An Extended Cuberille Model for Identification and Display of 3D Objects From 3D Gray Value Data

Xiaoqing Qu and Wayne Davis
Department of Computing Science
University of Alberta
Edmonton Canada T6G 2H1

Abstract

This paper presents an extended cuberille model for the identification, reconstruction, and display of 3D objects from 3D gray value data. 3D edge elements are gradients detected at voxels, and the orientations of gradients are quantized to 26 directions. The edge elements are then converted to the extended cuberille model. The model has four volume primitives. Besides a cube, voxels are extended to include three other polyhedra so that voxel faces are compatible with 26 gradient orientations. The merits of the three representation schemes: space occupancy enumeration, octree, and surface representation by the extended cuberille model are briefly discussed. To identify border voxels, asymmetric Gaussian filters are applied to compute second derivative at each voxel. Conditions are defined for identifying border voxels based on the sign of the second derivative. From these conditions, there exists exactly one layer of border voxels, and subsequent surface tracking is therefore straightforward. Experimental results of 3D surface identification, tracking, and display by the extended cuberille model on test data and medical data are given. Because there are only four types of external voxel faces in the model, a surface of any object consists of only the four types of external voxel faces.

1 Introduction

This paper presents an extended cuberille model for identification, reconstruction and display of 3D objects from 3D gray value data. In 3D identification, thresholding is widely used but restricted. In a variety of applications, not many objects can be identified by simple thresholding. Identification based on 3D edge detection is a more general method. Some 3D edge operators have been proposed to detect edge elements [MR81] using gradients.

The problem of how to group detected edge elements to reconstruct an integral object has not been discussed. An integral object representation is a relatively new topic [Man88]. It implies that if an object is represented by its surface, the surface must be closed and without missing faces. If the object is represented by a collection of voxels, each voxel is a solid with a thickness so that the space occupied by the object can be measured.

Edge elements have little geometrical information because their shape and size are undefined. As a result, they cannot be used to construct an object. What is needed are modeling primitives whose shape and size are defined and whose orientations are close to their edge counterparts, i.e., a **model**. Currently available models are: cuberille [HL79] and a polyhedral model [LC87]. Both models use thresholding to identify objects.

In the cuberille model, an object is frequently represented as a collection of cube shaped voxels. Volume rendering algorithms can be seen in [Rey87]. An object can be represented by voxel faces. Algorithms tracking a surface of a binary image have been seen in [AFH81] and [GU89]. Because of the regularity of voxels, an object can also be organized as an octree. Octree related algorithms include tree generation algorithms [Sam80, YS83], set operation algorithms [HS79], geometric transformation algorithms [JT80, Mea82], and display algorithms [ZD91].

In the polyhedral model [LC87], a cube is made up of eight voxels in eight vertices, that are either inside or outside determined by thresholding. The algorithm marches cube by cube, creating triangle faces to model a piece of surface within each cube. A surface made of triangle faces can be displayed using a graphics package.

Some results use 3D edge detection to identify a surface. The 3D boundary following algorithm by [CR89] constructs a 3D boundary by stacking 2D boundaries, that are extracted using a graph search algorithm [Mar76].

In the following, the motivation to extend the cuberille model for 3D edge detection and surface construction is discussed.

Suppose the 3D edge operator [MR81] is applied to gray value data. It detects edges by computing the gradient at each voxel. The gradient at a voxel can be written as a vector $\nabla = (\nabla_x, \nabla_y, \nabla_z)$ with the three components indicating intensity changes along three principal axes. The magnitude of the gradient, approximated by $|\nabla| = |\nabla_x| + |\nabla_y| + |\nabla_z|$, indicates the possibility that the voxel is an edge element. The larger the magnitude, the more likely that it is an edge voxel. The direction of the gradient vector determines the edge orientation. For simplicity, the direction of a gradient is quantized to one of 26 directions (see Fig 1.)

Suppose an edge is detected at a voxel with orientation

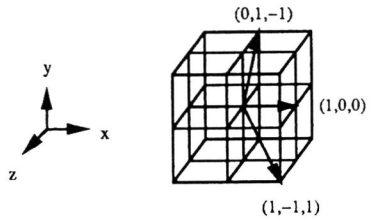


Figure 1: Three of the 26 gradient directions

(1,1,1), as shown in Fig 2(a). The cuberille model would represent the edge by three voxel faces whose average normal is (1,1,1), as shown in Fig 2(b). But it is natural to represent the edge by a face whose normal coincides with the edge orientation. For example, the triangle face in Fig 2(c) would construct a smoother surface than the three voxel faces. Of course, any other face with the same normal could also be chosen as a face primitive. So the problem is to choose a set of modeling primitives whose normals reflect the 26 gradient directions and therefore result in a smoother surface. To solve the problem the **Extended Cuberille** model is developed in the next section.

