PAPER

A "Group Marching Cube" (GMC) Algorithm for Speeding up the Marching Cube Algorithm

Lih-Shyang CHEN[†a)], ***Nonmember***, **Young-Jinn LAY**[†], ***Member***, **Je-Bin HUANG**[†], **Yan-De CHEN**[†], **Ku-Yaw CHANG**[††], ***and*** **Shao-Jer CHEN**[†††], ***Nonmembers***

SUMMARY Although the Marching Cube (MC) algorithm is very popular for displaying images of voxel-based objects, its slow surface extraction process is usually considered to be one of its major disadvantages. It was pointed out that for the original MC algorithm, we can limit vertex calculations to once per vertex to speed up the surface extraction process, however, it did not mention how this process could be done efficiently. Neither was the reuse of these MC vertices looked into seriously in the literature. In this paper, we propose a "Group Marching Cube" (GMC) algorithm, to reduce the time needed for the vertex identification process, which is part of the surface extraction process. Since most of the triangle-vertices of an iso-surface are shared by many MC triangles, the vertex identification process can avoid the duplication of the vertices in the vertex array of the resultant triangle data. The MC algorithm is usually done through a hash table mechanism proposed in the literature and used by many software systems. Our proposed GMC algorithm considers a group of voxels simultaneously for the application of the MC algorithm to explore interesting features of the original MC algorithm that have not been discussed in the literature. Based on our experiments, for an object with more than 1 million vertices, the GMC algorithm is 3 to more than 10 times faster than the algorithm using a hash table. Another significant advantage of GMC is its compatibility with other algorithms that accelerate the MC algorithm. Together, the overall performance of the original MC algorithm is promoted even further.

key words: *marching cube algorithm, iso-surface detection, surface rendering, speed-up, interactive applications*

1. Introduction

1.1 Voxel-Based Object Representation

Volume visualization is a method of extracting meaningful information from volumetric datasets through the use of interactive graphics and imaging, and is concerned with the representation, manipulation, and rendering techniques of volumetric datasets [1], [2]. These techniques consider the object space as a whole, divide it up into regular or cubic voxels, and label each voxel in the space according to object occupancy. This type of object representation is usually referred to as voxel-based object representation. Currently, there are many practical applications of voxel-based object representation. For instance, in medical image applications, many researchers have used voxel-based object representation to stack together a series of cross-sectional 2D medical images, such as magnetic resonance imaging (MRI), Computed tomography (CT) [4], and the like, to provide computer-generated 3D images of the 3D structures to be explored.

1.2 Needs for the GMC Algorithm

The MC algorithm [3] is one of the most popular methods for displaying a voxel-based object. However, its slow surface extraction process is usually considered one of its major disadvantages. The algorithm was proposed in 1987 and most of the papers regarding were published prior to 2000. Since then, there are not many contributions to the MC algorithm seen in the literature. However, the advent of novel image acquisition techniques which has lead to very large voxel datasets has made the need for MC speed-up techniques grow dramatically and therefore interest in related subject has increased dramatically.

In order to efficiently store and render the graphics primitives (e.g. triangles) modern graphics libraries, such as OpenGL (Open Graphics Library) or Direct3D, provide facilities called vertex arrays and index arrays to store the triangle data in their implementation. The vertex arrays allow triangle vertices and their attributes, such as vertex positions, colors, or normals, to be specified in arrays while the index arrays allow its integer indices to index the vertices in the vertex array to form triangles as shown in Fig. 1. The five triangle vertices are represented in the vertex array as [v1, v2, v3, v4, v5], whereas the four triangles are represented in the index array as [1 2 5, 2 3 5, 1 5 4, 4 5 3]. In this case, the data of each vertex can be shared by many triangles. As a result, the vertex and index array can save a considerable amount of memory space and computation time. We will further discuss this in Sect. 2.3 of this paper.

There are many different aspects to improve the computation time of the MC algorithm, such as (1) avoiding traversal of the inactive cubes [5]–[7] and (2) speeding up the vertex-identification process, since most of the triangle-vertices of an iso-surface are shared by several triangles [8]. The vertex-identification process is needed to avoid the duplication of the vertices in the vertex array of the resultant triangle data. It was pointed out that for the original MC algorithm, we can limit vertex calculations to once per

Manuscript received October 22, 2010.
Manuscript revised February 17, 2011.

[†]The authors are with the Department of Electrical Engineering, National Cheng Kung University, No. 1, University Rd., Tainan 701, Taiwan.
[††]The author is with the Department of Computer Science and Information Engineering, Da-Yeh University, Changhua County, Taiwan.
[†††]The author is with the Buddhist Tzu Chi General Hospital, Chiayi County, Taiwan.
a) E-mail: chens@mail.ncku.edu.tw

DOI: 10.1587/transinf.E94.D.1289

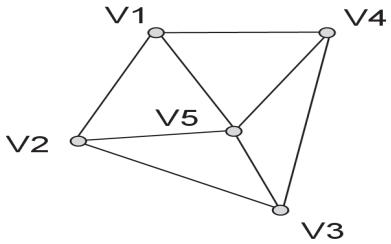


Fig. 1 Five vertices and four triangles.

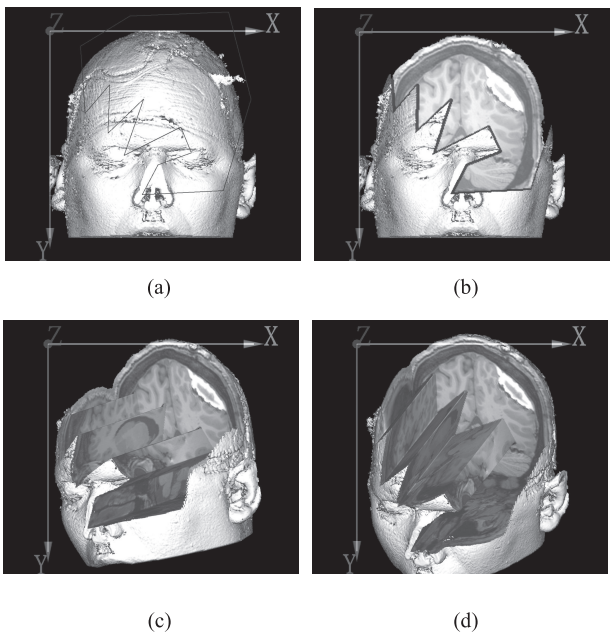


Fig. 2 (a) The manipulation contour drawn by the user on the screen. (b) After the portion within the manipulation contour is removed, the resultant image shows the internal structure of the object. (c) and (d) The object is rotated to another viewpoint.

vertex to speed up the surface extraction process and in the meantime, the reuse of these MC vertices is easy to understand. However, how this process could be done efficiently was not mentioned and was not looked into seriously in the literature.

