Pergamon

Comput. & Graphics, Vol. 21, No. 6, pp. 737–748, 1997

Printed in Great Britain
0097-8493/97 $17.00 + 0.00

PII: S0097-8493(97)00053-8

Technical Section

EFFICIENT POLYGONIZATION OF CSG SOLIDS USING BOUNDARY TRACKING†

CHUNG-WEN CHUNG, JUNG-HONG CHUANG‡ and PEI-HUAN CHOU

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, ROC
e-mail: jhchuang@csie.nctu.edu.tw

Abstract—In this paper, we propose a procedure that directly polygonizes the boundary surface of CSG solids. The procedure consists of a preprocessing step and a polygonization process followed by a postprocessing that recovers rounded edges and corners. In the preprocessing step, the minimum bounding volumes, called S-bounds, of all nodes in the given CSG tree are computed and then used as a basis for subdividing the S-bound of the tree's root. In addition, a regular grid embedded on the root's S-bound is constructed. The polygonization is performed in such a way that among leaf voxels of the space subdivision only voxels that overlap the solid's boundary are traversed and inside such voxels only grid cells transversal to the solid's boundary are tracked. The surface–edge intersections and vertex normals are computed directly from the exact boundary surfaces of the CSG solid. In the postprocessing step, rounded edges and corners are detected and recovered using primitive geometry. © 1997 Elsevier Science Ltd. All rights reserved

1. INTRODUCTION

In solid modeling, the two most common schemes for representing solids are constructive solid geometry (CSG) and boundary representation (B-rep). CSG representation defines objects by applying transformations and regularized Boolean operations to a set of primitive solids, such as cubes, spheres, cylinders, cones, and tori. A CSG-solid is internally represented as a binary tree, called a CSG tree, in which the leaves represent primitives and the internal nodes Boolean operators. B-rep represents a solid as a graph whose nodes contain geometric data for each face, edge, and vertex and whose arcs represent adjacency and incidence relations between nodes. Although CSG representation provides convenient editing operations for constructing objects, certain computations, such as rendering, require explicit boundary geometry that is not immediately available from the CSG representation.

CSG-solids can be displayed directly, via methods such as ray tracing, or indirectly, by first converting a CSC tree to a B-rep and then applying traditional hidden surface removals. Ray tracing is able to effectively model a wide range of visual phenomena, such as specular reflection, transparency, and shadows; but it is very time consuming in general. Although several acceleration techniques exist, such as space subdivision [1, 2] and bounding hierarchy [3], the computational issues still prevent ray tracing from being used in an interactive environment. Indirect rendering is sometimes attractive, since once the B-rep is available the viewpoint can be changed without regenerating the B-rep. Nevertheless, converting a CSG tree to a B-rep, known as boundary evaluation, is in general a highly complex task. Recently, in [4], efficient and accurate algorithms have been proposed for Boolean combinations of solids composed of sculptured models.

Some applications, such as visual simulation and virtual reality, require fast or real-time display of a complex environment. Such applications often utilize shading hardware commonly found in modern workstations and hence require polygonal representation of the 3-D environment. Although in general shading techniques alone cannot produce realistic images, radiosity and texture mapping are becoming more popular for producing realistic images of polygonal environments. Polygonal representations of a CSG solid can be obtained either by polygonizing the corresponding B-rep, or by first polygonizing all primitives and then evaluating the CSG tree on the polygonized primitives. Both of these approaches, however, are complicated, time-consuming, and sensitive to numerical problems [5]. Recently, attempts were made to display CSG solids by applying marching cube method on discrete 3-D scalar data sets [6, 7]. The approach in [6] applied marching cube on discrete 3-D scalar data sets and hence is in general time consuming and error-prone due to the difficulty of estimating the distance function. Moreover, the missing small features and rounded edges/corners inevitably cannot be detected and recovered by this kind of approach. The method used in [7] is based on marching cubes with adaptive

† Work supported by the National Science Council of the Republic of China under grant NSC 85-2213-E-009-118.
‡ Author for correspondence.

Table 1. Combination rules for union, intersection, and difference

	$A \cup^* B$			$A \cap^* B$			$A -^* B$		
	in	*out*	*on*	*in*	*out*	*on*	*in*	*out*	*on*
in	*min*	$d(A)$	$d(A)$	*max*	$d(B)$	$d(B)$	$2\text{-}d(B)$	$d(A)$	$d(B)$
out	$d(B)$	*min*	$d(B)$	$d(A)$	*max*	$d(A)$	$d(A)$	$d(A)$	$d(A)$
on	$d(B)$	$d(A)$	*min*	$d(A)$	$d(B)$	*max*	$2\text{-}d(B)$	$d(A)$	$d(A)$

$d(A)$, $d(B)$: distances from the grid point to the boundary of solids A and B, respectively.
max: $max(d(A), d(B))$.
min: $min(d(A), d(B))$.

sizes and is able to produce the meshes with different resolutions.

In this paper, we propose a procedure that directly polygonizes the boundary surface of CSG solids, without converting the CSG tree to a B-rep or evaluating the CSG tree on polygonized primitives. The procedure consists of a preprocessing step and a polygonization process followed by a postprocessing. In the preprocessing step, S-bounds of each node in the given CSG tree are computed and then used as a basis for subdividing the S-bound of the tree's root as proposed in [2]. In addition, a regular grid embedded on the root's S-bound is constructed. The polygonization is performed in such a way that among leaf voxels of the space subdivision only voxels that overlap the solid's boundary are traversed and processed, and inside such voxels only grid cells transversal to the solid's boundary are tracked in a continuation manner. The surface-edge intersections and vertex normals are computed directly from the exact boundary surfaces of the CSG solid. In the postprocessing step, based on an edge-tracking scheme rounded edges and corners are detected and recovered using primitive geometry.