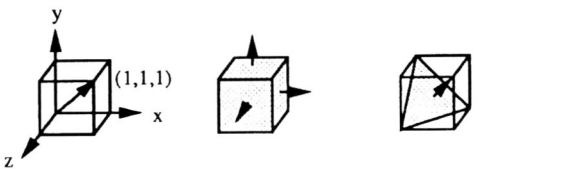


Figure 2: Different modeling primitives

2 The Extended Cuberille Model

To model voxels, the modeling volume primitives are extended to include three other polyhedra (see Fig 3) so that the voxel faces are compatible with the 26 gradient orientations.

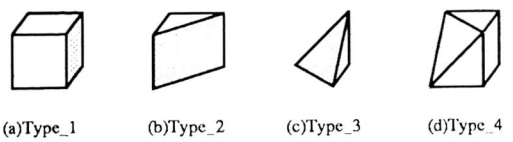


Figure 3: The set of volume primitives

As shown in Fig 3, cutting a cube voxel through diagonals of top and bottom faces gives the second polyhedron with a face orientation of $(1, 0, 1)$. Cutting a cube voxel through three vertices defines the third and the fourth polyhedra, and gives a face oriented at $(-1, 1, 1)$. Since the last three polyhedra are not symmetric, it is assumed that a volume primitive is invariant under 90 degree rotations about any principal axis. Hence the faces of the set of volume primitives give 26 orientations. Furthermore, to be able to represent objects hierarchically, it is also assumed that a primitive scaled by a power of two is also the same. The formal definition is given in the following.

Let V be the set of four volume primitives shown in Fig 3, i.e., V={*type_1*, *type_2*, *type_3*, *type_4*}. Let T denote the transformation of 90 degree rotations about the principal axes or power of two scalings. Thus T is an equivalent relation on V and each *type_k*, $k = 1, 2, 3, 4$, is an equivalence class by T. Let I be the 3D coordinate set, $I \subset Z^3$, Z is the set of integers, and f is a map, $f : I \rightarrow V$, then

Definition 1 *A voxel v_i is a pair of (i,f(i)), where $f(i)$ is a volume primitive, i.e., $f(i) \in V$, of minimum scale and i are the coordinates, $i \in I$.*

Arguably, a cube could be cut other ways, as shown in Fig 4, that could also result in 26 face orientations. But the set V is better than other choices for it is the smallest set that is closed under the subdivision operation.

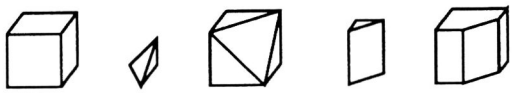


Figure 4: Another set of volume primitives

Theorem 1 *The set V is closed under the subdivision operation, and it has the least cardinality among those having the same property.*

Proof: It follows directly from subdiving the four volume primitives, as shown in Fig 5(a) to (d), that the V set is closed under subdivision. It remains to be shown that it has the least cardinality. Let U be any set of volume primitives whose type_3 and type_4 polyhedra are obtained by cutting a cube voxel, but not passing through face diagonals. Fig 5(e) shows different cuttings to get a face normal $(1, 1, 1)$. Assume the edge of a face is cut at a ratio $u/(1-u)$, as shown in Fig 5(f), then subdivision of the cube results in a new polyhedra with edge cut ratio of $u/(0.5-u)$. If U is closed, it must include at least two sets of polyhedra from the two ways of cutting to give the same face orientation. The same argument holds for a type_2 polyhedron. It follows $|U| > |V|$.

Since the set V is closed under subdivision, it is also possible to represent objects by octrees.

The mathematical definition of an object in the extended cuberille model is given in the following:

Definition 2 *An object S is a regularized union of voxels, $S = \cup^*_{i \in I} v_i$, where $\cup^*$ denotes the regularized union operation.*

The regularized set operations, see [Man88], defines $A \cup^* B = c(i(A \cup B))$, where $c(A \cup B)$ and $i(A \cup B)$ denote the closure and interior of $A \cup B$.

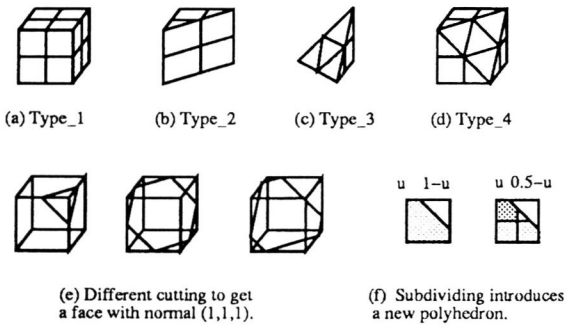


Figure 5: The set V is closed under subdivision.