In some applications, the algorithms avoiding traversal of the inactive cubes may not be applicable. For instance, when an object is manipulated interactively by the users and the surfaces of the object need to be re-computed for each manipulation, such as cutting the object by a contour with a specified depth as shown in Fig. 2, there are no "inactive cubes" in the application contexts.

Therefore, in this paper, we do not consider the first aspect at all since it has been fully investigated in the literature previously and may not be applicable in some cases. Instead, we focus on the second aspect by proposing the "Group MC" algorithm, or GMC for short. The GMC considers a cube and its neighboring cubes as a whole instead of considering an individual voxel alone; it takes advantage of the spatial coherence of 3D objects; and it explores many interesting features of the original MC algorithm that were not considered in the past. Based on the GMC, we invent a very important concept of "new-voxel vertex" that improves the efficiency of the MC algorithm. In other words, although the strategies of vertex reuse have already been used in MC implementations, the value of this paper's contribution lies in its description of a straightforward algorithm which is detailed enough to allow its reimplementation and usage. Therefore, we believe that the GMC has not only theoretical, but also practical value.

1.3 The Organization of This Paper

The rest of the paper is organized as follow. In Sect. 2, we discuss the relevant background materials including the problems with a hash table searching algorithm and the timing comparisons of different MC algorithms which justified the need of the GMC algorithm. In particular, based on our discussions with many people in the community, most think that all the research topics of the MC algorithm have been done more than a decade ago already and there is no need to further study it. Therefore, we also show the experiment results in this section (instead of at the end of the paper as most of the paper do) to explain the effectiveness of the GMC algorithm and to motivate the study in this paper. In Sect. 3, we introduce the important concept of the "new-voxel vertex" and how to use it to efficiently do the vertex identification process. In Sect. 4, we discuss the additional memory needed by the GMC algorithm. There are many existing acceleration MC algorithms proposed in the literature. Most of those algorithms can be incorporated with our GMC algorithm. In other words, the GMC algorithm is complementary to those algorithms rather than in competition with them. We also discuss how to combine the GMC algorithm with other existing MC acceleration algorithms. In Sect. 5, we present our conclusions.

2. Relevant Background Materials

2.1 The Slow Surface Extraction Process of the MC Algorithm

For the sake of argument, we will simply use "cube" to refer to the "cube" in the context of the MC algorithm. The MC triangle configurations within each cube consider only the 8 voxels at the 8 corners of the cube and nothing else. This concept makes the MC algorithm extremely simple to understand and therefore easy to implement. However, it does not attempt to take advantage of the spatial coherence of objects in 3D space. As a result, the slow surface extraction is usually considered as one of the major disadvantages of the MC algorithm.

The implementation of the MC algorithm consists of three steps: (1) compute the case type of each cube, (2) compute the vertices of each cube based on its case type, assign the vertex indices, and output the data of each vertex to the vertex array, and (3) compute the indices of the triangles of

each cube based on its case type and output them to the triangle index array. The vertex and index array can be more efficiently reused, stored, and rendered by OpenGL or Direct3D functions. In general, the first and second steps take up most of the execution time of the MC algorithm. The third step just simply outputs the results of the second step.

In the second step, we need a vertex identification process to check whether a vertex has been generated before or not. If it has, its index can be re-used for the current triangle. If it has not, it has to be generated and assigned a new index. In general, it may take a lot of time to search through the existing vertex array to find the answer. This search process is an important factor that invariably slows the MC algorithm and will be shown in Sect. 2.3.

2.2 The Problems with a Hash Table

The search process can be improved by a hash table with reasonable performance in some cases. However, the hash table has its own problems. For instance, it is difficult to estimate how large the hash table size should be in the context of the MC algorithm since the number of vertices of the objects we are interested is usually unpredictable. If the hash table is too full, the performance of the hash table may deteriorate to linear (i.e., O(N) instead of a constant time where N is the number of entries of the hash table) for each search due to unavoidable collisions. If the hash table is too empty, a lot of memory is wasted. It is also computationally expensive to expand the hash table if we find the table is too full at run time. Once the hash table is established, the expansion at run time involves modification of the hash function, creation of a larger table, and rehashing all items in the old table into the new one. Furthermore, although each search without a collision takes a constant time that is much shorter than a brute force search, the constant time is certainly not a very short one since it involves the computation of the hash function and various overheads.

2.3 The Effects of Hash Table on the Overall Performance of the MC Algorithm

In order to investigate how much the hash table affects the overall performance of the MC algorithm, we performed the following experiments. We discuss the timing results of various cases here instead of discuss them right before the conclusion section of the paper as most papers do. In particular, based on our experience of discussions with many people in the community, most people think that all the research topics of the MC algorithm have been done more than a decade ago already and there is no need to further study it. The experiment results shown here can also explain "why not simply just use the hash table to solve the problem" as most people suggested. The results also motivate further study in this paper. The well-known publicly available VTK (Visualization Took Kit) program [5] is used in the experiments. The VTK does not use an algorithm to avoid the traversal of inactive cubes and it uses a hash table to support the search process.

Table 1 The timing performance of different cases measured in seconds. (For an object with more than 1 million vertices, the GMC algorithm is 3 to more than 10 times faster.)

