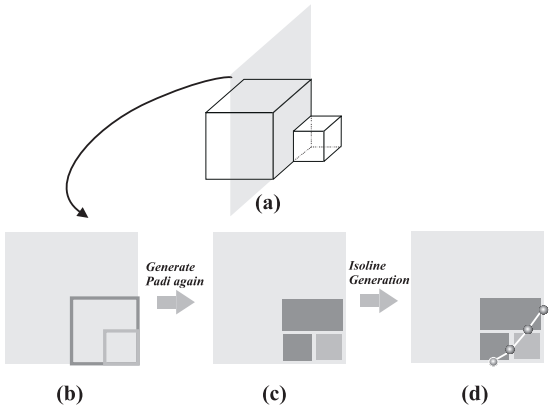


Figure 20: *Once we reinitialize the layout arrays to store the 3D highrices' layout,* ASC2D *can be run to generate padis that fit into the surface boundaries of both highrices.*

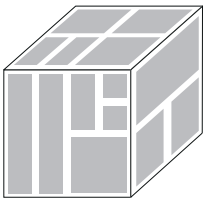


Figure 21: *The six faces of a highrice are tiled with padis after information sharing.*

on padis by looking up the 2D padi configuration table in Fig. 12, and connect them to form closed edge loops (Fig. 23(a)). Note that in our algorithm, we only need a 2D padi configuration table, no 3D voxel cube configuration table is needed.

Given an edge loop consisting of several vertices v_i, we emit triangles as follows. In each iteration, three consecutive vertices v_i, v_{i+1} and v_{i+2} are selected and one triangle is generated (Fig. 22). The vertex v_{i+1} is then removed from the edge loop. The algorithm continues until only two vertices are left.

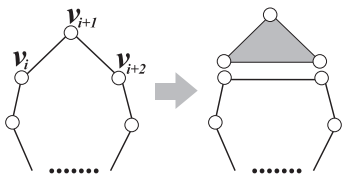


Figure 22: *In each iteration, one triangle is emitted and one vertex is removed.*

An edge loop can be triangulated in multiple ways. Different sequences give triangular meshes with identical triangle counts, but with different geometry (Fig. 23(b) and (c)). To generate a mesh that closely approximates the true isosurface, we make use of the gradient. We reject any triangle with planar normal vector $\vec{n_t}$ that largely deviates from the gradients $\vec{g_i}$ at three vertices. The deviation is measured by the dot product of $\vec{n_t}$ and $\vec{g_i}$. A threshold is used as a criteria. The threshold constraint will be relaxed if no triangle can be generated under current constraint.

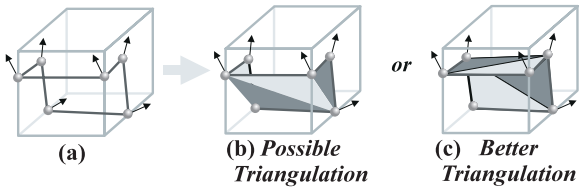


Figure 23: *Triangulate the edge loop to emit triangles.*

6. Multiresolution Isosurface Extraction

The proposed algorithm handles volume with the size of $(N+1) \times (N+1) \times (N+1)$, *i.e* a cubic *block*. To handle volume with different size, we can simply tile the blocks to cover the whole volume and apply ASC3D to each block. Recall that gaps will appear if no information is shared between adjacent highrices. Similarly, cracks will appear if information is not shared between adjacent blocks. Unlike the case of variable-sized highrices, each block has the same size. This simplifies the process. To share information between blocks, we simply perform MAX operations on each pair of layout arrays on the surfaces (which are also farms) of two adjacent blocks. The simple MAX operations effectively find out the largest padis that fit the constraints. This information sharing process must be done just after the information sharing among highrices.

Up to this moment, we have not yet discussed the effect of using different values of N, *i.e.* the size of the block. The block size constrains the maximum size of the highrices. When the block size is small, say $N = 1$, the largest highrice contains $2 \times 2 \times 2$ voxels, *i.e.* same as standard marching cubes. When a larger block size is used, larger highrices are allowed to be generated, hence larger triangles. In other words, by controlling the value N, we can generate isosurfaces in multiresolution. Note that parameter N is an indirect control, the actual mesh generated will also depend on the geometry of the true isosurface. More triangles will still be generated if the isosurface geometry is complex. Figure 25(b)-(e) shows the results of using different block sizes. From (b) to (e), the values of N are 1, 2, 4 and 8. As the block size increases, larger triangles are generated to approximate the smooth surface.