The definition gives the constructive process: because voxels can only touch in faces, edges or vertices, it implies $v_i \cap^* v_j = \phi$ for $i \neq j$. Thus $\cup^* v_i$ removes all internal voxel faces to make a solid object with one interior enclosed by a surface.

There are also several ways to represent objects in the extended cuberille model. In the next Section, the merits of three representation schemes, the enumeration scheme, octrees, and surface representation, are briefly discussed.

3 Merits of Representation Schemes

The spatial enumeration scheme in the extended cuberille model lists all voxels whose spaces are either **fully** or **partially** occupied by an object. The interior of an object is filled by type_1 voxels because they are fully occupied. All type_2 to type_4 voxels represent partially occupied space and are therefore on the border of the object. Type_1 voxels could also be on the border if the surface passes through at least one of its faces. For example, Fig 6(a) to (c) show an mathematically defined object, its space enumeration representation in the extended cuberille model, and its space occupied in the cuberille model. Although the representation gives a smoother surface, it is not as storage efficient as its counterpart in the cuberille model, for it needs two arguments – coordinates, type or gradient direction – to list a voxel.

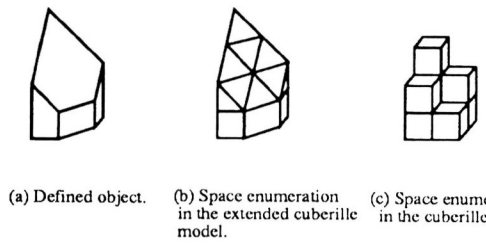


Figure 6: A mathematically defined object and its representation.

An octree in the extended cuberille model may have **five** types of leaf nodes. In addition to the cube shaped black and white leaf nodes, type_2, type_3 and type_4 leaf nodes represent partially occupied space and are all black.

An octree representing the object in the previous example is shown in Fig 7(d). Compared with the octree in the cuberille model (Fig 7(c)), the octree in the extended model is more concise because it includes leaf nodes to represent certain partially occupied space. Now consider two extreme cases shown in Fig 8. For the object in Fig 8(a), the octrees for the two models would be the same. For the object in Fig 8(b), the subdivision around the border in the cuberille model would reach the voxel level, whereas there is no subdivision in the extended model, because the root node is a leaf node. For the object with a surface slope as shown in Fig 8(c), the subdivision in the extended mode would, in the worst case, reach the voxel level. The following conclusion results:

Theorem 2 *To represent an object, the size of an octree in the extended cuberille model is at most the size of an octree in the cuberille model.*

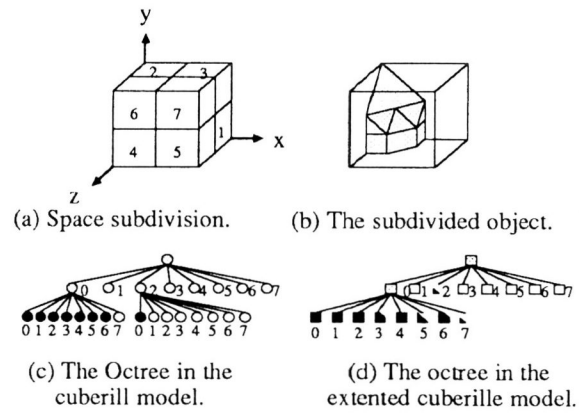


Figure 7: An octree represented object in the extended cuberille model.

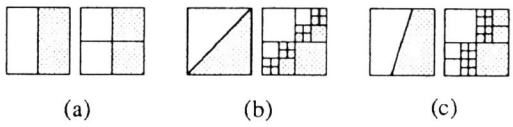


Figure 8: Comparing tree size in two models.

A surface representation lists all voxel faces on the surface. Since any object is made up of four types of voxels, and by examing Fig 3, there are only four types of voxel faces, as shown in Fig 9. It therefore follows:

Theorem 3 *The four types of external voxel faces shown in Fig 9 can close the surface of any object.*

Since there are only four types of voxel faces in the extended cuberille model, it is expected that the implementation of a surface representation is not very complicated. On the other hand, the surface of an object may not always be very smooth.

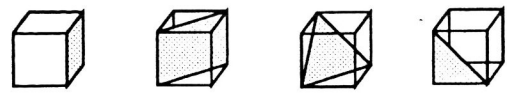


Figure 9: Four types of external voxel faces.

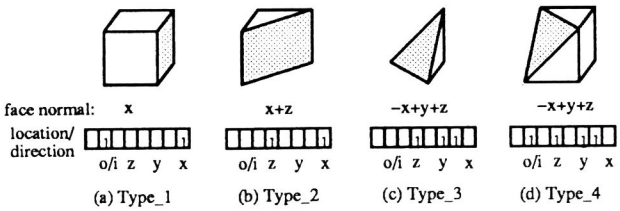


Figure 10: The location/direction codes for the four volume primitives.