In Section 2, we briefly survey previous work on polygonization of CSG solids. In Section 3, we describe the proposed polygonization procedure. Section 4 addresses implementation issues and gives some tested examples. Finally, in Section 5, we summarize the results of our work.

2. PREVIOUS WORK

The adaptive marching cubes proposed in [7] is based on octree subdivision to provide the basic discretization of the CSG solid. The vertices of the final mesh are derived using ray casting, which is in general a rather time consuming task.

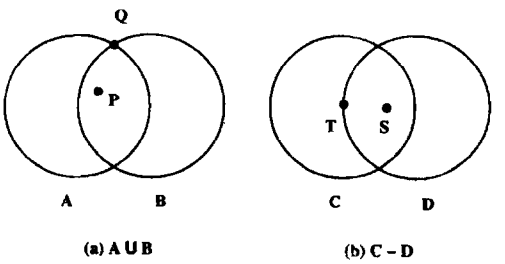


Fig. 1. Incorrect distance evaluations.

The constructive cube algorithm proposed by Breen [6] directly polygonizes the boundary of the CSG solid. The method begins by constructing a bounding box of the CSG solid from which a regular grid is constructed with a user-specified cell size. A distance function is then evaluated at each grid point to approximate the distance between the point and the boundary of the solid. Finally, a triangular mesh is computed using the standard marching cube procedure [8].

The distance function is evaluated in a way similar to the point/solid classification. The grid point is first sent to the root of the CSG tree and propagated down to the leaves. At each leaf primitive, a distance is computed and normalized such that the distance is less than 1 when the point is inside the primitive, equal to 1 when on the solid, and greater than 1 when outside the primitive. The distance values obtained for the leaves are propagated upward toward the root according to Table 1. The resulting value approximates the distance of the grid point from the solid's boundary.

The distance evaluation is simple, but highly error-prone. Consider the example shown in Fig. 1(a). The derived distance from P to $A \cup^* B$ is $min(d(A), d(B))$ according to Table 1, which is incorrect, since the closest point to P on the solid's boundary is Q. For the example depicted in Fig. 1(b), the distance from S to $C -^* D$ is $d(C)$ according to Table 1, which is also incorrect, since the closest point from the boundary of $C -^* D$ to S is T. Errors that occur in the distance evaluation will be propagated in the upward combination process and lead to inaccurate computations for the surface-edge intersection and normal.

Other major problems with the constructive cube algorithm are that all cells, including empty cells, are traversed and that at each grid point the distance is evaluated with respect to the original CSG tree. In general, octree decomposition of the object space is effective in preventing the marching cube process from traversing voxels of no interest. Moreover, with octree decomposition the CSG tree can be localized to each leaf voxel, in turn reducing the complexity of the distance evaluation. Such a reduction in localized CSG trees is, however, not as significant as that based on the space subdivision proposed in [2], which

will be briefly described in Section 3. Also, in general a detection and recovery of missing small features and sharp edges and corners will be inevitable for the constructive cube algorithm and consequently a finer grid is required for a better result.

3. EFFICIENT POLYGONIZATION OF CSG SOLIDS

The proposed procedure consists of a preprocessing step and a polygonization process followed by a postprocessing step. In the preprocessing step, the S-bounds of each node in the given CSG tree are computed and used as a basis for subdividing the S-bound of the tree's root as proposed in [2]. Such a space subdivision not only significantly reduces the localized CSG tree in each of its leaf voxels, which in turn greatly reduces the amount of computation involved in the point classification process, but also enables us to perform successful boundary tracking and to recover lost small features. In addition to the subdivision, a regular grid embedded on the root's S-bound is constructed. The polygonization is performed in such a way that among leaf voxels of the space subdivision only voxels that overlap the solid's boundary are traversed and inside such voxels only cells transversal to the solid's boundary are tracked. The surface-edge intersection and vertex normal, which are usually interpolated in the standard marching cube algorithm [8], are computed directly from the exact boundary surfaces of the CSG solid. Since geometry of primitives are used to compute the vertices of polygonal mesh, edges or corners, which are generally rounded by the polygonizer, can be easily recovered by refining those triangles that round the edges or corners. In the postprocessing step, rounded edges and corners are detected and recovered using primitive geometry and edge tracking scheme.

3.1. *Preprocessing: a space subdivision using S-bounds*

Space subdivision with a localized CSG tree in each leaf voxel is well suited for performing point/solid classification on CSG solids [9]. Traditional space subdivisions, such as octree and binary space partition, generally do not take the geometry of primitives into account. We have found that subdividing the space based on the face planes of the S-bounds [10] produces effective reduction of the size of localized CSG trees [2]. The S-bounds computation aims to obtain a minimum bounding box for each node of a CSG tree. Initially each primitive is associated with a bounding box. In the upward propagation, we evaluate the CSG tree on the basis of the current bounding boxes of the primitives, and derive a new bounding box for each internal node, including the root. In the downward propagation, we propagate the current bounding box at the root downward toward the leaves and at each level calculate the intersection to further refine the bounding box of each child node. With repeated application of the upward and downward propagations, the size of the bounding box of each node, including the root, internal nodes, and leaves, is minimized. This space subdivision consists of a preprocessing step and a subdivision computation. In the preprocessing step, we perform the S-bounds computation and obtain the S-bound for each node of the CSG tree. The space subdivision results in a hierarchical structure, called the *space subdivision hierarchy* and assigns to each leaf voxel of the subdivision hierarchy a localized CSG tree, called a *sub-CSG tree*. To construct the subdivision hierarchy of a CSG tree, we subdivide the S-bound of the root by propagating upward the subdivision hierarchies at both child nodes. This process is applied recursively to each internal node of the CSG tree. Consequently, the subdivision proceeds in a bottom-up fashion, starting from the subdivision hierarchies of primitives (*i.e.* the S-bounds of the primitives). The sub-CSC tree associated with each leaf voxel of the subdivision hierarchy is assigned or updated in the process of the upward propagation. The data structure for the voxel on each level of the subdivision hierarchy contains pointers to its child voxels and pointers to its neighboring voxels. Each leaf voxel of the subdivision hierarchy also contains, in addition to pointers, the associated sub-CSG tree. Figure 2 illustrates the resulting space subdivision and subdivision hierarchy for a CSG solid.