Object	Dataset size	Number of vertices	Case 1	Case 2 (2G)	Case 3 (2G)	Case 4 (4G)	Case 5 (4G)
A	512X512X50	1,360,742	0.8	4.5	1.3	3.1	1.3
B	512X512X100	3,655,099	1.6	15.5	2.5	11.0	2.5
C	512X512X200	9,325,839	3.2	54.0	5.1	27.0	5.1
D	512X512X355	15,145,631	5.7	200.4	9.0	51.0	9.0
E	512X512X400	9,418,833	6.4	80.6	10.1	40.4	9.9
F	512X512X585	12,800,214	9.3	181.0	15.1	46.0	15.0
G	512X512X50	2,677	0.8	1.3	1.2	1.3	1.2
H	512X512X100	2,665	1.6	2.5	2.4	2.5	2.4
I	512X512X200	2,689	3.2	5.0	4.9	5.0	4.9
J	512X512X355	2,521	5.7	8.8	8.7	8.9	8.7
K	512X512X400	2,772	6.4	10.0	9.8	10.0	9.8
L	512X512X585	3,181	9.3	14.6	14.4	14.6	14.4
M	512X512X50	22,673	0.8	1.3	1.3	1.3	1.2
N	512X512X100	61,693	1.6	2.7	2.4	3.1	2.4
O	512X512X200	271,800	3.2	7.6	4.9	6.6	4.9
P	512X512X355	651,503	5.7	21.1	8.8	16.9	8.8
Q	512X512X400	85,445	6.4	17.3	9.8	12.0	9.8
R	512X512X585	142,547	9.3	36.8	14.5	25.6	14.5

This may be due to the fact that VTK tries to deal with more general cases instead of the regular grid problems we discuss here. The programs were developed by experts in the visualization field and the codes are reasonably efficient. Since VTK was developed originally for general-purpose usages, we modified its codes to speed up about 10% of the original performance specifically for this study by eliminating some general cases. We have tested more than 50 different objects for the algorithms described here and obtained similar results consistently. We give only some examples and their timings here to show how the algorithms perform in Table 1. Although the experiment results may be different from machine to machine depending on the hardware configurations, such as memory size, cache size, CPU speed, bus bandwidth, and so on, this table gives readers a rough idea of the computation time for different situations.

All the experiments use the modified VTK program codes. Case 1 is the execution time of the MC algorithm for a threshold value high enough so that the case types of all the cubes are 0s based on the case type definition of the MC algorithm. In other words, no MC vertex is generated. The time measured is similar to that of traversing all the cubes only. This is the best case in terms of execution time since no vertex is generated. Case 2 is the execution time to generate some MC triangles with a hash table algorithm for the search process. Case 3 is the same as case 2 except that it does not use the search process (without the hash table mechanism). Instead, a table look-up proposed in this paper, i.e., the GMC algorithm, is used for the vertex identification. The computer system used for the experiments had an Intel Pentium 4 Processor 631+ CPU with

3.0 GHz clock, 2 MB cache memory, 2 GB main memory, and a Microsoft Windows XP operating system. Case 4 and Case 5 are the same as Case 2 and Case 3 respectively except that the machine had an Intel Pentium Dual-Core Processor T3200 CPU with 2.0 GHz clock, 1 MB cache memory, 4 GB main memory. The program is a single-thread process. In other words, only one core was used.

In case 1, since the program simply goes through all the 3D cubes to compute the case types, the measured times are almost linearly proportional to the sizes of the corresponding 3D spaces in question, as we expected. However, in cases 2 and 3, since the situations are more complicated, the measured times are just more or less proportional to the numbers of generated MC vertices and the sizes of the corresponding 3D spaces.

Please note that the execution times in Table 1 are not linearly proportional to the numbers of vertices since the marching cube types of the cubes in the 3D scene are different from object to object and different types need different amounts of processing time. When the number of vertices is small, case 2 and case 3, and case 4 and case 5 are almost the same since the hash table algorithm performs very well. However, as the number of vertices increases, cases 3 and 5 required much less time to execute the MC algorithm. The hash table seems to take up a significant amount of time when the number of vertices is particularly large. This is in fact what we originally expected due to the problems described above. When the hash table is nearly full, the hash table algorithm becomes an O(N) algorithm or needs a computationally expensive re-hash function instead of a constant-time algorithm. As seen in the cases 2 and 4 on Table 1, the larger the memory is, the better the hash table algorithm performs.

Itoh et al. [8] proposed a propagation-based algorithm, called a cell-edge centered propagation algorithm, which does not use a vertex search algorithm for the vertex-identification process in the MC triangle generation. The algorithm visits all cells sharing an iso-surface cell-edge at the same time, and the vertex that lies on the cell-edge is registered onto all the triangles inside the visited cells. The vertex is no longer required in this process, and therefore the vertex search algorithm is not necessary in the algorithm. Their experimental tests showed that the method was about 20 percent faster than the conventional propagation implementation. However, the algorithm still needs to decide whether a given cell has been inserted in an FIFO (First In First Out buffer) and whether the triangles in a cell have been constructed or not. These operations described in the pseudo-code of their implementation still need some search processes although the number of search processes may be reduced. The search processes may be implemented by hash tables. We believe that this is why only 20 percent speed-up was gained. Based on our experiments, if the search processes were completely eliminated, there would be more speed obtained.

One can also avoid the search process by simply duplicating the vertices in the vertex array. In this case, the program may produce about 6 times more MC vertices on average based on our experiments with the MC algorithm. In other words, the benefit we gain from the vertex identification process is to reduce the number of vertices to 1/6 of the number of vertices if we simply duplicate the vertices. The number of vertices will affect the time for CPU to compute the normal and other various feature values (such as texture coordinates and colors), the space for storing the data of vertices, the time to send the vertices from CPU to GPU for rendering, the time for rendering, e.g., a higher hit ratio for the vertex cache in GPU when the transformed vertices are reused, and so on. The costs for the search process and vertex duplication may vary from object to object depending on the size and complexity of the object in question. Furthermore, as the CPU and GPU technologies continue to progress, it may be hard to argue whether the search process is justifiable. In any case, it is always favorable if the search process can be eliminated and the duplication of the vertices can be avoided. In particular, scientific research should not be limited to the current technology. It is always both practically and theoretically interesting to eliminate the search process while avoiding the vertex duplication.