Unlike the triangle reduction algorithms [2, 3, 4, 5] which generate coarser mesh based on the high resolution mesh, the proposed approach generates coarser mesh directly from the

original volume data. This ensures no distortion or error is introduced before the triangle reduction. More importantly, the proposed algorithm is a on-the-fly process which requires no time-consuming postprocessing triangle reduction. In fact, the algorithm produces coarser mesh in a smaller amount of time (see Table 1). This is quite different from those triangle reduction algorithms. Although our approach may not reduce triangles as much as mesh optimizer does, it is a cost effective method to significantly reduce triangles in a short period of time.

7. Implementation and Results

In practical implementation, there is no need to process every block of voxels. Since many blocks are empty, *i.e.* contains no isosurface, we can simply ignore them without performing the computation intensive merging processes. This can be done in the early stage of the algorithm. Once we have initialized the values in `OCC[]` for each lign in the block, the emptiness of the block can be immediately identified.

Table 1 and Fig. 24 quantify for various datasets the results of our implementation of adaptive skeleton climbing with four block sizes, $N = 1, 2, 4$ and 8. Triangle counts and CPU times on an SGI Onyx are compared with the Wyvill implementation [6] of marching cubes algorithm. Fig. 25 shows the corresponding images. Gouraud shaded isosurfaces are overlaid with triangle edges for clarity.

The "knot" data sets are sampled from an algebraic function, at three resolutions. Fig. 25(b–e) show the extracted isosurfaces in multiple resolutions, while Fig. 25(a) shows the isosurface generated by the marching cubes algorithm. The mesh generated by marching cubes method and that by the proposed method with $N = 1$ are visually very similar. Volume "Mt. Alps" (Fig. 25(f–j)) is a landscape heightfield dataset. We also tested two medical computed tomography (CT) datasets, "Head" (Fig. 25(k–o)) and "Arteries" (Fig. 25(p–t)). The data set "Arteries" contains no large smooth sheets, and its geometrical complexity inherently requires a finer mesh for topological correctness.

In general, as the block size N increases, both the triangle count and the CPU time decrease (Fig. 24 (a) and (b)). There are about four to twenty-five times reduction in the triangle count. Fig. 25 reveals little change in shape as the triangle count decreases. Note that in some cases increasing the block size N may slightly increase the triangle count when the complex geometry of the isosurface requires sufficient triangles to represent. In three out of six tested cases, the optimal block size (in term of triangle count) is $N = 4$. Depend on the geometry complexity of the true isosurface, this optimal value may vary. From the experiments, the proposed algorithm is not efficient to generate the highest-resolution mesh as the marching cubes algorithms do. However, it can generate coarser meshes in an amount of time comparable to that of marching cubes algorithm. In the test cases of "knot256" and "Mt. Alps", the running times of generating coarse meshes ($N = 8$) are faster than that of marching cubes. This result suggests that the coarse mesh generated by adaptive skeleton climbing may be a better initial start for mesh optimization than the highest resolution mesh from marching cubes, since the extra time spending on generating coarse mesh using our method is usually smaller than the extra time spending in the mesh optimizer if the initial start is the highest resolution mesh.

8. Conclusions and Future Directions

Adaptive skeleton climbing produces isosurfaces in times comparable to marching cubes [1], with substantially fewer triangles, and without the gap-filling problems of adaptive marching cube methods [9]. It directly uses the volume data and produces isosurface in multiple resolutions.

Since we use simple rectangular boxes instead of octree cubes, this approach provides more flexibility in partitioning the volume, hence captures more isosurface regions with simple geometry. It is faster than postprocessing mesh simplification methods [3, 4, 5, 2], though the mesh may not be optimally reduced. The proposed algorithm can serve as a companion to the mesh optimizer, since the coarse mesh it produced can be a better initial guess for the optimizer. Adaptive skeleton climbing is a fast heuristic algorithm, rather than a path to a strict optimum.