Such a space subdivision of the object space is useful in four ways. First, the space subdivision provides a spatial partition such that volumes of no interest will not be explored in the mesh generation process. That is, each leaf voxel of the space subdivision can be classified as *inside* or *outside* the CSG solid or *overlapping* the boundary surface of the CSG solid, and only overlapping voxels need to be explored in the polygonization process. Second, the space subdivision can localize the CSG tree to its leaf voxels and hence speed up the point classification computation. Third, the subdivision scheme ensures that the surface tracking will succeed without missing

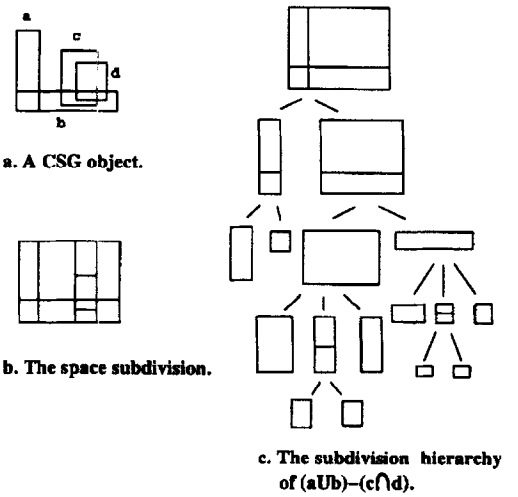


Fig. 2. The space subdivision and its hierarchy for solid $(a \cup b) - (c \cap d)$.

any surface component. Last, small features are bounded by leaf voxels of the subdivision and hence can be detected easily. In Fig. 3, thick line segments form the space subdivision in which V_1, V_2, and V_3 are examples of overlapping, inside, and outside voxels, respectively.

To set up a volumetric space for mesh generation, a regular grid that covers the S-bound at the root of the CSG tree is constructed. For a user-specified cell size s, the extent of the regular grid will be bounded by $floor((a_x, a_y, a_z)/s)s$ and $ceil((b_x, b_y, b_z)/s)s$, where *ceil* and *floor* are ceiling and floor functions, respectively, and (a_x, a_y, a_z) and (b_x, b_y, b_z) are bounds for the S-bound at the root of the CSC tree. Thin line segments in Fig. 3 construct a regular grid for the space subdivision.

3.2. *Polygonization of the CSG solids*

The polygonization process traverses the space subdivision hierarchy and for each overlapping leaf voxel it derives triangular meshes by tracking those grid cells that are transversal to the solid's boundary. A cell is called transversal if its edges intersect the solid's boundary. The surface tracking process begins by finding a *transversal* cell as a seed and then propagates the tracking to adjacent transversal cells. The surface tracking process is able to deal with cases in which more than one surface component is found inside a voxel. When a cell lying on both the current voxel and an adjacent voxel is visited, if the cell has not been visited before and, in the meantime, the adjacent voxel has been traversed, we then conclude that we are traversing into the adjacent voxel via a component that has not been tracked. In this case, the pair consisting of the cell and the adjacent voxel is placed into a queue for later processing to recover that component. In Fig. 3, the polygonization process traverses only overlapping voxels and in each of which only transversal cells are tracked.

3.2.1. *Tracking transversal cells in an overlapping voxel.* Leaf voxels of the space subdivision hierarchy are traversed in a manner similar to tree traversal. During the traversal, voxels that are inside or outside the solid are ignored; only overlapping voxels are dealt with. The overlapping voxel in general contains cells that are inside or outside the solid or on the boundary of the solid. The surface tracking procedure we propose is a continuation-like algorithm; that is, starting from a seed transversal cell, the tracking is propagated to adjacent transversal cells in a recursive manner. The seed transversal cell for a voxel is obtained by first checking if there exist such cells deposited by the tracking process on some adjacent voxels, and if not, then by a sequential search over cells in the voxel until a transversal cell is found.

From the cell c currently being visited in voxel V, we propagate to the adjacent transversal cells by the recursive call SurfaceTracking(V, c). When propagating the surface tracking to an adjacent transversal cell, we have to cheek to see if the cell has been explored before and bypass it if this is the case. The procedure SurfaceTracking(V, c) terminates when all transversal faces of cell c are either on the voxel's boundary or their corresponding adjacent transversal cells have been processed.