In this paper, we propose the concept of the "new-voxel vertex" to completely eliminate the search process and avoid the duplication of vertices so that we can efficiently generate MC triangles. The details are described in Sect. 3.3.

2.4 Discussion of the Timing Performance of the MC Algorithms

In Table 1, when the number of vertices is small, more than 65% of the time is spent on the traversal of all the cubes to compute the case types of the cubes alone if we compare case 1 (the time mostly for the case type computation) and case 2 (the total time for the MC algorithm). In other words, it is not surprising that those algorithms that avoid the traversal of the inactive cubes can achieve 10 or even 100 times of improvement in performance if the number of active cubes (or the number of vertices) is small. Those algorithms do make a significant contribution in this case. However, if the number of the vertices is very large, the time spent for generating the vertices becomes a dominant performance factor in the whole MC algorithm as shown in Table 1. In this case, GMC is much faster than the original one.

Furthermore, in many interactive visualization applications where the objects of interest may be inside another object or blocked by other objects in the 3D space, or in the case when users may want to visualize the internal structure of objects of interest, some manipulations, such as cut-away operations, may be needed to reveal the objects of interest as shown in Fig. 2 that we have implemented in this study. In other words, the definition of the objects of interest can not be done by a simple threshold operation. Various automatic or semi-automatic volume manipulation operations are needed in order to reveal the objects of interest for visualization purposes. In this case, since the initial objects in the 3D space have been defined, the manipulation

Table 2 The timing performance of object surface detections and object manipulations measured in seconds.

Dataset size	Number of vertices	GMC algorithm Execution time	Manipulation time
256*256*283	675,490	0.97	0.045
512*512*250	969,930	1.59	0.073
512*512*256	1,073,790	1.62	0.067
205*205*102	169,790	0.39	0.013

operations are usually done on a volume-representation data structure [4], [9]–[12] rather than a surface-representation data structure alone, such as MC triangles, that does not contain the internal structure of the objects. In this case, a "new object" (the resultant object) is generated after the manipulation. If the MC algorithm is used to detect the surfaces of the "new object", the traversal of all the cubes in the 3D space is not needed since the cubes of the object have been determined. In this case, those algorithms, such as interval-based algorithms, may not be very helpful in speeding up the MC algorithm. In this type of user interactions, near real-time performance is critical to the usability of a visualization system. In our experiments, the manipulation time (the time for the program to remove unwanted parts from the 3D scene) can be ignored as shown in Table 2 based on the current software technology. The MC triangle generation for the display of the new object becomes the bottleneck of the operations. In many other visualization applications [13], [14], the size of the original data set and the objects of interest are even larger than those listed in Table 1. For many medical applications, the objects usually have more than 1 million triangles and it is not appropriate to use a lower resolution to extract the triangle surfaces. In particular, when the objects are manipulated, the execution time of generating the new object surfaces is extremely critical to the user interactions. In this case, the GMC algorithm can be used to improve the performance of the MC algorithm.

3. The Surface Extraction Algorithm

3.1 Pre-Processing

Generally speaking, most of the images need some kind of pre-processing before they can be used for 3D object reconstruction. Therefore, in the pre-processing, without loss of generality, we can add an extra layer of 0's around the 3D scene to ensure that all the voxels of the objects of interest will not be on the boundary of the overall 3D scene.

3.2 Introduction to the Concept of the GMC

In the context of the MC algorithm, a "cube" has 8 corners each of which is a voxel in the voxel-based object representation. Other than this relationship, the voxels and cubes have nothing to do with each other. However, if we think about the definition of the "cube" from a different perspective, there is a duality between the cubes and voxels. For simplicity, let us consider a one-dimensional case as

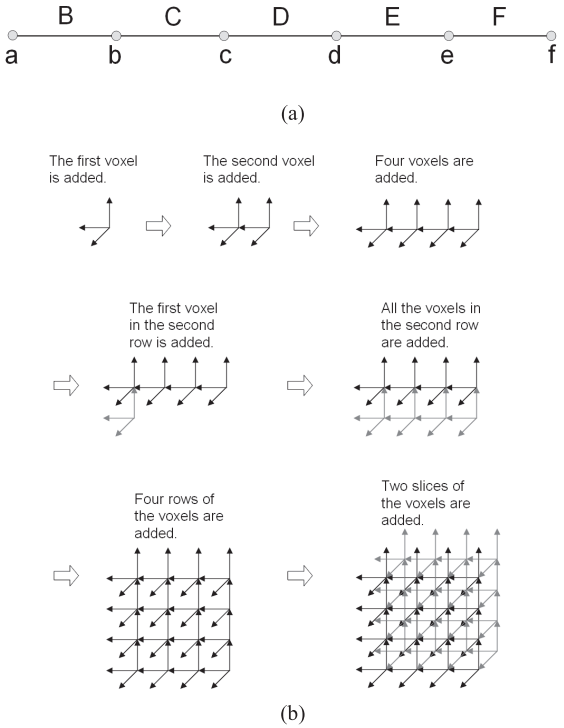


Fig. 3 (a) The relationship between the intervals and points. (b) The new-voxel edges form the whole 3D scene.

shown in Fig. 3 (a). There are six points and five intervals on a line. If we ignore the boundary point "a" in Fig. 3 (a), there is a one-to-one correspondence between the intervals and points: interval B to point b, interval C to point c, and so on. As described in the pre-processing, we add an extra layer of 0s around the original 3D scene. If we ignore the first boundary voxel in each row of cubes, there is a one-to-one correspondence between the "voxel world" and the "cube world" in the 3D space. In other words, there is a duality between the voxels and the cubes. Each MC cube in the 3D scene has its own corresponding voxel.

