Pergamon

Comput. & Graphics, Vol. 19, No. 6, pp. 805–813, 1995

0097-8493/95 $9.50 + 0.00

0097-8493(95)00051-8

Technical Section

# EFFICIENT GENERATION OF ISOSURFACES IN VOLUME RENDERING†

JUNG-HONG CHUANG‡ and WOAN-CHIAUN LEE

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, ROC

**Abstract**—An efficient method for extracting isosurfaces from volume data is proposed. The method utilizes a modified branch-on-need octree to bypass regions of no current interest. In addition, during the generation of triangle meshes neighboring triangles are merged according to certain criteria. Methods are also given to significantly reduce the space required for octrees. The method is more efficient and generates far fewer triangles than the marching cube algorithm. The performance of the proposed method is compared with that of several existing methods.

## 1. INTRODUCTION

Many scientific computations and medical applications produce volumetric data sets defined on 3-D grids. Since volume data is generally complex it is hard to understand without visual displays. Volumetric data can be visualized by either *surface rendering* [1, 2] or *direct volume rendering* [3, 4]. Surface rendering techniques extract the isosurface of a particular threshold and approximate the surface by intermediate polygonal meshes. Although the surface rendering is generally simple and efficient, it is unable to effectively display the interior of the volumetric data. Direct volume rendering, on the other hand, directly visualizes the volumetric data without generating any intermediate geometry and allows the user to probe into the interior of the volumetric data.

The marching cube method proposed in Ref. [1] has been recognized as an effective and simple method for isosurface extraction; nevertheless, this method has three notable problems. First of all, the marching cube method explores all cells, including those containing no surface of interest [5]. Second, the marching cube method generates an excessive number of triangles [6–8]. Finally, this method may produce surfaces with holes, because an ambiguity may be found on the common face of adjacent cubes [9, 10].

In this paper, a two-phase method is proposed to resolve the first two of these problems. The method combines and extends the octree structure proposed in Ref. [5] and the splitting-box approach in Ref. [7]. In the first phase, the set-up phase, a branch-on-need octree is constructed and represented as a linear octree. The second phase performs mesh reduction using the branch-on-need bisection tree while generating triangle meshes for the isosurface. Methods are proposed to greatly reduce the amount of memory required for the octree. Comparative studies show that the proposed method is effective in speeding up the surface generation, in reducing the size of triangle meshes, and in reducing the amount of memory required.

## 2. THE MARCHING CUBE ALGORITHM

In the marching cube algorithm, a cube is formed by eight adjacent data points on two consecutive slices. Cubes are processed in a row–column and slice by slice order. Surface points on the edges of cubes are found by linear interpolation and their normals are derived by using the central difference.

The basic marching cube algorithm traverses all cubes, including cubes that contain no surfaces of interest. This useless exploration can be avoided by representing the volumetric space with a hierarchical structure such as an octree, especially the branch-on-need octree (BONO) proposed in Ref. [5]. If each node of the octree is associated with the minimum and maximum densities in the subvolume corresponding to that node, exploration of the subvolume will be unnecessary if the threshold is outside the range of the minimum and maximum densities. The characteristic of the branch-on-need octree is to delay subdivision until it is absolutely necessary. To construct the BONO for a volumetric space, we first convert ranges of the volume in the $x$, $y$, and $z$ directions into binary code. Note that the range of the volume in an axis is one less than the number of intervals between data points in that axis. The directions of the volume that must be subdivided are those whose ranges have 1 as the leftmost bit. The designated range is split into the *lower* part and the *upper* part. The lower part always covers the largest possible exact power of 2, that is, the lower part is a

† Supported by the National Science Council of the ROC under grant NSC 82-0408-E-009-428.
‡ Author for correspondence.

Table 1. Branch-on-need subdivision of a $5 \times 4 \times 3$ volume

| Direction | Parent | | Lower | | Upper | |
|---|---|---|---|---|---|---|
| | Range | Binary | Range | Binary | Range | Binary |
| $x$ | 4 | 100 | 3 | 011 | 0 | 000 |
| $y$ | 3 | 011 | 3 | 011 | 3 | 011 |
| $z$ | 2 | 010 | 2 | 010 | 2 | 010 |

bit string of 1's which is one bit shorter than the original code. The upper part is the original code with the leftmost 1 bit removed. Consider a volume of resolution $5 \times 4 \times 3$ as an example. To see how the BONO is built, we first convert three ranges $5-1=4$, $4-1=3$, and $3-1=2$ into binary code as shown in Table 1. Since only the range of the $x$ direction has a 1 as the first bit, the volume is subdivided in the $x$ direction in level one and the range of the $x$ direction is split into two parts. The code of the lower part of the corresponding subdivision is a bit string of 1's that is one bit shorter than the original code. The code of the upper part is the original code with the leftmost bit removed. The codes of the ranges in the $y$ and $z$ directions remain unchanged (Table 1).