The surface tracking procedure will produce a connected triangular mesh in the voxel being traversed. When a voxel contains multiple triangular meshes, meshes that have not been obtained by previous trackings can be tracked from adjacent voxels intersecting the left-over meshes, as shown in Fig. 3. To robustly obtain all meshes inside a voxel, we should have a mechanism to ensure that left-over meshes will be recovered later on in the tracking process. Whenever we propagate to a cell c^* that is outside the current voxel V, we cheek to see if c^* has been visited. If it has not and the voxel V^* containing c^* has been traversed, we are traversing into V^* via a component that has not been traced and we put the pair (V^*, c^*) into a queue Q to wait for another

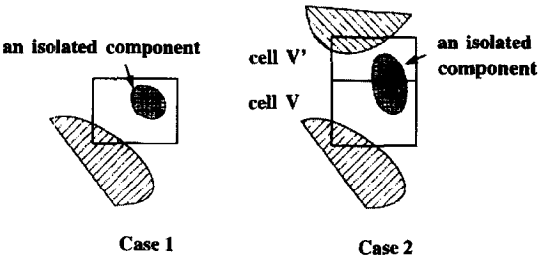


Fig. 4. Impossible configurations for missing components.

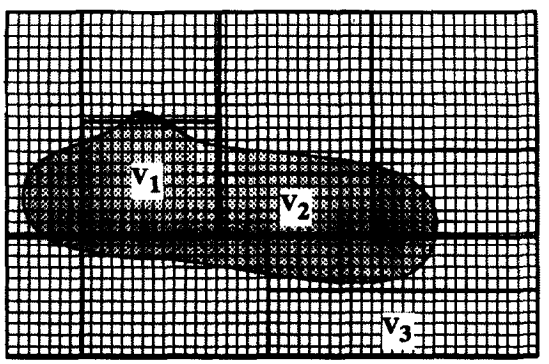


Fig. 3. The space subdivision, grid, and boundary tracking.

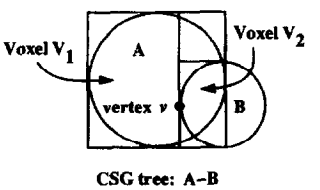


Fig. 5. Different classifications for a cell vertex.

tracking initiated from c^*. If V^* has not been traversed, c^* is deposited to serve as a seed for V^*. Since V^* might contain more than one mesh component, more than one seed cell might be deposited before V^* is traversed. For this reason, all cells deposited must be processed when V^* is traversed.

After performing surface tracking on all overlapping voxels by traversing the space subdivision hierarchy, we retrieve a pair (V, c) from queue Q and perform surface tracking on V using c as a seed cell. When performing surface tracking on such voxels, we may still encounter voxels that need to be tracked again for an additional mesh component. If this is the case, the encountered voxel and its associated cell is put into queue Q for later processing.

In summary, the polygonization process can be described as follows:

```
for each overlapping voxel V { /* via hierarchy
  traversal */
  Obtain a seed cell c for V;
  SurfaceTracking(V, c);
}
while queue Q is not empty {
  Retrieve a pair (V, c) from the queue Q;
  Surface Tracking (V, c);
}
```

SurfaceTracking() is described in the following pseudocode:

```
SurfaceTracking(V, c){
  Classify vertices of c;
  Compute triangles in c;
  for all transversal faces of c
    if the adjacent cell c* has not been tracked
      if cell c* is outside voxel V
        if the adjacent voxel V* that contains c*
            has been traversed
          push (V*, c*) into queue Q;
        else deposit c* as a seed for V*;
      else SurfaceTracking(V, c*);
}
```

We now show that for each overlapping voxel the proposed boundary tracking scheme does not miss any mesh component, *i.e.* only one sequential search for a seed cell suffices for each overlapping voxel. If a component in a voxel V is missed, it must be an isolated object that either lies fully inside V or overlaps a voxel V' adjacent to V, as shown in Fig. 4. Moreover, inside V or V' there should exist components other than that missing component; otherwise, the missing component should be detected by a sequential search for a seed cell inside voxel V or V'. Such a configuration is impossible, since that isolated object should be bounded by an S-bound and hence voxel V and V' should be further subdivided. This contradicts the fact that both V and V' are leaf voxels of the space subdivision hierarchy. We have thus shown that for each overlapping voxel the proposed boundary tracking scheme does not miss any mesh component.

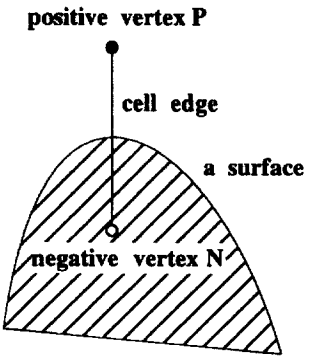
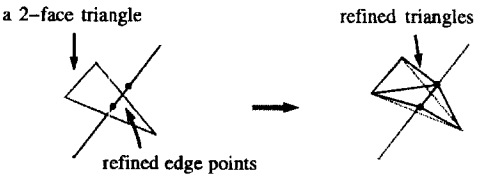


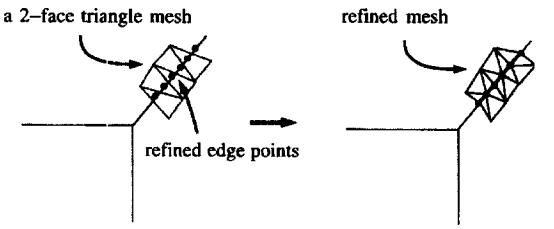
Fig. 6. An intersection on a transversal cell edge.