We assume that the cubes are scanned in a row-by-row, slice-by-slice fashion. We can imagine that the original 3D scene is empty and there is no voxel in the scene. When we scan through each cube to generate its vertices and triangles based on the case types, we can imagine that its corresponding voxel is added into the 3D scene. For instance, in Fig. 4, when the cube is considered, its corresponding "new voxel" v6 is added into the scene. We assume that we scan the 3D scene from left-to-right, top-to-bottom, and front-to-back. The "new voxel" V6, together with the 7 existing neighboring voxels that have been added into the 3D scene, form the "current MC cube". Furthermore, three new edges x, y, and z in Fig. 3 (b) are also added into the scene and are referred to as "new-voxel edges" (imagine that a "new voxel" and three "new edges" are added into the scene, together with the existing voxels and edges in the scene, to form a "new cube"). Three potential new vertices located on the three new-voxel edges are referred to as "new-voxel vertices" (one vertex for each dimension) if they exist.

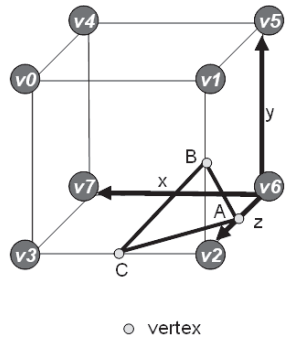


Fig. 4 The arrangement of weights of voxels for the case type values.

In other words, each cube also has three corresponding new-voxel edges and new-voxel vertices. Therefore, as we scan through each cube, its corresponding new-voxel edges are built up as shown in Fig. 3 (b) in sequence. Eventually all these new-voxel edges form the overall 3D scene. Please note that the arrows in Fig. 3 (b) are for explanatory purposes only and are not the coordinate axis.

For each case type of an MC cube, its corresponding new-voxel vertices can be pre-computed and stored in a look-up table called "new-voxel-vertex table". With this concept in mind, there are two different types of MC vertices defined in an MC cube: (1) new-voxel vertices that are the only vertices newly generated for the cube and (2) the rest of the vertices that have been generated when the neighboring cubes of the current cube were considered previously. Therefore, the search process, which was discussed in Sect. 2 and used to decide whether a vertex in a cube was generated previously, is not necessary. This will significantly speed up the surface extraction time. For instance, in Fig. 4, the v6 is the now-voxel for the cube. If v2 is the only voxel that is inside the object, the case type is 4 and vertices A, B, and C form a triangle. Through the new-voxel-vertex table, without the need for a search process, we know vertex A is the only new-voxel vertex that needs to be generated and assigned a new vertex index, while vertices B and C were generated in the previous cube we scanned before. To generate the triangle indices, we need to know the indices of vertices B and C, i.e., the vertex identification process.

Therefore, we need to set up two "layers" of cubes corresponding to the previous and current slices of cubes shown in Fig. 3 (b), called a "vertex-index" array to store the vertex indices that have been assigned to vertices whenever the vertices are generated. Each element of the "vertex-index" array represents a cube and has three entries (instead of 12 entries) to store the indices of its three new-voxel vertices in each cube, if such vertices do exist. Therefore, we can obtain the indices of vertices B and C from this array and produce the triangle indices. For the given new-voxel, knowing how and where to obtain the entries of vertices B and C in the array and the indices of the vertices B and C efficiently will be explained in the next sub-section.

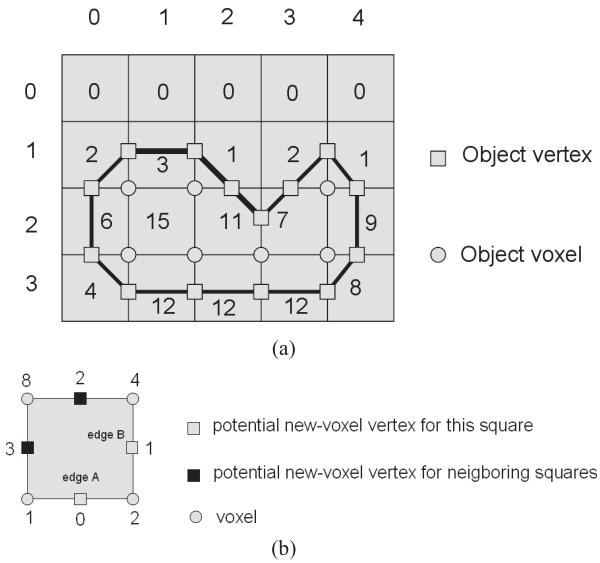


Fig. 5 (a) A simple 2D example. (b) The local vertex name assigned to each vertex and the value assigned to each voxel for the case type computation.

3.3 A Simple 2D Example

Our algorithm works for 3D objects. The experiment results are shown in Table 1 described in 2.3 previously. However, since the 3D case is harder to draw in a 2D figure, we simply go through a 2D example as shown in Fig. 5 (a) to explain how the algorithm really works. The same mechanism can be directly applied to a 3D case without any new complication since only an additional dimension is needed.

In a 2D case, the object space is represented as an array square[5, 4], each element of which is a square (equivalent to a cube for a 3D case in the MC algorithm). The number inside each square is its case type, which is computed based on the setting shown in Fig. 5 (b). The value around each voxel is used to compute the case type if the voxel is inside an object while the value around each vertex is the "local vertex name" assigned to the vertex and has only local significance for a given square. For instance, the case type of square$[2, 0] = 2 + 4 = 6$ and the case type of square$[2, 2] = 1 + 2 + 8 = 11$ where 6 and 11 are the MC case types. Each square can have two new-voxel vertices with the local vertex names 0 and 1 in Fig. 5 (b). We say that the new-voxel vertices 0 and 1 belong to the square. After all case types are computed, the object space is scanned on a row-by-row basis to generate the vertex and index arrays for MC triangles.

At this point, our goal is to efficiently generate the vertex and index array for the squares with their case types so that we can store all the triangle data and use the OpenGL indexed vertex rendering function to display the 3D objects as discussed in 2.3. When a square with its case type is examined to generate the vertices and indices of its triangles, we have to do two-step computations: (1) find out which

Table 3 (a) The edge table. (b) The new-voxel-vertex table. (c) The square-offset tables. (d) The vertex-offset tables.