The number of triangles produced by the basic marching cube algorithm is generally large. The triangle meshes can in general be reduced either during or after the surface generation [7, 8]. The splitting-box (SB) algorithm proposed in Ref. [7] does the mesh reduction while generating the triangle mesh and results in a reduced triangle mesh with an error less than the size of the cell. The MC property is used throughout the splitting-box algorithm. An edge of a box is called MC if it possesses at most one transition of the isosurface. A face of a box is MC if all its four edges are MC. A box is called MC if its faces are all MC faces. The SB algorithm takes the entire volume data as the initial box and recursively bisects the box down to the cell level. The box is bisected perpendicularly to its longest edge or randomly if the edges are of equal length. Whenever a box becomes MC during the bisection process, *i.e.* all its edges contain at most one transition either from black to white or from white to black, the contour chains (polygons) in the box are found by the standard marching cube algorithm. The contour chains derived are passed to the descendant boxes after the bisection and are checked to see if they are 'legitimate' approximations of the contour chains found in the descendant boxes. Each contour chain is associated with a Boolean *valid* to identify whether the chain is a legitimate approximation. If a contour chain in the upper level of the bisection process is a legitimate approximation of the contour chains in the lower level, the latter are replaced by the former.

## 3. EFFICIENT GENERATION OF ISOSURFACES

The splitting-box algorithm traverses all cells, including cells in regions of no current interest, and results in a time complexity of $O(n^3)$, where $n^3$ is the number of input grid points [7]. In this section, we present an algorithm that first constructs the branch-on-need octree and from which a *branch-on-need bisection tree* is derived. With the branch-on-need bisection tree, the number of triangles generated is reduced by adapting the size of triangles to the surface's shape. Methods are also given to greatly reduce the memory required for the BONO. This is crucial when only limited memory space is available.

### 3.1. *Linear branch-on-need octree*

Each node of the BONO contains fields for storing the *minimum-density*, *maximum-density*, *branch*, and *address*. The *minimum-density* and *maximum-density* represent the minimum and maximum densities, respectively, in the volume represented by the node. The *branch* represents how the node is subdivided; the branch is usually three bits long, with 1 representing the required subdivision in a particular direction. The *address* field is usually a pointer that points to the leftmost child of the node.

Since the number of nodes on each level of the BONO can be computed from the binary representations of the ranges, the space required for storing the linear BONO can be allocated beforehand. In the linear BONO, the four-byte-long pointers can be replaced by an *index* which normally requires only three bytes. The *index* in the record of the linear BONO refers to the record of the leftmost child of the node. The index of the $i$-th leftmost child of a node can be referred to by adding $i-1$ to the index of the node. The scheme evidently allows to traverse the whole tree. Each record of the linear BONO contains *minimum-density*, *maximum-density*, *branch*, and *index*. Figure 1 depicts a BONO with one node on level one, four nodes on level two, and 12 nodes on level three, together with the corresponding linear BONO. In the linear octree, the record for the leftmost child of node 2 can be indicated by the *index* of node 2, which is 7. The record for the third leftmost child of node 2 can be indicated by adding 2 to the *index* of node 2, which is 9.

### 3.2. *Branch-on-need bisection tree*

After constructing the linear BONO for the given volumetric data, we next traverse the BONO to generate the isosurface for a given threshold and adapt the size of the polygon meshes to the shape of isosurface. As the BONO is traversed, an octree

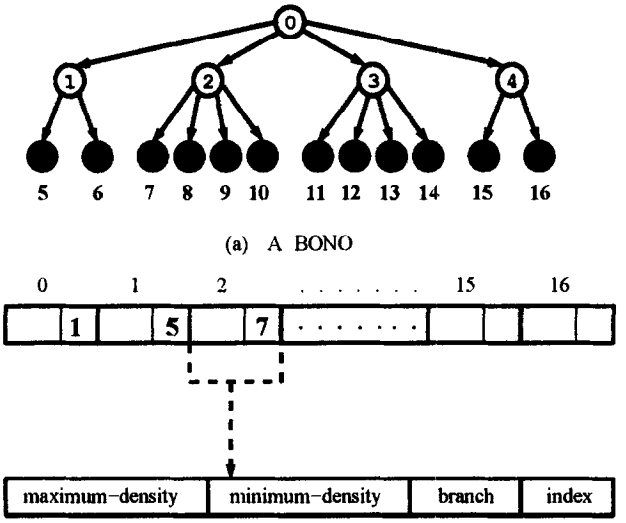



Fig. 1. A BONO and its linear BONO.