3.2.2. *Computing triangles in a cell*

3.2.2.1. *Classifying vertices.* Since the CSG tree within a leaf voxel is localized to a sub-CSG tree, to determine the *on*, *out*, or *in* status of a cell's vertex we classify the vertex against the sub-CSG tree associated with the voxel containing the vertex. For cells intersecting the boundary of the current voxel, a cell's vertices might lie on more than one adjacent voxel. When this is the case, sub-CSG trees associated with these adjacent voxels must be taken into account in the point classification. We will now present a method that can resolve this problem for the case in which a vertex v lies on the boundary of two adjacent voxels V_1 and V_2. For cases in which more than two voxels are involved, generalizations are straightforward. Let T_1 and T_2 be the sub-CSG trees of V_1 and V_2, respectively, and r_1 and r_2 be the classification results of v against T_1 and T_2, respectively. There are nine combinations for (r_1, r_2); among them (*on*, *on*), (*out*, *out*), and (*in*, *in*)



(a) Refining a 2-face triangle



(b) Refining a 2-face triangle mesh

Fig. 7. Recovering an edge.

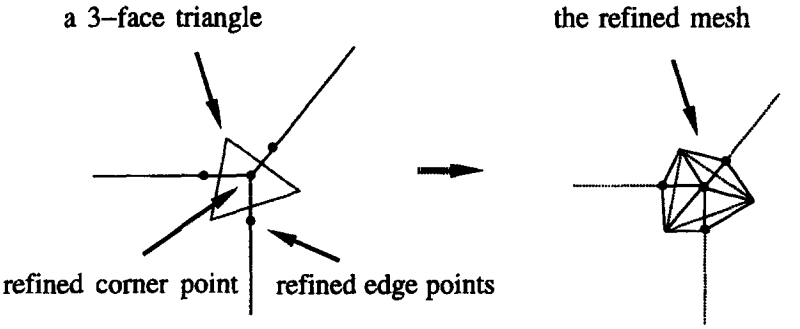


Fig. 8. Recovering a corner.

exhibit no conflict. The cases (*out*, *in*) and (*in*, *out*) are impossible, while (*out*, *on*) and (*on*, *out*) exhibit no conflict since *on* is considered as *out*. Only the case of (*on*, *in*) and (*in*, *on*) indicates a conflict and the result of *on* should be reported. Figure 5 depicts such an example, in which voxels V_1 and V_2 possess sub-CSG trees $T_1 = A$ and $T_2 = B$, respectively., and cell vertex v is classified against T_1 and T_2 as *in* and *on*, respectively.

For cells having vertices outside the voxel currently being traversed, we have to locate in which voxel the vertex lies. This search can be done by following the face pointer associated with the current voxel [2]. For each face F of the leaf voxel V we maintain a pointer pointing to the adjacent voxel that is on the lowest possible level of the hierarchy and which has a face that totally encloses F. When locating the voxel containing a cell vertex v from the current voxel V, we determine the face these two adjacent voxels share and then retrieve the face pointer. If the neighboring voxel is a leaf, we have found the next voxel. If the neighbor is an interior node, the point that we want to query is used to perform a downward search on the neighbor's subdivision hierarchy.

3.2.2.2. *Computing edge–surface intersections and normals.* After the cell's vertices are classified, an edge (P, N) with vertices of positive (*on*/*out*) and negative (*in*) value should report an intersection on it; as shown in Fig. 6. Such an intersection can be generally obtained by interpolation, as in [6]. The result, however, may be erroneous since the distance value on the cell's vertices is error-prone. Here, for a transversal edge (P, N), we determine a primitive whose boundary surface is transversal to the edge. There may be more than one such transversal primitive or surface, and it is reasonable to choose any one of them when the cell's size is sufficiently small. Let T_P and T_N be sub-CSG trees associated with voxels that contain P and N, respectively. The algorithm is as follows:

(1) For the positive vertex P, we derive two sets: S_P^-, the set of primitives of T_P for which P is classified as *in*, and S_P^+, the set of those for which P is classified as *on* or *out*. Similarly, we derive S_N^- and S_N^+ for vertex N.
(2) If T_P and T_N are identical, there will be at least one primitive in S_P^+ and S_N^- whose boundary face is transversal to the edge (P, N). So we select a primitive from $S_P^+ \cap S_N^-$.
(3) If T_P and T_N are different, two cases are possible: (a) If $S_P^+ \cap S_N^- \neq 0$, we select a primitive from $S_P^+ \cap S_N^-$. (b) If $S_P^+ \cap S_N^- = \phi$, two cases are possible:

   (i) T_P is empty: there will be at least one primitive in S_N^- for which P is classified as *out*. We select such a primitive.
   (ii) T_P is not empty: we obtain a primitive transversal to edge (P, N) as follows:

      A. Select a primitive in S_N^- for which P is classified as *out* or *on*.
      B. Select a primitive in S_N^+ for which P is classified as *in*.

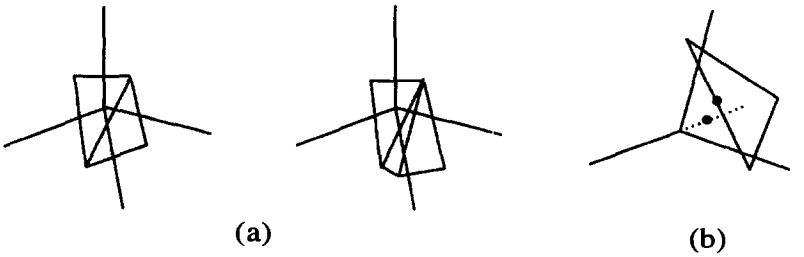


Fig. 9. Complex corner refinement.