Case type	Edge
0	
1	(0, 3)
2	(0, 1)
3	(1, 3)
4	(1, 2)
5	(0, 3) (1, 2)
6	(0, 2)
7	(2, 3)
8	(2, 3)
9	(0, 2)
10	(0, 3) (1, 2)
11	(1, 2)
12	(1, 3)
13	(0, 1)
14	(0, 3)
15	

(a)

Case type	New-voxel vertex
0	
1	0
2	0, 1
3	1
4	1
5	0, 1
6	0
7	
8	
9	0
10	0, 1
11	1
12	1
13	0, 1
14	0
15	

(b)

Local vertex name	Offset
0	0
1	0
2	L
3	1

(c)

Local vertex name	Offset
0	0
1	1
2	0
3	1

(d)

new-voxel vertices should be generated for this square and generate them accordingly. The new-voxel vertices are the only new vertices needed for the given square. (2) generate the vertex indices of all the edges (or triangles for a 3D case) in this square (or cubes for a 3D case). We are going to discuss these two steps in 3.3.1 and 3.3.2 respectively.

3.3.1 The Generation of New-Voxel Vertices for a Square

In the original MC algorithm, an edge table "edge[case type]" is used to generate the vertices and triangle edges for a given case type. The contents of the edge table are shown in Table 3 (a). For instance, "edge[2] = {0, 1}" means that a square with a case type 2 has a triangle edge with two end vertices 0 and 1 (where 0 and 1 are the "local vertex names" defined in Fig. 5 (b)). Please note that the contents of edge[case type] table are the "local vertex names" (local to a square or a cube for 2D or 3D cases). In the 3D case, this "edge[case type]" becomes "triangle[case type]" that stores the local vertex names for all the triangles of the given case type so that the MC algorithm can generate all the triangles for the given case type.

We use the table "new-voxel-vertex[case type]" to store the new-voxel vertices for a given case type shown in Table 3 (b). For instance, the first edge detected is in square[0, 1] with a case type 2. "new-voxel-vertex[2] = {0, 1}" means that a square with a case type 2 has two new-voxel vertices: 0 and 1 (the local vertex names) as shown in Fig. 5 (b). These two vertices have to be created and assigned vertex indices for the corresponding square. The "new-voxel-vertex[3] = {1}" means that a square with a case type 3 has only one new-voxel vertex 1. Please note that the new-voxel vertices are the only new vertices generated for the current square. These new-voxel vertices will be generated in the vertex array and assigned some new vertex indices which will be stored in the vertex-index array described at the end of 3.2 for future references.

3.3.2 The Generation of the Vertex Indices for All the Edges (or Triangles for a 3D Case)

Here we discuss the generation of the vertex indices for all the edges (or triangles for a 3D case) in the corresponding square (or cube for a 3D case) in the vertex-index array where the indices of all the existing vertices are stored. Notice that these vertices are any vertices other than the new-voxel vertices we computed in the first step. To store the vertex indices of all existing vertices, we use an array "vertex-index[5x4, 2]", since the dimension of the 2D array is $m \times n$ where $m = 5$ and $n = 4$ in this case, and the size of the vertex index array is $m \times n$). The first array index represents a linear array for the 5×4 object space shown in Fig. 5 (a). In other words, we use a linear array to represent the 2D space. A linear pointer LP is used to point to the current square under process in the vertex-index array. In this example, the first square that has vertices is square[1, 0]. Therefore, LP = 5 since the row-wise array index is used and LP = $m \times 1 + 0$ where $m = 5$. Each square can have at most two new-voxel vertices. Therefore, the second index of the vertex-index array has values 0 or 1 to store indices of the two new-voxel vertices belonging to each square respectively if they exist. In other words, the first array index is a "global address" of a cube and the second one indicates which new-voxel vertex is stored. Therefore, we have vertex-index[5, 0] = 0 and vertex-index[5, 1] = 1 for the current square (please note that the array index starts with 0 based on the C++ convention); the 0 and 1 are the vertex indices assigned to the two new-voxel vertices respectively. The data of these two new vertices, such as coordinates, are output to the vertex array for OpenGL rendering later.

The next step is to generate the edge indices and output them to the edge index array (triangle index array for the 3D case so that an OpenGL function can render the 3D object). We know that the indices of the previously created vertices are stored in the vertex-index array. The question is where they are in the vertex-index array described previously. In other words, we have to start with the local vertex names 0, 1, 2, or 3 obtained from the edge[case type] array (the triangle[case type] for the 3D case), and find the global addresses of the two vertex indices in the vertex-index array. Another two tables: "square-offset[local vertex name]" and "vertex-offset[local vertex name]", can be set up to help us compute where the indices of the vertices are. The contents

of these two tables are shown in Table 3 (c) and 3 (d) respectively. We know currently LP = 5. For a given local vertex name, the square-offset table tells us the offsets of the squares the vertex belongs to while the vertex-offset tells us the offsets of the vertex locations in the square the vertex belongs to.

For instance, "square-offset[2] = −L" means that a vertex with a local vertex name "2" belongs to the top square (relative to the current square as shown in Fig. 5 (b)) and the offset of the global address of that square from the current square is −L where L is the width of the object space (since we use a linear array to represent this 2D object space). Likewise, "square-offset[3] = −1" means that a vertex with a local vertex name "3" belongs to the left-adjacent square of the current square and the offset is just −1. "vertex-offset[2] = 0" means that the vertex with a local vertex name "2" in the current square has a local vertex name "0" in the square it belongs to, i.e., the top square of the current square, and is stored at the entry 0 of the top square.

In this example, "square-offset[0] = square-offset[1] = 0" indicates that both vertices belong to the current square. "vertex-offset[0] = 0" and "vertex-offset[1] = 1" mean that the vertex indices of both vertices are stored in the entries 0 and 1 of the current square respectively. Therefore, we can access these two vertex indices and output them to the edge index array (equivalent to the triangle index array in 3D).