node is discarded if the threshold is less than its minimal value or larger than its maximum; otherwise, the subdivision for the octree node is replaced by a sequence of at most three bisections starting from the box represented by the current octree node. The bisection is performed in the $z$ direction, then in the $y$ direction, and finally in the $x$ direction. Whether a bisection in a specific direction is necessary is determined by the *branch* field of the octree node. The *branch* field of the internal octree node consists of three bits representing how the node is subdivided in the $z$, $y$, and $x$ direction. For example, *branch* = 111 means subdivisions in each of the $z$, $y$, and $x$ direction and *branch* = 101 means subdivisions in only the $z$ and $x$ direction. That is, the bisection order follows the order of the 1-bit in the *branch* field. After the bisection, the *branch* fields of subboxes will represent the remaining directions for further subdivision. For example, for *branch* = 101, the *branch* fields of the subboxes after subdivision in the $z$ direction are assigned to be 001. For a specific direction, the bisection is performed identically to the subdivision performed on the BONO in that direction. This is why the bisection tree is called a *branch-on-need bisection tree* (BONBT). Consequently, each subdivision of BONO is correctly replaced by at most three bisections.

Each node of the BONBT contains information on the minimum and the maximum coordinates of the corresponding box, the 12 Booleans for the MC property on the 12 edges, the 12 contour points, the 12 contour segments, and the four contour chains. Although many fields are needed in a node, the space requirement of the bisection tree is on the order of log $m$, where $m$ is the number of grid points, since at most one path of the tree is required to be in the memory.

3.3. *Generation of reduced triangular meshes*

For a given threshold, we traverse the BONO, discard the volume whose [*minimum-density*, *maximum-density*] does not contain the threshold, and substitute the subdivision of the node whose [*minimum-density*, *maximum-density*] contains the threshold with at most three branch-on-need bisections. For such a sequence of bisections, methods similar to the splitting-box algorithm [7] are employed to compute contour chains of boxes and examine whether the contour chains of two bisected boxes can be approximated by the contour chains of their parent box within a specific error $\delta$, and $\delta$ is the size of the cell. The bisections are performed recursively following the traversal of the BONO from top to bottom until a leaf node of the BONO is reached. Contour chains are then collected according to their legitimation of approximation during the bottom-to-top phases of the tree traversal.

Here we discuss the steps of mesh generation involved in the bisection process. While the BONO is traversed, an octree node $A$ is discarded if the threshold is less than its minimal value or larger than its maximum; otherwise, the subdivision for this octree node is replaced by a sequence of at most three branch-on-need bisections starting from the box $B$ represented by the current octree node. We first derive all contour chains on $B$ using marching cube computations. Each contour chain is associated with two fields, *valid* and *pred*. When $B$ is bisected into two subboxes, say $B_1$ and $B_2$, we verify whether $B_1$ and $B_2$ are MC. If any of $B_1$ and $B_2$ is not an MC box, the *valid* fields of all contour chains of $B$ are set to *false*. If one of $B_1$ and $B_2$ is MC and the other is not, we compute all contour chains of the MC subbox and continue to recursively examine the quality of the contour chains. If both $B_1$ and $B_2$ are MC boxes,

contour chains for both are derived and the contour chains of $B$ are examined to see if they approximate the contour chains of $B_1$ and $B_2$ within $\delta$. We examine every intersection between contour segments of $B$ and four edges on the common face of $B_1$ and $B_2$. If an intersection point lies between a pair of consecutive vertices of different signs, then this point is called a *feasible* point and the respective contour points of $B_1$ and $B_2$ are replaced by this point. After all the intersections are processed, we bisect each contour chain $C$ of $B$ into two subchains $C_1$ and $C_2$, where $C_1$ is on $B_1$ and $C_2$ is on $B_2$, and check to see if there exist contour chains $CB_1$ on $B_1$ and $CB_2$ on $B_2$, respectively, that coincide with $C_1$ and $C_2$. If such chains do exist, $CB_1$ and $CB_2$ are legitimately approximated by $C$ and the *valid* of $C$ and the *pred* of $CB_1$ and $CB_2$ are set to *true*. Otherwise, the *valid* of $C$ is set to *false*. After all contour chains of $B$ are examined, $B_1$ and $B_2$ are bisected and contour chains on $B_1$ and $B_2$ are examined recursively.