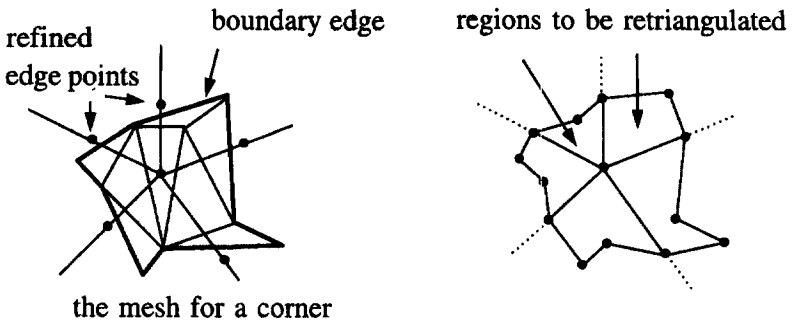


Fig. 10. A corner refinement.

C. Select a primitive in S_P^- for which N is classified as *out* or *on*.
D. Select a primitive in S_P^+ for which N is classified as *in*.

Whenever a primitive transversal to (P, N) is obtained, the intersection and the associated surface normal can be easily computed. One problem that requires special attention is the normal direction, since the difference operator will reverse the surface normal. We resolve this problem by propagating from the primitive upward toward the root of the respective sub-CSG tree and reversing the computed normal whenever a difference operator is encountered from the right descendant.

3.3. *Postprocessing: recovering edges and corners*

Edges and corners are often rounded by polygonizers based on vertex evaluation. Edges and corners are, however, important features that should be maintained for finer polygonization. Since triangle vertices are derived from exact primitive geometry, we are able to detect and refine those triangles that round the edges or corners. A triangle that rounds an edge or a corner can be detected, depending on whether its vertices are derived from two or three face surfaces. A triangle whose vertices are derived from 2 or 3 face surfaces is called a *2-face triangle* or a *3-face triangle*, respectively. A triangle edge whose vertices lie on 2-face surfaces is called a *2-face edge*. For a 2-face triangle that rounds an edge, we first determine its 2-face edges and then refine their middle points to the intersection of the corresponding face surfaces. After that, we remove those 2-face edges and finally connect the refined points and original vertices to form a new mesh, as shown in Fig. 7(a). For a 3-face triangle that rounds a corner bounded by three faces, we can first refine the middle points of its 2-face edges to the intersection of corresponding face surfaces, and in the meantime, refine the triangle's center to the corner point. After removing the 2-face edges and connecting the refined corner point, refined edge points, and original vertices, we form a new mesh, as depleted in Fig. 8. A corner formed by more than three faces is generally rounded by more than one 3-face triangle together with some 2-face triangles whose vertices are not on two adjacent faces; see Fig. 9(a). Even for a corner formed by three faces, refining only the 3-face triangle is sometimes erroneous since the middle point of a 2-face edge might be refined to a point that is outside the solid; as shown in Fig. 9(b).

To circumvent the problems and provide a uniform approach for both edge and corner refinement, for each corner we seek a mesh that consists of the corresponding 3-face triangles and their adjacent 2-face triangles. The centers of 3-face triangles will be refined to the corner point and middle points of boundary 2-face edges of the mesh refined to edges. The refined corner and edge points and original vertices on the mesh boundary are finally connected to form regions, which will be retriangulated; see Fig. 10.

The edges and corners are refined uniformly in the following manner. We retrieve the triangle list, from the head to the tail, until a 2-face or a 3-face triangle is found. If it is a 3-face triangle, we refine the corner and then recursively propagate the refinement along each edge branching from the corner. If it is a 2-face triangle, we regard that triangle as a starting triangle and propagate recursively to adjacent triangles along the edges until all connected edges are refined. On the way of edge refinement, we might visit a 3-face

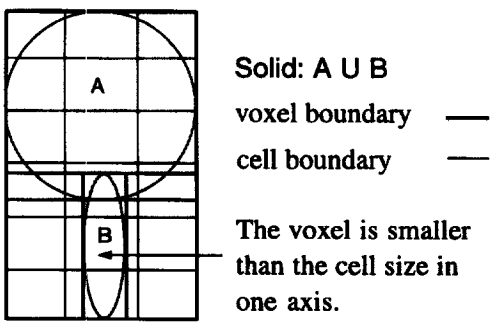


Fig. 11. A voxel is smaller than the cell size in some axis direction.

Table 2. Voxel classification result

	Model-1	Model-2	Model-3	Model-4	Model-5	Model-6
No. of inside voxels	0	0	0	0	2	5
No. of outside voxels	0	11	16	0	210	1229
No. of overlapping voxels	3	35	39	75	135	854

triangle, for which a recursive propagation for each branching edge will be performed. A triangle is deleted from the list once it is visited. Whenever a set of connected edges is refined, we continue to retrieve the remaining triangle list for a starting triangle for another set of connected edges. The whole refinement process is finished when the triangle list is empty. By doing this way, each triangle in the list will be visited only once and the edges and corners can be recovered in a uniform manner. The pseudocode for edge and corner refinement is as follows:

```
for each triangle T on the triangle list
if T is a 2-face triangle
   Refinement-propagation(2, T, NULL)
else if T is a 3-face triangle
   Refinement-propagation(3, T, NULL);
```

in which Refinement-propagation() is described in the following pseudocode:

```
Refinement-propagation(type, triangle T, already-
refined-vertex)

switch type
case 2:
   if there is a 3-face triangle T' adjacent to T
      then Refinement-propagation(3, T', NULL)
   else if already-refined-vertex is NULL
      Refine middle points of two 2-face edges
      else Refine the middle point of another 2-face
         edge;
   Put the pairs of adjacent triangle and the refined
      edge point to Seed-list;
   Retriangulation-for-edge;
   for each pair (T', p') in Seed-list
      Refinement-propagation(2, T', p');
   return;
case 3:
   Corner = Gather-corner-related-mesh(T,
      NULL);
   Search for boundary edges on Corner_mesh;
   for each boundary 2-face edge
      Perform edge refinement and put the pair of
         adjacent
      triangle and refined edge point into Seed-list;
   Retriangulation-for-corner;
   for each pair (T', p') in Seed-list
      Refinement-propagation(2, T', p')
   return;
```