The next edge detected is in square[1, 1] with a case type 3. Currently, LP = 6. Based on "new-voxel-vertex[3] = {1}", only one new-voxel vertex is newly generated for this square and assigned a vertex index 2 which will be stored in vertex-index[6, 1]. Note that vertex-index[6, 0] is left empty since there is only one new-voxel vertex for this square. For the output of edge indices, based on edge[3] = {3, 1}, we can access the vertex indices from vertex-index[LP + square-offset[3], vertex-offset[3]] and vertex-index[LP + square-offset[1], vertex-offset[1]] and output them to the edge index array. With the concept of the new-voxel vertex and the associated tables, the vertex array and edge index array are generated efficiently without the need for searching.

The same process will be repeated continuously until all squares have been processed. In fact, since the vertices created for each square will be used only by the adjacent squares, the vertex-index array only needs to store two rows of the vertex indices instead of the whole object space. The additional space needed is rather small and can be ignored compared to the size of the original input data.

4. Discussions of the Related Issues

4.1 Discussion of the Additional Memory Needed by the GMC Algorithm

Several look-up tables have been described previously. Apparently the space needed for these tables is very small and can be ignored. Since we assume that the cubes are scanned in a row-by-row, slice-by-slice fashion, the additional data structure we use, i.e. the vertex-index-array, can be reduced to only two slices in size, i.e. the current slice the GMC algorithm is scanning currently, and the previous slice. Therefore, the additional memory needed for the GMC algorithm can be ignored.

4.2 How to Combine the GMC Algorithm with the Other Existing MC Acceleration Algorithms

We did not compare our algorithms with other acceleration MC algorithms that also speed up the original MC algorithm significantly [15]. This is because the GMC algorithm tries to eliminate the "search process", which was never considered by other algorithms before. The GMC algorithm also does not conflict or compete with other algorithms. The existing acceleration MC algorithms mostly try to avoid traversal of inactive cubes for dynamic threshold range applications that are not considered by the GMC algorithm. Therefore, it does not make too much sense to directly compare the performance of the GMC algorithm with that of those acceleration MC algorithms. In fact, some of those algorithms can be incorporated with our algorithm. In other words, the GMC algorithm is complementary to those algorithms rather than in competition with them.

For instance, for the interval-based algorithms [6], [16]–[18], the active cubes can be extracted for a given threshold range first. Then, all the active cubes can be easily converted to an auxiliary data structure such as the quadtree-segment data structure [10] that organizes all the active cubes in a row-by-row, slice-by-slice fashion. Since all the coordinates of the active cubes are integers, this implies that we do not really need to use an O(N log N) sorting algorithm to sort all the active cubes according to their geometrical coordinates. This "sorting" process can be done within an O(N) time where N is the number of active cubes. Furthermore, if the number of active cubes is relatively small compared with that of the overall cubes in the 3D space, this conversion can be done very efficiently. In this case, the interval-based algorithm can be viewed as a pre-processing of the GMC algorithm.

As for the hierarchical-geometric algorithms [7], [19], [20], in general, the hierarchical-geometric data structure can be traversed in a front-to-back, top-to-bottom fashion that is equivalent to the way the GMC algorithm scans the cubes described previously. In fact, this fashion of scanning the cubes is also usually used in the hierarchical-geometric algorithms for hidden surface removal process. Therefore, the same GMC idea can be applicable to speed up the original algorithms.

As for the propagation-based algorithms [8], [21], [22] since it is difficult to control the traversal direction in a row-by-row, slice-by-slice fashion, they can not be easily incorporated into the GMC algorithm. In this case, Itoh et al. [22] discussed in Sect. 2.3 can be used to improve the performance. These algorithms avoid the traversal of inactive cubes and can save a significant amount of time for

the surface detection if the visualization space is huge and the object of interest is relatively small. They can also be used for detecting a connected object while others described above can not. In other words, different algorithms are used for different situations and there is no single winner.

4.3 How to Parallelize the GMC Algorithm

Recently, due to the advances in CPU (Central Processing Unit) and GPU (Graphics Processing Unit) development, parallel computing is becoming very important and popular. Because the GMC algorithm is based on the original MC algorithm that determines object surfaces within a cube locally, both the MC and GMC algorithms can be easily parallelized in the following fashion to further speed up the execution time.

We can simply partition the whole 3D space into several 3D sub-spaces, each of which can be independently computed in parallel by GPGPU (General-Purpose Computing on Graphics Processing Units) using some software computing engines, such as CUDA (Compute Unified Device Architecture), or by one of the cores in a multi-core CPU system. We have implemented the parallel version of our software system, and based on our experiments the system indeed speeds up the execution time linearly with the number of cores in a multi-core CPU system. This is due to the fact that the parallel version of the GMC algorithm uses a straight-forward space-partition method, and there is virtually no parallelization overhead.

5. Conclusions

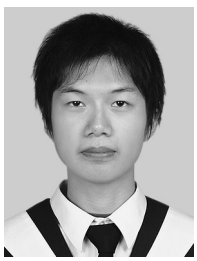
Although it was pointed out that for the original MC algorithm, we can limit vertex calculations to once per vertex to speed up the surface extraction process, however, neither how this process could be done efficiently nor the reuse of these MC vertices was looked into seriously in the literature. The concepts of the GMC and the new-voxel vertex take advantage of spatial coherence to make the MC algorithm very efficient, and can be applied to any kinds of data structures as long as they scan the data in a row-by-row, slice-by-slice fashion. Furthermore, those algorithms that speed up the MC algorithm in various ways as discussed in Sect. 2.1 and 4.2 do not conflict with the proposed algorithm and can be incorporated into our system to boost up the performance even further when necessary. The GMC triangles can be efficiently extracted without the need for a search process to determine whether a vertex has been defined or not. All of the operations involved consist mainly of table look-ups. In practice, this results in a very efficient execution of the algorithms for interactive applications.