4 Converting to the Extended Cuberille Model

This Section gives the implementation for converting edge elements to modeling primitives in order to display an edge image.

The characteristic of the extended cuberille model is that each voxel has a face whose normal coincides with the edge orientation. This face is termed a **face primitive**. A voxel can therefore be referred to as either a volume primitive or a face primitive. The next section gives the implementation using a one-byte code to record edge orientation.

4.1 The location/direction code

An edge specified by a gradient vector has both magnitude and orientation. The magnitude is stored as an integer, and the orientation is converted to a one-byte location/direction code, loc_dir code for short. Since there are 26 edge orientations, the orientation of an edge is recorded in the lower six bits of its loc_dir code with one bit for each $x, -x, y, -y, z$ and $-z$ direction. Each location/direction code corresponds to a volume primitive. Fig 10 gives the loc_dir code for the four volume primitives. Since type_3 and type_4 face primitives have the same normal, they are distinguished by the fact that a type_3 voxel is outside an object and a type_4 voxel is inside. Bit six of the location/direction code is the inside/outside bit. Because a type_4 voxel is inside, bit six of its loc_dir code is set to one.

The inside/outside question is tested with the edge direction as follows: if v_1 and v_2 are the two voxel neighbors in the $3 \times 3 \times 3$ neighborhood along the gradient direction, then v_0, the central voxel, is inside if $I(v_0) \geq [I(v_1) + I(v_2)]/2$, otherwise it is outside. $I(v_i)$, $i = 1, 2$, is the intensity value at voxel v_i and it is assumed that objects have larger intensity values than the background.

A type_1 voxel is always set to inside. A type_2 voxel can be set to either inside or outside. In the implementation, it is set to outside.

Bit 7 of the loc_dir code is used to mark a voxel visited in the surface tracking algorithm (see Sec. 6).

Hence applying the 3D edge operator to a 3D array results in an array of edge magnitudes and location/direction codes.

4.2 Display an Edge Image

The location/direction code, along with the voxel coordinates, specifies the type (shape), location and normal of the corresponding face primitives, but not the size. This section describes how to store and access the geometrical information of face primitives so that an edge image can be displayed.

A data structure, called TABLE, is created to save the geometrical information of the four face primitives. The first two levels of the table necessary to display an edge image are depicted in Fig 11. The table has 128 entries, corresponding to the lower seven bits of loc_dir codes. The entry index specifies the type of face primitives stored in the entry. Each table entry has two fields: a pointer, called face, pointing to a BASIC_FACE structure where the indexed face is stored, and a coordinate transformation matrix. Since the lower seven bits are not fully used, some entries are empty.

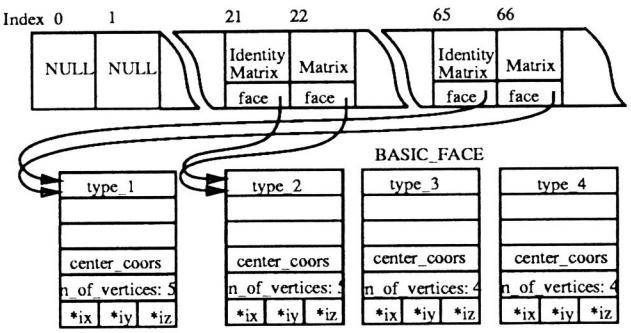


Figure 11: The TABLE structure

A face primitive in an basic_face entry is defined by three lists of vertex coordinates, *ix,*iy,*iz. Some other information such as the number of vertices, the center coordinates of the face are also stored in the structure. To display an edge, the edge loc_dir code is used to index a table entry, and the face vertices pointed to by the face pointer are read out and take a coordinate transformation with the entry matrix so that the resulting face normal is consistent with the edge orientation.

5 Border voxel identification

Once the edge elements are converted to modeling primitives, the next problem is to identify border voxels so that it is possible to reconstruct and display the surface.

Since the second directional derivative of an intensity change has a steep slope around the zero crossing, that cor-

responds precisely to the peak of the gradient (see Fig 12), Marr and Hildreth [MH80] suggested that zero-crossings can be used to detect edge elements. To compute zero-crossings, a Gaussian filter is used to smooth the noise. Because the intensity change of an image is usually unknown, the directional second derivative is taken against the Gaussian filter. Hence zero-crossings are zero points resulted from convolving the directional second derivative of a Gaussian filter with image intensity. In this paper, a border identification method based on signs of second derivatives is proposed.