Gather-corner-related-mesh(T, Point) receives input a 3-face triangle, T, and a point, Point, and returns a record containing two fields: the corner point, Corner_point, refined using T and the mesh,

Table 3. Experiment data

	No. of primitives	Cell size	Method used	Preproc. time	Polyg. time	Total time	G.F.	N_b	Refine. time	N_a
Model-1	2	0.30	CC	0.24	1.01	1.35	–	2864		
			BT	0.00	1.37	1.37	0.99	2864	0.06	3038
Model-2	9	0.20	CC	1.56	1.19	2.75	–	3752		
			BT	0.01	2.75	2.76	1.00	3754	0.41	5384
Model-3	11	0.22	CC	2.93	1.63	4.56	–	5028		
			BT	0.02	2.77	2.79	1.63	5039	0.65	7539
Model-4	18	0.18	CC	8.20	4.04	12.24	–	11184		
			BT	0.02	6.55	6.57	1.86	11099	0.78	14323
Model-5	44	0.23	CC	71.45	8.29	79.74	–	11448		
			BT	0.10	8.49	8.59	9.28	11479	1.69	17427
Model-6	265	0.23	CC	539.95	718.27	1258.22	–	96536		
			BT	0.75	79.01	79.76	15.78	96728	11.76	139452

Method used: CC—constructive cube, BT—proposed method.
Preproc. time (in seconds): distance evaluation time for CC or space subdivision time for BT.
Polyg. time (in seconds): marching cube time for CC or polygonization time for BT.
G.F.: gain factor on total time for mesh generation.
N_b: No. of triangles without refinement.
N_a: No. of triangles after edge/corner refinement.

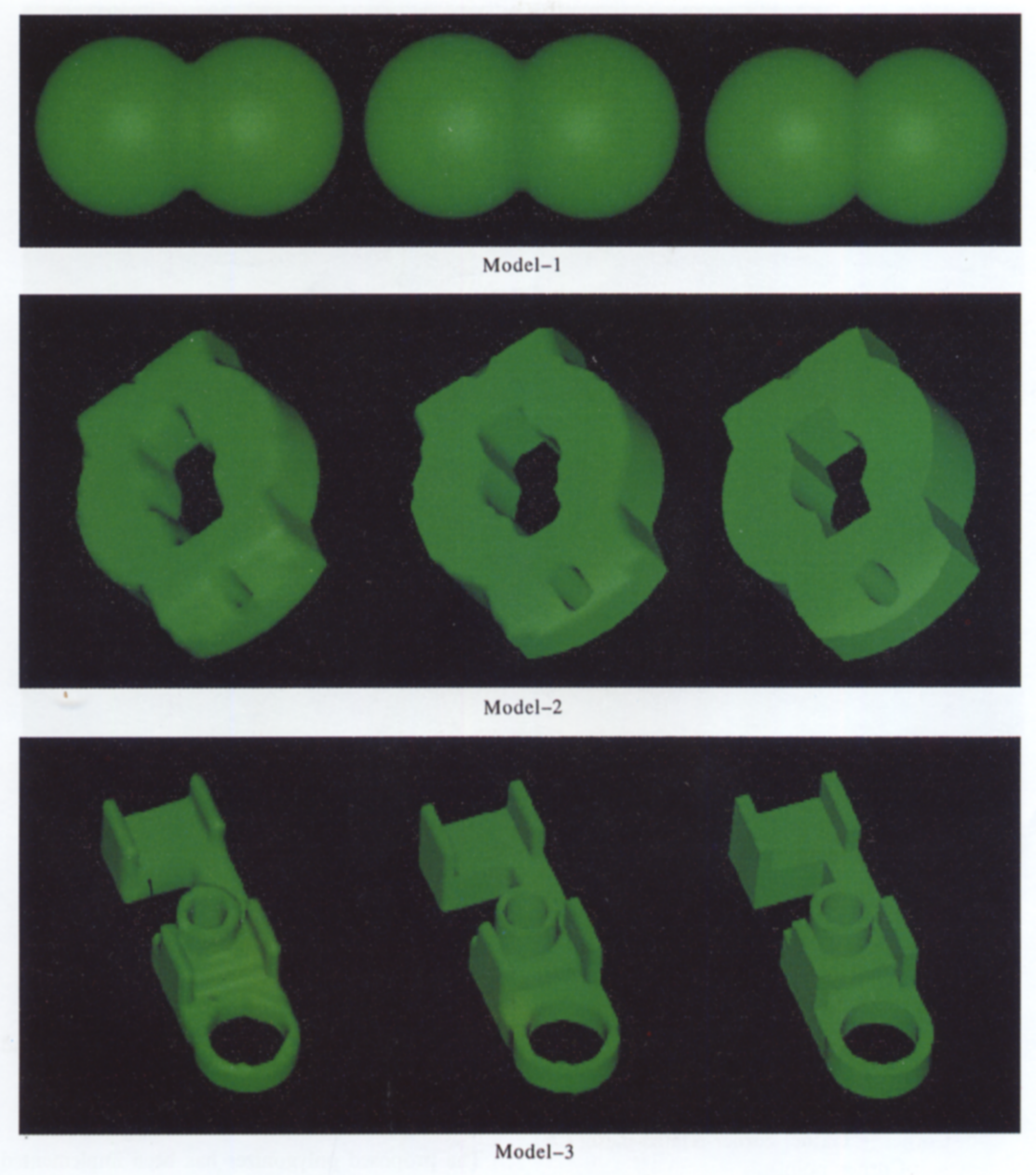


Fig. 12. Images of Model-1, Model-2, and Model-3.