The above steps are performed recursively in a traversing order of the binary BONBT until a $2\times2\times2$ box is obtained. Contour chains are collected on the way up during the traversal of the bisection tree. A contour chain is output if its *valid* is *true* and *pred* is *false*, since there is no contour chain in the parent box that is able to approximate it within the error of $\delta$. Now we summarize the recursive procedure for the surface generation as follows: assume the box represented by the root of BONBT is $B$ and all contour chains of $B$ have been computed.

```
PROCEDURE LBONO-SB(B)
begin
bisect B into B1 and B2;
compute all feasible points on the common edges of
B1 and B2;
compute all contour points of B1 and B2;
replace the contour points by the corresponding
feasible points in B1 and B2
  if there is one;
if (not both B1 and B2 are MC) then
  for (each contour chain C of B)
    set the valid of C to false;
  Compute the contour chains of the MC box;
else begin /* if both B1 and B2 are MC */
  compute all contour chains in B1 and B2;
  for (each contour chain C in B)
    begin
    split C into C1 and C2;
    if (there exists CB1 in B1 and CB2 in B2 coincide
    with C1 and C2)
      set the valid of C and the pred of CB1 and CB2
      to true
    else
      set the valid of C and the pred of CB1 and CB2
      to false
    end
  end
if (B branches only in one direction) then
  /* B1 and B2 are the octree nodes */
  begin
  if (the corresponding octree node of B1 is empty)
  then set B1 to NULL
  else LBONO-SB(B1);
  if (the corresponding octree node of B2 is empty)
  then set B2 to NULL
  else LBONO-SB(B2);
  end
else /* B1 and B2 are not octree nodes */
  LBONO-SB(B1); LBONO-SB(B2);
for (each contour chain C in B)
  if (the valid of C is true and the pred of C is false)
    if (not both the sub_chains of C are valid) then
      output the valid sub_chain for rendering and
        set the pred value to true;
for (each contour chain C in B1 and B2)
  if (the valid of C is true and the pred of C is false)
    output C for rendering;
end
```

### 3.4. *An improved linear branch-on-need octree*

As described previously, each node of the branch-on-need bisection tree contains fields representing the minimum and the maximum coordinates of the corresponding box. The value of the *branch* field in a node of the BONO can be derived from the minimum and the maximum coordinates of the corresponding BONBT node, since the range of the volume represented by the tree node can be obtained from the minimum and maximum coordinates and the value of the *branch* can be derived from the range using the branch-on-need strategy. Hence the *branch* normally found on the internal node of the standard BONO can be deleted, with additional computations, of course. Note that the *branch* field is not necessary for leaf nodes. Moreover, the *address* field of leaf nodes can be deleted since it can be derived directly from the minimum coordinate found on the corresponding node in the BONBT.

To further reduce the space needed to store a BONO, we can partition the value of the density into, say eight intervals, and use a field *min-max* of 8-bits to represent the status of the data values for the node; see Ref. [11]. In the course of constructing a BONO, we assign the *min-max* of a node by setting the bits whose corresponding density intervals contain data values to 1 and the bits whose density intervals contain no data values to 0. For example, volume data with 8-bits for each vertex has densities ranging from 0 to 255. If the density range of [32, 128] is found to be more important for rendering, the density range 0–255 can be partitioned into 0–31, 32–47, 48–63, 64–79, 80–95, 96–111, 112–128, 129–255, each of which corresponds to bit $i$ of the field *min-max*, $i=0, \ldots, 7$. As an example, for a node containing data values ranging from 69 to 75 and from 100 to 110, the associated *min-max* is 00010100.