References

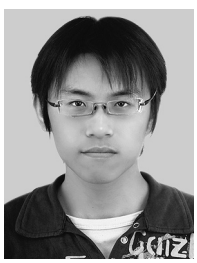
[1] A. Kaufman, D. Cohen, and R. Yagel, "Volume graphics," Computer, vol.26, no.7, pp.51–64, July 1993.
[2] M. Levoy, "Display of surfaces from volume data," IEEE Comput. Graph. Appl., vol.8, no.3, pp.29–37, May 1998.
[3] W. Lorensen and H. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," Comput. Graph., vol.21, no.4, pp.163–169, July 1987.
[4] P-W. Liu, L-S. Chen, S-C. Chen, J-P. Chen, F-Y. Lin, and S-S. Hwang, "Distributed computing: New power for scientific visualization," IEEE Comput. Graph. Appl., vol.16, no.3, pp.42–51, May 1996.
[5] http://www.vtk.org/
[6] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno, "Speeding up isosurface extraction using interval trees," IEEE Trans. Vis. Comput. Graph., vol.3, no.2, pp.158–170, April 1997.
[7] PM. Sutton and CD. Hansen, "Accelerated isosurface extraction in time-varying fields," IEEE Trans. Vis. Comput. Graph., vol.6, no.2, pp.98–107, April 2000.
[8] T. Itoh, Y. Yamaguchi, and K. Koyamada, "Fast isosurface generation using the volume thinning algorithm," IEEE Trans. Vis. Comput. Graph., vol.7, no.1, pp.32–46, Jan. 2001.
[9] J. Beyer, M. Hadwiger, S. Wolfsberger, and K. Bühler, "High-quality multimodal volume rendering for preoperative planning of neurosurgical interventions," IEEE Trans. Vis. Comput. Graph., vol.13, no.6, pp.1696–1703, Nov. 2007.
[10] L-S. Chen and M. Sontag, "Representation, display, and manipulation of 3D digital scenes," Computer Vis. Graph. Image Process., vol.48, no.2, pp.190–216, Nov. 1989.
[11] S. Islam, D. Silver, and M. Chen, "Volume splitting and its applications," IEEE Trans. Vis. Comput. Graph., vol.13, no.2, pp.193–203, March/April 2007.
[12] A. Joshi, D. Scheinost, KP. Vives, DD. Spencer, and LH. Staib, "Novel interaction techniques for neurosurgical planning and stereotactic navigation," IEEE Trans. Vis. Comput. Graph., vol.14, no.6, pp.1587–1594, Nov./Dec. 2008.
[13] H. Childs, MA. Duchaineau, and K-L. Ma, "A scalable, hybrid scheme for volume rendering massive data sets," Proc. 6th Eurographics Symposium on Parallel Graphics and Visualization, pp.153–162, Portland, Oregon, May 2006.
[14] A. Dietrich, E. Gobbetti, and S-E. Yoon, "Massive-model rendering techniques," IEEE Comput. Graph. Appl., vol.27, no.6, pp.20–34, Nov. 2007.
[15] TS. Newman and H. Yi, "A survey of the marching cubes algorithm," Comput. Graph., vol.30, no.5, pp.854–879, Oct. 2006.
[16] UD. Bordoloi and H-W. Shen, "Space efficient fast isosurface extraction for large datasets," Proc. 14th IEEE Visualization, pp.201–208, Seattle, WA, Oct. 2003.
[17] Y-J. Chiang, "Out-of-core isosurface extraction of time-varying fields over irregular grids," Proc. 14th IEEE Visualization, pp.217–224, Seattle, WA, Oct. 2003.
[18] H-W. Shen, "Isosurface extraction in time-varying fields using a temporal hierarchical index tree," Proc. Conf. Visualization '98, pp.159–166, North Carolina, Oct. 1998.
[19] C. Montani, R. Scateni, and R. Scopigno, "Decreasing isosurface complexity via discrete fitting," Computer Aided Des., vol.17, no.3, pp.207–232, March 2000.
[20] F. Velasco and JC. Torres, "Cells octree: A new data structure for volume modeling and visualization," Proc. Vision Modeling and Visualization '01, pp.151–158, Stuttgart, Germany, Nov. 2001.
[21] T. Itoh and K. Koyamada, "Automatic isosurface propagation using an extrema graph and sorted boundary cell lists," IEEE Trans. Vis. Comput. Graph., vol.1, no.4, pp.319–327, Dec. 1995.
[22] T. Itoh, Y. Yamaguchi, and K. Koyamada, "Fast isosurface generation using the cell-edge centered propagation algorithm," Proc. 3rd International Symposium on High Performance Computing, pp.547–556, Tokyo, Japan, Oct. 2000.

Lih-Shyang Chen received the BS and MS degrees in electrical engineering from the National Cheng-Kung University in 1978 and 1980, respectively and the PhD degree in computer and information science from the University of Pennsylvania in 1987. He is currently a professor of electrical engineering at National Cheng-Kung University. His research interests include computer graphics, image processing, computer vision, and distributed computing.

Young-Jinn Lay received the MS degree in electrical engineering from the Dayeh University in 1996. He is currently a PhD student in the Department of Electrical Engineering at National Cheng-Kung University. His research interests include computer graphics, visualization, and image processing.



Je-Bin Huang received the BS degree in Math from the National Cheng-Kung University in 2009. He is currently MS student in the Department of Electrical Engineering at National Cheng-Kung University. His research interests include computer graphics, visualization, and image processing.



Yan-De Chen received the BS degree in CS from the Fu-Jen University in 2009. He is currently MS student in the Department of Electrical Engineering at National Cheng-Kung University. His research interests include computer graphics, design patterns, and image processing.

Ku-Yaw Chang received the BS, MS, and PHD degrees in electrical engineering from National Cheng-Kung University in 1993, 1995, and 2002 respectively. He is currently an assistant professor of Da-Yeh University, Taiwan. His research interests include medical image diagnosis, image processing, and pattern recognition.

Shao-Jer Chen received the PHD degree from the National Cheng-Kung University in 2008. He is currently a doctor in Buddhist Tzu Chi General Hospital, Taiwan. His research interests include medical image diagnosis and image processing.