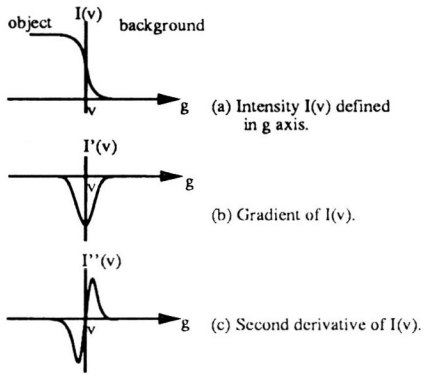


Figure 12: The zero-crossing corresponds precisely to the peak of the gradient.

Since intensity value has maximum change rate in the gradient direction, it is reasonable to assume that in a small neighborhood of a voxel, intensity doesn't change significantly in the perpendicular direction. This suggests that an asymmetric 3D Gaussian filter can be used to smooth the data. The long scale of the Gaussian filter coincides with the gradient direction of a voxel, which is one of the 26. Because of shorter scales in the other two dimensions, computing 3D convolutions can be speeded up.

Let $I(x,y,z)$ be the intensity function at voxel $v(x,y,z)$, and G''_g be the second derivative of an asymmetric Gaussian filter in the gradient direction g of voxel v. Denote $w(x,y,z)$ the result of the discrete convolution of $G''_g(x,y,z)$ with $I(x,y,z)$,

$$w(x,y,z) = G''_g(x,y,z) * I(x,y,z). \quad (1)$$

To compute the discrete convolution, a $\{v;n,t,b\}$ coordinate system is established at every voxel v. The axis $\vec{n}$ at v is always set in the gradient direction of v, one of the 26, so the system $\{v;n,t,b\}$ changes from voxel to voxel. Transform the coordinate system $\{0;x,y,z\}$ to $\{v;n,t,b\}$ and express the above convolution in terms of (n,t,b) as

$$\tilde{w}(n,t,b) = \sum_i \sum_j \sum_k G''_n(i,j,k) I(n-i,t-j,b-k). \quad (2)$$

At the origin v,

$$\tilde{w}(0,0,0) = w(x,y,z).$$

Zero-crossings are those voxels whose $w(x,y,z) = 0$. Asymmetric Gaussian filters for 26 gradient directions have been designed (see [QD92]).

The $w(x,y,z)$ in (2) is defined on voxels of integer coordinates and is undefined between voxels. Because of the discrete nature of voxels, not many voxels have $w(x,y,z) = 0$. Observe Fig 12, however, that for a bright object surrounded by a darker background and for those voxels $v(x,y,z)$ close to the border, $w(x,y,z)$ is negative inside the object and positive outside. There exists exactly one layer of voxels on which $w(x,y,z)$ is negative and changes sign for neighbors in the g-direction. This layer is called the negative layer. Similarly, there exists exactly one positive layer of voxels. It is possible to define either the negative layer or the positive layer as the border. Since the negative layer is part of the object, the border is defined as the negative layer of voxels. zero-crossing voxels can be treated as either positive or negative, and are also included in the border set.

Therefore, for a bright object surrounded by a darker background, if $v(x,y,z)$ is a border voxel, it must simultaneously satisfy the following inequalities:

$$\begin{array}{ll} w(x,y,z) \le 0, & \\ w(x-1,y,z) < 0,\ w(x+1,y,z) > 0, & \text{if } \nabla_x \neq 0, \\ w(x,y-1,z) < 0,\ w(x,y+1,z) > 0, & \text{if } \nabla_y \neq 0, \\ w(x,y,z-1) < 0,\ w(x,y,z+1) > 0, & \text{if } \nabla_z \neq 0. \end{array} \quad (3)$$

Similarly, for a dark object surrounded by a brighter background, the border voxels can be defined as the positive layer of voxels. The gradient components $\nabla_x, \nabla_y, \nabla_z$ of a voxel v are saved in its loc_dir code and can be easily tested.

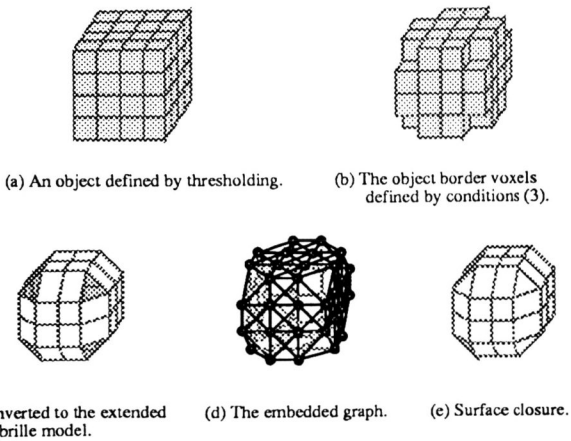
The inequalities (3) are conditions to identify border voxels. Obviously, there is only one layer of voxels that will satisfy the conditions. This makes subsequent surface tracking straightforward. A surface tracking algorithm is given in the next section.