To check if the current BONO node contains the surface of a given threshold, we need two auxiliary flags, *value-1* and *value-2*, with the same length as *min-max* to encode the threshold. Bits in *value-1* and

*value-2* are initially set to 0. For the given threshold, we determine the range in which the threshold lies and set the corresponding bit in *value-1* and *value-2* to 1. Moreover, the bits on the right of this bit of *value-1* and those on the left of this bit of *value-2* are set to 1. Now the 1 bits of *value-1* represent ranges with values equal to or greater than the threshold and the 1 bits of *value-2* represent ranges with values equal to or less than the threshold. To decide whether the volume represented by the current node is of current interest, we need to perform the following two logical **and** operations:

$$r_1 = \textit{min-max}\ \textbf{and}\ \textit{value-1}$$
$$r_2 = \textit{min-max}\ \textbf{and}\ \textit{value-2}.$$

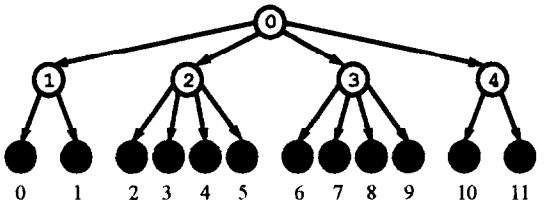
If $r_1 = 0$ or $r_2 = 0$ the volume of the current node contains no isosurface of the given threshold since all the data values in the volume are below or above the threshold. If both $r_1$ and $r_2$ are nonzero, the volume contains an isosurface of interest. The data fields needed in the improved BONO are the following: For the internal nodes, we need two fields, *min-max* and *address*. For leaf nodes, only *min-max* is required.

As described above, the improved BONO can be implemented as a linear BONO. Since the structure of leaf nodes is different from that of internal nodes, we now have a linear octree for internal nodes and an array for leaf nodes. The linear BONO for internal nodes is basically similar to the one described above except that the parent node of a leaf node now has an *index* referring to the index of that leaf node in the array for leaf nodes. So each record of the improved linear BONO for internal nodes contains a *min-max* and an *index* and each record of the array for leaf nodes contains only *min-max*.

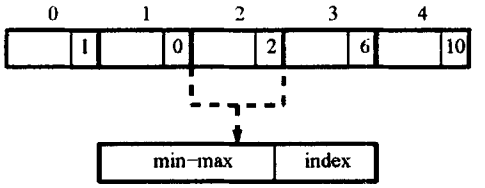
Figure 2 depicts the same BONO as that shown in Fig. 1, together with the corresponding improved linear octrees. The record for the leftmost child of node 0 is referred to by *index* of node 0, which is 1. The record for the third leftmost child of node 2 is indicated by adding 2 to the *index* of node 2, which is 4.

### 3.5. *Implementation and examples*

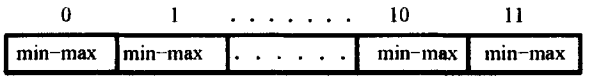
The proposed method has been implemented on an IRIS Indigo Entry with an R4000 CPU and 16MB of RAM. For comparison, three other isosurface generation algorithms, the marching cube algorithm (MCUBE) [1], BONO approach (BONO) [5], and Splitting-Box (SB) algorithm [7], were also implemented. The linear BONO data structure described above was implemented in the BONO approach and in the proposed method. In addition, the improved linear BONO structure described above was implemented in the proposed method. The proposed method using the linear BONO is denoted by LBONO-SB-1 and the proposed method using the improved linear BONO is denoted by LBONO-SB-2. As shown below, the improved linear BONO significantly reduces the memory space required with a much smaller additional cost. The current implementations of these four methods do not consider effective ways to reuse previously computed intersection points. For the BONO approach and the proposed method, the amount of time required for




(a) A BONO with internal nodes and leaf nodes numbered separately.




(b) Linear octree for internal nodes




(c) Array for leaf nodes

Fig. 2. An improved BONO and its linear octrees.

Table 2. The statistics for the head data with a threshold of 48

| Method | Time (s) | | | Triangle | | Space ratio (%) |
|---|---|---|---|---|---|---|
| | *S* | *G* | Ratio (%) | Number | Ratio (%) | |
| MCUBE | 0 | 305 | 100 | 419,753 | 100 | |
| BONO | 16 | 138 | 50.49 | 419,753 | 100 | 100 |
| SB | 0 | 456 | 149.51 | 166,002 | 39.5 | |
| LBONO-SB-1 | 16 | 167 | 60 | 139,905 | 33.3 | 100.13 |
| LBONO-SB-2 | 12 | 202 | 70.16 | 139,905 | 33.3 | 23.03 |

*S*: time for setting up BONO.
*G*: time for isosurface extraction.