One future research direction is to further speedup the algorithm by combining an indexing scheme such as the kd-tree indexing approach [10, 11] which can rapidly and efficiently locate the non-empty cells (those cells contain isosurface). Another direction is to parallelize the algorithm. Besides the information sharing process, all other parts of the algorithm can be easily parallelized. We are developing a multithread implementation of the proposed algorithm. Further speedup is expected since the information sharing process uses only a small fraction of computational time.

Web Availability

A robust implementation of the algorithm is available for download at the web site:
`http://www.cse.cuhk.edu.hk/~ttwong/papers/asc/asc.html`

Acknowledgements

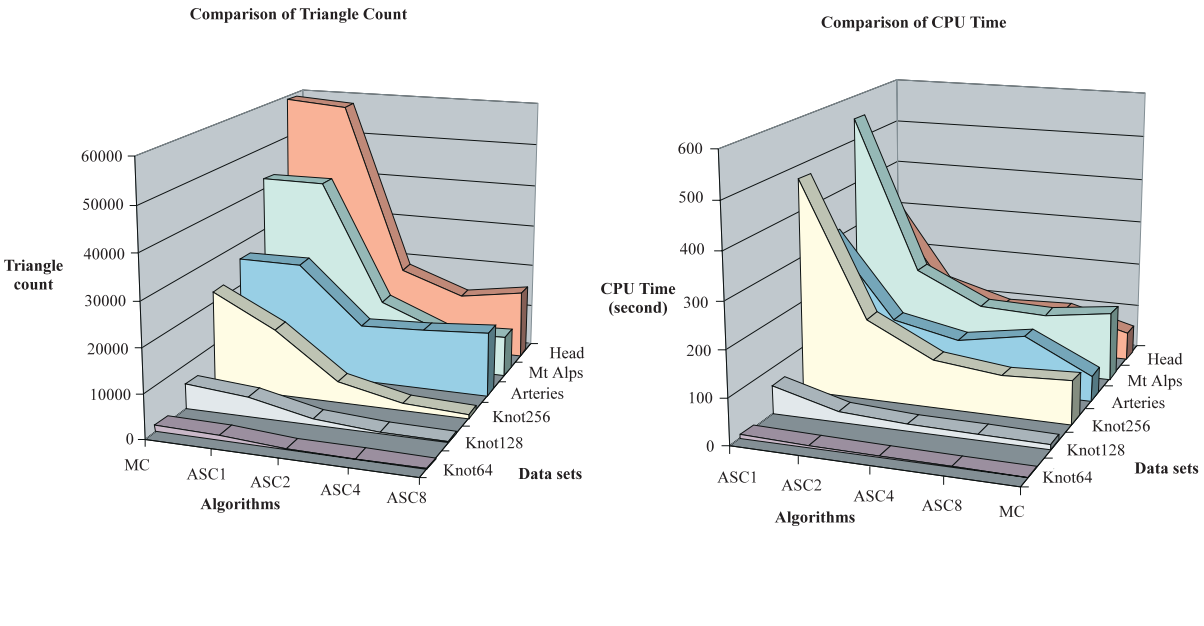
This work is supported by RGC Earmarked Grant CUHK4162/97E of Hong Kong and CUHK Direct Grant. We would also like to thank Dr. Tushar Goradia at Johns Hopkins University for providing the "Arteries" data set.