6 The Surface Tracking Algorithm

The surface tracking algorithm traverses border voxels, converts the voxels to face primitives defined in the extended cuberille model, and connects the faces in a closed surface.

An example is given in Fig 13 (a) to (e) to show how an object is coverted to a surface. The object is a $4 \times 4 \times 4$ cube defined by thresholding. According to conditions (3), however, the object border is the set of voxels shown in (b). The border voxels are converted to the extended cuberille model in (c), where each border voxel is represented by a face primitive. Because each voxel is converted to one face primitive, the resulting surface in (c) is not closed. Therefore there is one more step to fill missing faces to close the surface, as shown in (e).

The surface tracking algorithm is outlined in Fig 14. It has basically two tasks: calling procedure `Border_face_tracking()` in line 6 to traverse all the border voxel faces, and calling procedure `Close_surface()` in line 7 to connect border voxel faces to close the surface.



(a) An object defined by thresholding. (b) The object border voxels defined by conditions (3).

(c) Converted to the extended cubrille model. (d) The embedded graph. (e) Surface closure.

Figure 13: An example to show the surface tracking steps.

```
    Surface_tracking(object)
    char *object;
    {
1       read_data(object);
2       edge_detector();
3       zero_crossing();
4       init_lists();
5       make_table();
6       Border_face_tracking();
7       Close_surface();
    }
```

Figure 14: The surface tracking algorithm

Before tracking the border voxel faces, from lines 1 to line 3, the procedure `read_data()` reads the data named `object` to an 3D unsigned character array `scene`. The procedure `edge_detector()` applies a 3D edge operator to the data array, resulting in an 3D unsigned character array `grad_loc_dir` of location/direction codes, and a 3D short integer array `mag` of gradient magnitudes. The loc_dir code of a voxel reflects the face normal that is necessary information for display. The gradient magnitude is used as a rough threshold to assist in border voxel identification. The procedure `zero_crossing()` computes $w(x, y, z)$ at every voxel $v(x, y, z)$, resulting in a 3D short integer array `w`. Implementations of these procedures are straightforward, hence no details will be given here.

In lines 4 to 5, procedures `init_lists()` and `make_table()` initialize all associated data structures, such as queue, lists, tables, etc., used by the `Border_face_tracking()` and `Close_surface()` procedures.

As shown in Fig 13(d), if border voxels and the adjacency relations between voxels can be modeled as a graph, in which nodes are border voxels and edges are adjacency relations between pairs of voxels, a standard breadth-first search can be used for tracking border voxels. The `Border_face_tracking()` procedure is outlined in Fig 15.

```
    int       x,y,z;
    Q_CELL    *current_voxel;

    Border_face_tracking()
    {
        unsigned char   index;

1       get_start_face;
2       span_start_face();
3       while(current_voxel=de_queue_front() {
4         x=current_voxel->x;
5         y=current_voxel->y;
6         z=current_voxel->z;
7         index=grad_loc_dir[x][y][z] & ~MARK;
8         Neighbor_face(index);
9         add_to_display_table(x,y,z,index); }
    }
```

Figure 15: The border face tracking procedure.

The underlying data structure is a queue, where each queue cell has three integers, `x,y,z`, for recording the coordinates of a border voxel.

In the procedure, the micro `get_start_face` in line 1 reads the coordinates of a start border voxel, and the procedure `span_start_face()` in line 2 searches its adjacent border voxels, marks them and queues them.

The `while` loop in line 3 starts tracking border voxels. A queue cell is obtained from the queue as the current voxel. In line 8, procedure `Neighbor_face()` is called to search for the current voxel's adjacent border voxels. It scans all adjacent voxels, testing if any satisfies the condition (3). Meanwhile if a border voxel is unmarked, mark it and queue it. After the procedure `Neighbor_face()` returns, `add_to_display_table()` is called to add the current voxel coordinates, `x y z`, and its loc_dir code, `index`, to a structure array to display the voxel face later. While the queue is not empty, the loop continues to the next cell in the queue. The `while` loop stops once the queue is empty.

The time complexity of `Border_face_tracking()` depends on the `while` loop and the procedure call `Neighbor_face()`. It is analyzed below. The data to start tracking are three 3D arrays, `grad_loc_dir`, `grad_mag`, and `w`. Bit seven of `grad_loc_dir` is designated for marking, hence checking and marking a voxel visited can be done in constant time. As a result, the time to execute `Neighbor_face()` depends on the number of adjacent voxels that a border voxel could have. Since each unmarked border voxel is placed in the queue once, the `while` loop is executed only once for every border voxel. Denote the number of adjacent voxels that a border voxel has as k, and the total number of the border voxels as n_B, the time com-

plexity of `Border_face_tracking()` is therefore $O(kn_B)$.