Table 3. The statistics for the head data with a threshold of 80

| Method | Time (s) | | | Triangle | | Space ratio (%) |
|---|---|---|---|---|---|---|
| | *S* | *G* | Ratio (%) | Number | Ratio (%) | |
| MCUBE | 0 | 334 | 100 | 585,031 | 100 | |
| BONO | 16 | 173 | 56.59 | 585,031 | 100 | 100 |
| SB | 0 | 493 | 147.6 | 296,292 | 50.64 | |
| LBONO-SB-1 | 16 | 203 | 65.57 | 247,472 | 42.3 | 100.13 |
| LBONO-SB-2 | 12 | 234 | 73.65 | 247,472 | 42.3 | 23.03 |

*S*: time for setting up BONO.
*G*: time for isosurface extraction.

Table 4. The statistics for the brain data with a threshold of 45

| Method | Time (s) | | | Triangle | | Space ratio (%) |
|---|---|---|---|---|---|---|
| | *S* | *G* | Ratio (%) | Number | Ratio (%) | |
| MCUBE | 0 | 467 | 100 | 1,321,498 | 100 | |
| BONO | 19 | 233 | 53.96 | 1,321,498 | 100 | 100 |
| SB | 0 | 826 | 176.87 | 894,646 | 67.69 | |
| LBONO-SB-1 | 19 | 284 | 64.88 | 824,306 | 62.37 | 100.10 |
| LBONO-SB-2 | 15 | 317 | 71.09 | 824,306 | 62.37 | 23.03 |

*S*: time for setting up BONO.
*G*: time for isosurface extraction.

setting up BONO is denoted by $S$ and that needed for isosurface generation by $G$. The comparisons of speed and the number of triangles generated are made relative to the performance of MCUBE.

Two data sets were tested. The data values of each set take one byte for storage and range from 0 to 255. For the improved linear BONO, the range of the data values is partitioned into the following eight intervals: 0–31, 32–47, 48–63, 64–79, 80–95, 96–111, 112–128, 129–255. The first data set tested was volume data of a 3-D head with resolution of $256 \times 256 \times 80$. Table 2 shows the statistics obtained for the threshold of 48. The LBONO-SB-1 takes 60% of the time taken by MCUBE, which is about 9% more than that needed by BONO and about 89% less than that needed by SB. Moreover, LBONO-SB-1 and LBONO-SB-2 eliminate about 6% more triangles than SB. This may be because these two methods use the branch-on-need bisection strategy, rather than the even-subdivision strategy used in SB. One notable observation is that LBONO-SB-2 requires only 23.03% of the memory space needed by LBONO-SB-1 with about 10% additional computation cost. As described above, LBONO-SB-1 requires four bytes for the fields *branch* and *index* and two bytes for *minimum-density* and *maximum-density*. As shown above, LBONO-SB-2 needs four bytes for each internal node and only one byte for each leaf node. The BONO of the head data has 93,669 internal nodes and 655,360 leaf nodes. Consequently, a total of 4,494,174 bytes are needed for the linear BONO used in LBONO-SB-1 and 1,035,886 bytes are needed for the improved linear BONO in LBONO-SB-2. Note that the branch-on-need bisection tree has 23 levels and requires only $23 \times 130$ bytes, where 23 is the number of levels ($=\log_2(256 \times 256 \times 80)$) and 130 is the number of bytes necessary for a tree node. So the bisection tree requires only $(1 + 22 \times 2) \times 130$ bytes. Table 3 shows the statistics