References

1. William E. Lorensen and Harvey E. Cline, "Marching cubes: A high resolution 3D surface construction algo-

Data Set	ASC, $N = 1$	ASC, $N = 2$	ASC, $N = 4$	ASC, $N = 8$	MC
knot64	12,712△	3,682△	1,772△	2,054△	13,968△
$64 \times 64 \times 64$	8.16*sec.*	3.61*sec.*	2.59*sec.*	2.43*sec.*	1.79*sec.*
knot128	44,760△	13,088△	4,692△	3,918△	56,208△
$128 \times 128 \times 128$	61.52*sec.*	24.00*sec.*	15.61*sec.*	14.31*sec.*	12.81*sec.*
knot256	152,080△	48,562△	15,370△	8,829△	225,736△
$256 \times 256 \times 256$	470.95*sec.*	178.54*sec.*	105.31*sec.*	87.78*sec.*	94.62*sec.*
Mt. Alps	423,638△	147,562△	90,434△	94,339△	423,640△
$258 \times 258 \times 256$	547.70*sec.*	207.54*sec.*	137.70*sec.*	132.83*sec.*	151.05*sec.*
Head	580,771△	186,331△	136,909△	159,207△	592,368△
$256 \times 256 \times 113$	339.21*sec.*	138.59*sec.*	97.31*sec.*	100.55 *sec.*	61.91*sec.*
Arteries	263,686△	131,769△	139,636△	149,251△	263,438△
$256 \times 256 \times 148$	311.97*sec.*	134.00*sec.*	103.68*sec.*	128.07*sec.*	56.09*sec.*

Table 1: *Comparison of marching cubes (MC) and adaptive skeleton climbing (ASC), with block sizes $N = 1, 2, 4, 8$, and marching cubes, in term of triangle count (△) and CPU time.*



(a) (b)

Figure 24: *Graphical presentation of the results shown in Table 1. (a) Graph of triangle count. (b) Graph of CPU time.*

rithm," in *Computer Graphics (SIGGRAPH '87 Proceedings)*, July 1987, vol. 21, pp. 163–169.

2. Michael J. Dehaemer and Michael J. Zyda, "Simplification of objects rendered by polygonal approximations," *Computer & Graphics*, vol. 15, no. 2, pp. 175–184, 1991.

3. Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle, "Mesh optimization," in *Computer Graphics (SIGGRAPH '93 Proceedings)*, August 1993, pp. 19–26.

4. William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen, "Decimation of triangle meshes," in *Computer Graphics (SIGGRAPH '92 Proceedings)*,

July 1992, vol. 26, pp. 65–70.

5. Greg Turk, "Re-tiling polygonal surfaces," in *Computer Graphics (SIGGRAPH '92 Proceedings)*, July 1992, vol. 26, pp. 55–64.

6. B. Wyvill and D. Jevans, "Table driven polygonisation," SIGGRAPH *1990 course notes* **23**, pp. 7–1–7–6, 1990.

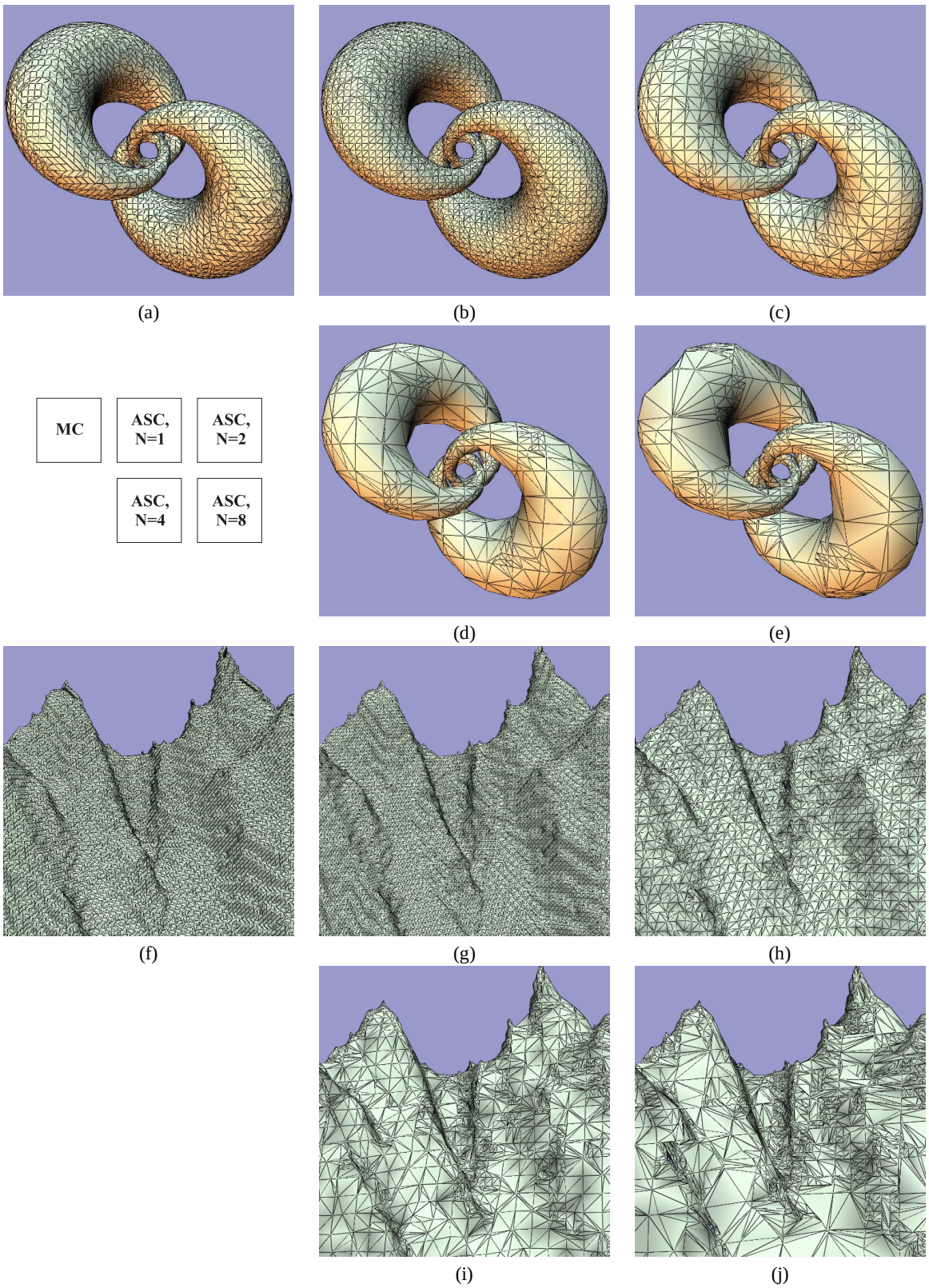
7. R. Shu, Z. Chen, and M. S. Kankanhalli, "Adaptive marching cubes," *The Visual Computer*, vol. 11, pp. 202–217, 1995.