For a given object, n_B is determined by the conditions (3) and is fixed. But k varies with the number of adjacent voxels that a border voxel could have. A way to define the adjacency relation between pair of voxels is by digital topology, where two voxels are adjacent each other if they are either face, edge or vertex connected. By this definition, each voxel has 26 adjacent voxels. Hence the constant k is about 26.

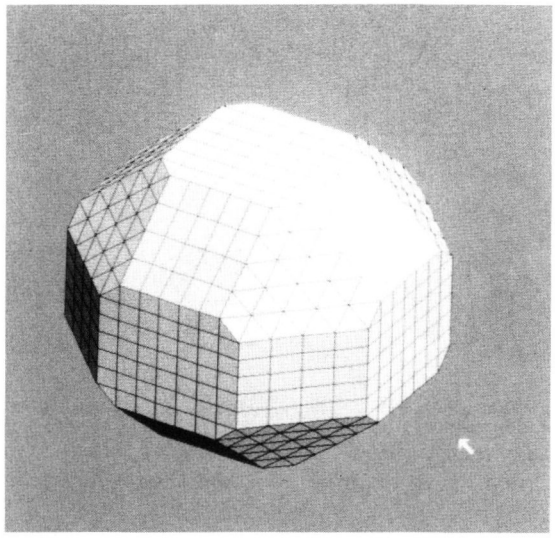
Observe from Fig 13(c), however, that in the extended cuberille model, every border voxel can be converted to a face primitive, and all the face primitives are on a surface. Intuitively, a face normal shouldn't have dramatic change from one border voxel to an adjacent one because the surface is supposed to be smooth. This suggests that the face normal of a border voxel can be used to assist in defining adjacent voxels. This results in less than 26 adjacent voxels. Furthermore, face primitives are either square or triangular face, see Fig 3, hence a face primitive has at most four edges. As a result, there are at most four ways to connect the current face to the next face. The neighbor connections are called **outways** of the current voxel. In the implementation, the `Neighbor_face()` procedure searches outways instead of all adjacent voxels. Whenever a border voxel of an outway is found, the search breaks and proceeds to the next outway. This reduces the constant k to about half, and speeds up border face tracking. No further details will be given. The interested reader may refer to [Qu92].

For each adjacent border voxel found for an outway, the `Close_surface()` procedure checks face connections between the current face primitive and the adjacent face primitive. If they are connected, do nothing. Otherwise the procedure tries to find missing voxel faces and adds the missing faces to the display table. Since procedure `Close_surface()` executes only for disconnected border voxel faces, it is expected to be faster than `Border_face_tracking()`.

Back to the `Surface_tracking` algorithm in Fig 14, since the first three procedures work on the entire data volume, the algorithm is a volume based algorithm after all. Once voxel faces of a surface are saved in the display table, however, the time of all subsequent operations such as display, rotation and scaling, etc., operate on the border voxel faces, and therefore is an order of the number of border voxel faces of an object, i.e., $O(n_B)$.

7 Experimental Results

Fig 16 shows surfaces of a test object and a medical object obtained from real data. The test object is a sphere of volume $40 \times 40 \times 40$ with 16 gray values. Since the test object is darker than the background, its border is a positive layer of voxels. The surface of the sphere has 509 border voxels and 797 voxel faces. It can be seen that the surfaces consist of four types of voxel faces. The images is displayed using a graphics package WINDLIB [GB87] on a Sun3/60.



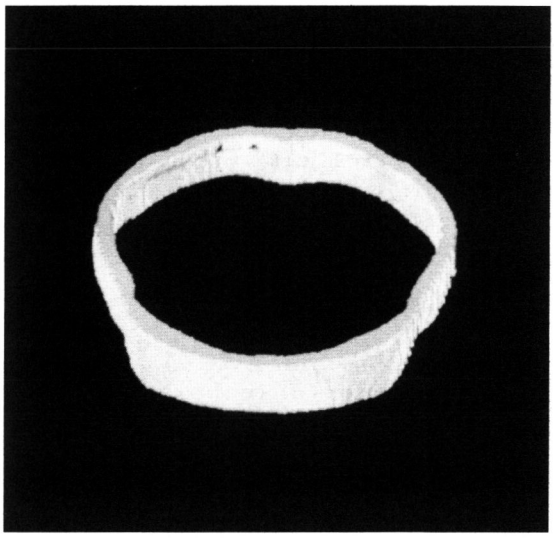


Figure 16: Surfaces of a test object and a medical object from real data.