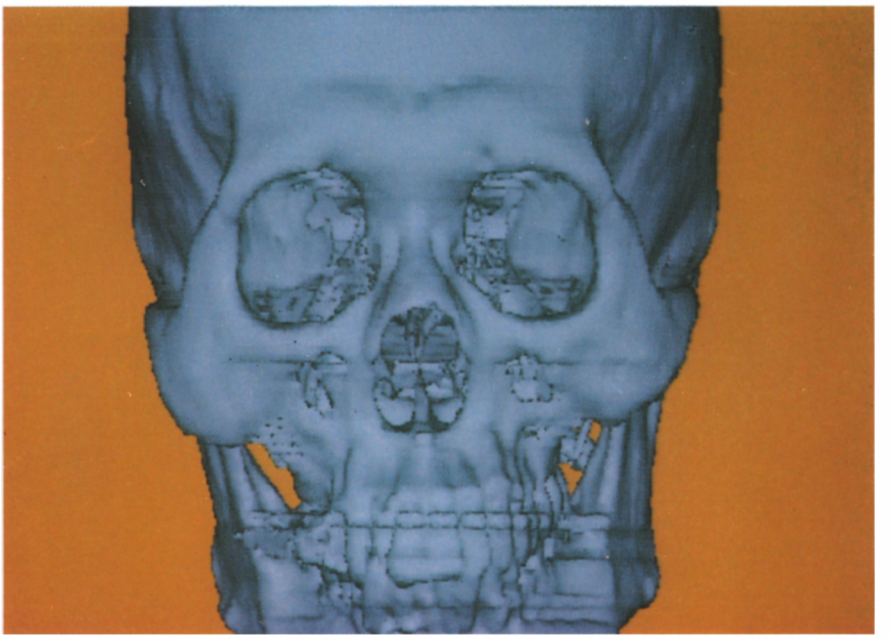


Fig. 3. Head image with a threshold of 80 produced by MCUBE and BONO.

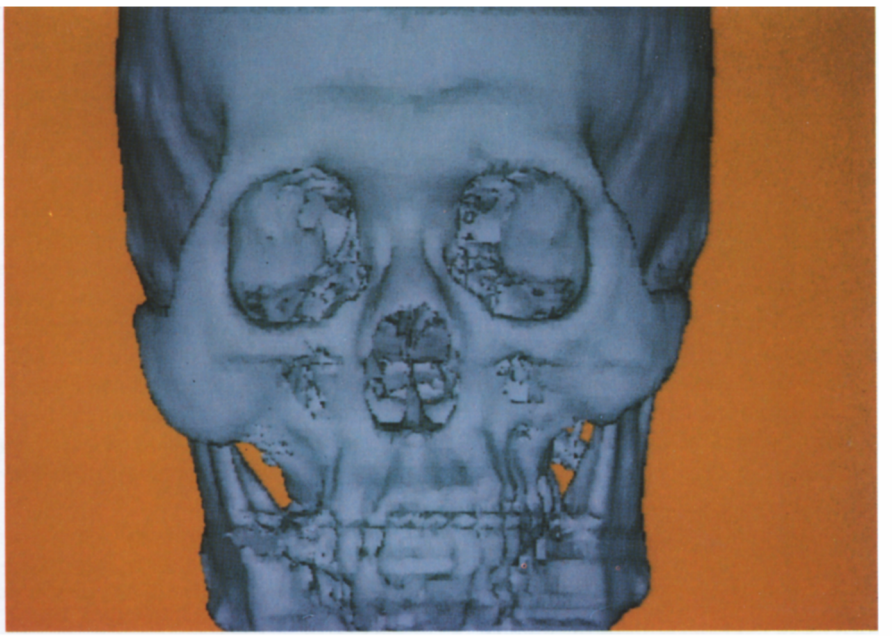


Fig. 4. Head image with a threshold of 80 produced by SB.

obtained for the threshold of 80; the results in Table 3 are similar to those in Table 2. The LBONO-SB-1 takes 65.57% of the time needed by MCUBE, which is about 9% more than that needed by BONO and about 82% less than that needed by SB. Moreover, LBONO-SB-1 and LBONO-SB-2 eliminate about 8.3% more triangles than SB and LBONO-SB-2 requires only 23.03% of the space needed by LBONO-SB-1 with about 8% additional computation cost. Table 4 shows statistics for the isosurface generation on a brain data, with a threshold of 45. The brain data has a resolution of $256 \times 256 \times 109$ with one byte for each item of data. The BONO for the brain data consists of 127,013 internal nodes and