8. R. Shekhar, E. Fayyad, R. Yagel, and J. Cornhill, "Octree-based decimation of marching cubes surfaces," in *IEEE Visualization '96 Proceedings*, Oct 1996, pp. 335–342.

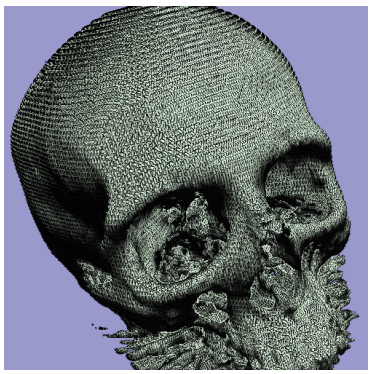
9. Jane Wilhelms and Allen Van Gelder, "Octrees for faster isosurface generation," *ACM Transactions on Graphics*, vol. 11, no. 3, pp. 201–227, July 1992.

10. Jon Louis Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, September 1975.

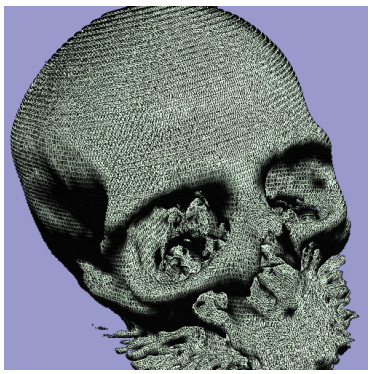
11. Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson, "A near optimal isosurface extraction algorithm using the span space," *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73–84, March 1996.



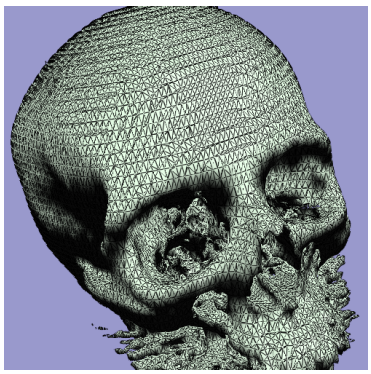
[Poston, Wong & Heng] Figure 25. Visual comparison of the effects of block size, for an algebraic surface, a landscape, and CT data for bones and blood vessels in the head. (a)-(e): Mathematical data set ”knot64”. (f)-(j): Landscape data ”Mt. Alps”.