The data size of the medical object is of $208 \times 208 \times 26$, resulting from linearly interpolating 6 CT slices producing 5 between successive pairs. The gray values were also linerly mapped to the range from 0 to 255. The object has a brighter color than the background, therefore, its border is a negative layer of voxels. There are 17,742 border voxels detected and 26,409 voxel faces on the surface. Running `gprof` shows that the execution time for `Border_face_tracking` on a Sun4 is about 55.32 seconds, and checking face connections and close the surface takes 33.26 seconds.

The image is displayed on a Silicon Graphics Iris station 4D/35. To display 26,409 faces takes only seconds.

8 Conclusion

The extended cuberille model introduced in this paper provides a way to identify, reconstruct and display 3D objects based on 3D edge detection rather than thresholding.

3D edge elements are gradients, and orientations of gradients are quantized to 26 directions. The model has four volume primitives. Besides a cube, voxels are extended to include three other polyhedra so that voxel faces are compatible with 26 gradient orientations. The merits of the three representation schemes: space occupancy enumeration, octree, and surface representation by the extended cuberille model are briefly discussed.

To identify border voxels, asymmetric Gaussian filters are convolved with the gray value data to compute second derivatives of intensity changes at every voxel. Conditions for identifying border voxels based on the signs of the second derivatives are given in inequalities (3). From these conditions, there exists exactly one layer of border voxels, and subsequent surface tracking could be simply be a breadth first search.

A surface tracking algorithm using the conditions to identify border voxel is given. The surface tracking algorithm has two tasks: traverse border voxel faces and close a surface. Analysis and experimental results have shown that the the time complexity of the surface tracking algorithm depends on the border face tracking, and is an order of number of border voxels of a tracked object. Tracking outways instead of 26 adjacent voxels is further suggested to speed up border voxel tracking.

Experimental results of 3D surface identification, tracking, and display by the extended cuberille model on a test object and a medical object from real data are given. Because there are only four types of voxel faces in the model, a surface of any object consists of only four types of voxel faces.

References

[AFH81] E. Artzy, G. Frieder, and G. T. Herman. The theory, design, implementation and evaluation of a three-dimensional surface detection algorithm. *Computer Vision, Graphics, and Image Processing*, 15:1–24, 1981.

[CR89] John Danilo Cappelletti and Azriel Rosenfeld. Three-dimensional boundary following. *Computer Vision, Graphics, and Image Processing*, 48:80–92, 1989.

[GB87] M. Green and N. Bridgeman. *WINDLIB Programmer's Manual*. Department of Computing Science, University of Alberta, Edmonton, Alberta, September 1987.

[GU89] D. Gordon and J. K. Udupa. Fast surface tracking in three-dimensional binary images. *Computer Vision, Graphics, and Image Processing*, 45:196–214, 1989.

[HL79] G. T. Herman and H. K. Liu. Three-dimensional display of human organs from computed tomograms. *Computer Vision, Graphics, and Image Processing*, 9:1–21, 1979.

[HS79] G. M. Hunter and K. Steiglitz. Operations on images using quad tres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):145–153, 1979.

[JT80] C. L. Jackins and S. L. Tanimoto. Oct-treeess and their use in representing three-dimensional objects. *Computer Vision, Graphics, and Image Processing*, 14:249–270, 1980.

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, 1987.

[Man88] M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[Mar76] Alberto Martelli. An application of heuristic search methods to edge and contour detection. *Communication of the ACM*, 19(2):73–83, February 1976.

[Mea82] D. J. Meagher. Geometric modeling using octree encoding. *Computer Vision, Graphics, and Image Processing*, 19:129–147, 1982.

[MH80] D. Marr and E. Hildreth. Theory of edge detection. *Proceeding of the Royol Society of London*, 207:187–217, 1980.

[MR81] D. G. Morgenthaler and A. Rosenfeld. Multidimensional edge detection by hypersurface fitting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(4):187–217, July 1981.

[QD92] X. Qu and W. A. Davis. Three dimensional border identification. *Proceedings of Vision Interface '92*, page to be appear, 1992.

[Qu92] X. Qu. *Identification and Display of 3D Objects From 3D Gray Value Data*. PhD thesis, University of Alberta, to be published in 1992.

[Rey87] R. A. Reynolds. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38:275–298, 1987.

[Sam80] H. Samet. Region representation: Quadtrees from binary arrays. *Computer Vision, Graphics, and Image Processing*, 13(1):88–93, 1980.

[YS83] M. Yau and S. N. Srihari. A hierarvhical data structure for multidimensional digial images. *Communication of the ACM*, 26(7):504–515, 1983.

[ZD91] J. Zhao and W. A. Davis. Fast display of octree representations of 3d objects. *Proceedings of Graphics Interface '91*, pages 160–167, 1991.