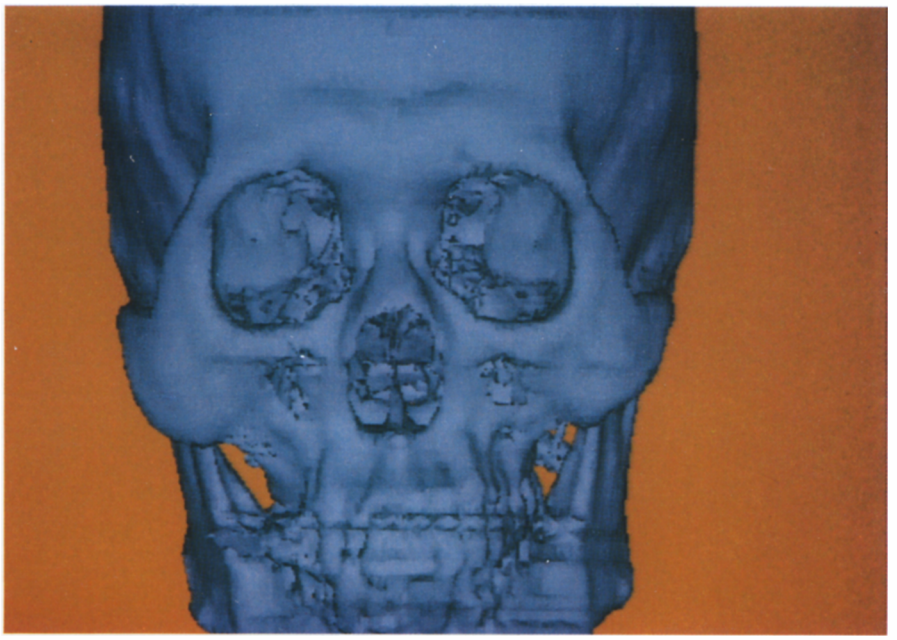


Fig. 5. Head image with a threshold of 80 produced by LBONO-SB-1 and LBONO-SB-2.

884,736 leaf nodes. A total of 6,070,496 bytes are required for the linear BONO used in LBONO-SB-1 and 1,398,638 bytes for the improved linear BONO. The branch-on-need bisection tree needs 5850 bytes, since the tree is of height 23 ($=\log_2(256 \times 256 \times 109)$). So LBONO-SB-1 and LBONO-SB-2 require 6,076,344 and 1,398,638 bytes, respectively, and LBONO-SB-2 requires only 23.03% of the space needed for LBONO-SB-1 with about 6% additional computation cost. The LBONO-SB-1 takes 64.88% of the time needed by MCUBE, which is about 11% more than that needed by BONO and about 112% less than that needed by SB. Moreover, LBONO-SB-1 and LBONO-SB-2 eliminate about 5.3% more triangles than SB. From the tested examples, we can make the following general observations:

- LBONO-SB-1 takes about 10% more time than BONO and reduces the number of triangles generated by 33% to 63%.
- LBONO-SB-2 requires only about 23% of the space needed by BONO and reduces the number of triangles by 33% to 63%, with about 14% to 22% additional computation cost.
- Both LBONO-SB-1 and LBONO-SB-2 not only consume much less time than SB but also produce fewer triangles.
- LBONO-SB-2 consumes only about 23% of the space needed by LBONO-SB-1 with little additional computation time.

Figures 3–5 are images of the head data with a threshold of 80.

## 4. CONCLUDING REMARKS

While the marching cube method for isosurface generation is simple and effective, it explores volumes that are of no current interest and produces an extremely large number of triangles. In this paper we have described a two-phase algorithm that resolves both of these problems. The algorithm employs a branch-on-need octree to avoid useless exploration of volume and reduces the number of generated triangles by adapting the size of the triangles to the surface's shape. Methods are also given that greatly reduce the amount of space required for the branch-on-need octree and generate identical results with little additional computational cost. The proposed algorithm is effective in speeding up the isosurface generation, in reducing the size of triangle meshes, and in decreasing the amount of memory required.

## REFERENCES

1. W. E. Lorensen and H. E. Cline, Marching cubes: a high resolution 3D surface construction algorithm. *Comp. Graph.* **22,** 163–169 (1987).
2. D. Meyers, S. Skinner, and K. Sloan, Surfaces from contours. *ACM Trans. Graph.* **11,** 228–258 (1992).
3. M. Levoy, Display of surfaces from volume data. *IEEE Comp. Graph. Appl.* **8,** 29–37 (1988).
4. L. Westover, Footprint evaluation for volume rendering. *Comp. Graph.* **24,** 367–376 (1990).
5. J. Wilhelms and A. Van Gelder, Octrees for faster isosurface generation. *ACM Trans. Graph.* **11,** 201–227 (1992).
6. H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, Mesh optimization. *Proc. SIGGRAPH '93,* 19–26 (1993).
7. H. Müller and M. Stark, Adaptive generation of surfaces in volume data. *The Visual Comp.* **9,** 182–199 (1993).

8. W. J. Schroeder, J. A. Zarge, and W. E. Lorensen, Decimation of triangle meshes. *Comp. Graph.* **26,** 65–70 (1992).
9. G. M. Neilson and B. Hamann, The asymptotic decider: Resolving the ambiguity in marching cubes. *Proc. Visualization '91*, 83–90 (1991).
10. J. Wilhelms and A. Van Gelder, Topological consideration in isosurface generation. *Comp. Graphics*, **24,** 79–86 (1990).
11. C.-C. Lin, A fast volume rendering algorithm. Master's thesis, National Chiao Tung University (1993).