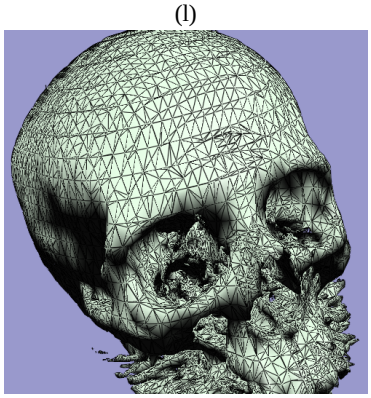
(k)



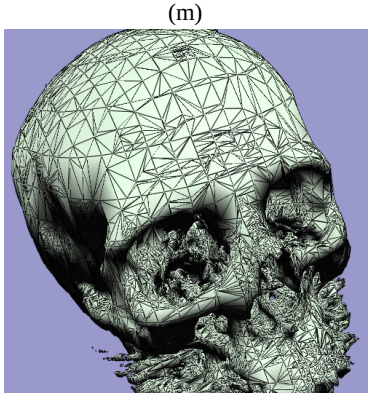
(l)



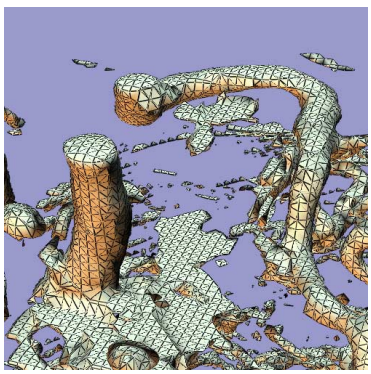
(m)



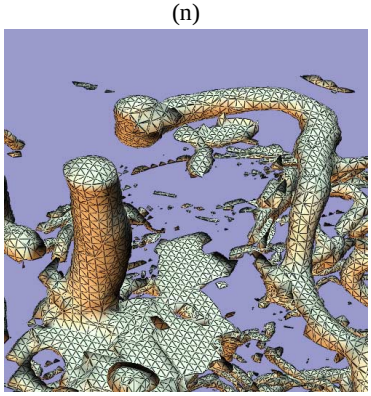
(n)



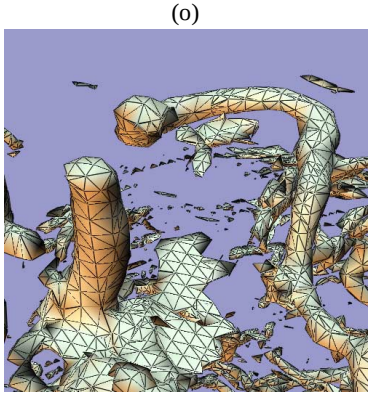
(o)



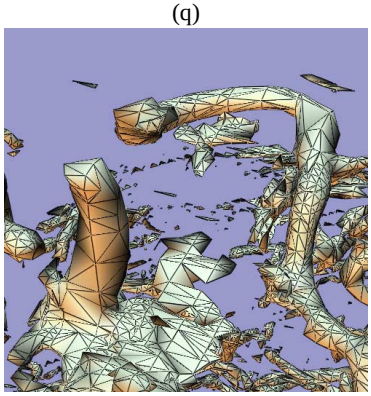
(p)



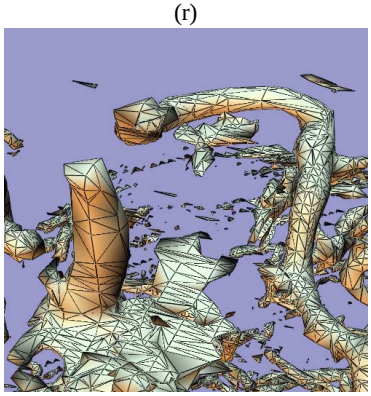
(q)



(r)



(s)



(t)

[Poston, Wong & Heng] Figure 25 (cont'd) Visual comparison of the effects of block size, for an algebraic surface, a landscape, and CT data for bones and blood vessels in the head. (k)-(o): "Head". (p)-(t): "Arteries".