Corner_mesh, that consists of the 3-face and 2-face triangles relevant to the corner point, Corner_point. The found 3-face triangles are those whose centers are refined to Corner_point within a user-prescribed tolerance while the found 2-face triangles are immediately adjacent to those 3-face triangles.

```
Gather-corner-related-mesh(T, Point)
  if Point = NULL
    then Corner_point = Refine-corner-vertex(T);
    else Corner_point = Point;
  Corner_mesh = T;
  for each unprocessed neighboring triangle T' of
          T {
    if T' is a 3-face triangle {
      Refined-point = Refine-corner-vertex(T');
      if Refined-point closes to Corner_point within
          a prescribed tolerance {
        Corner' = Gather-corner-related-mesh(T',
        Corner_point);
        Append Corner'_mesh to Corner_mesh;
        }
      }
    else if T' is a 2-face triangle {
      Append T' to Corner_mesh;
      for each unprocessed neighboring 3-face
        triangle T* of T' {
```

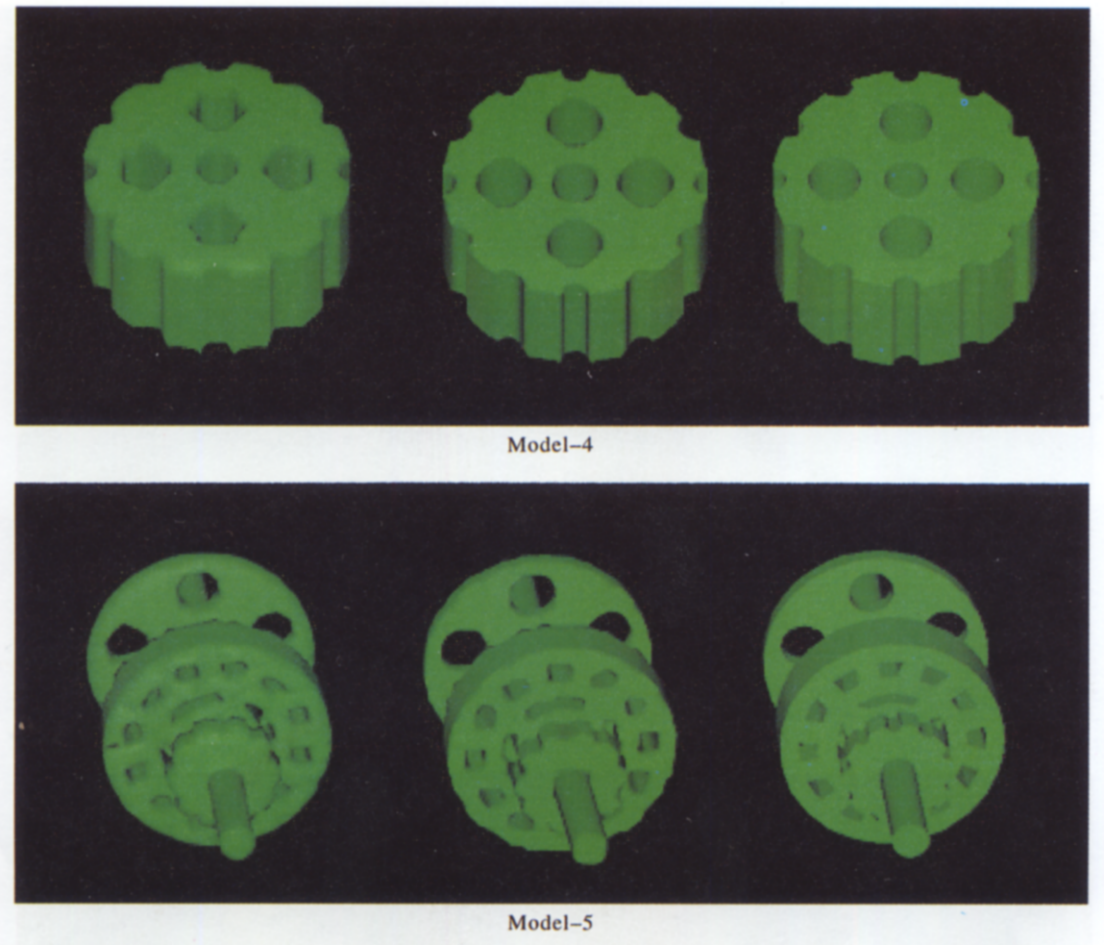


Fig. 13. Images of Model-4 and Model-5.

```
        Refined-point = Refine-corner-vertex(T*);
        if Refined-point closes to Corner_point within
              a prescribed tolerance {
          Corner* = Gather-corner-related-mesh(T*,
              Corner_point);
          Append Corner*_mesh to Corner_mesh;
          }
        }
      }
    }
    return (Corner),
```

Refining the middle point of a 2-face edge to the edge curve of the corresponding face surfaces is performed using the point refinement method, *Three-Plane-Method*, proposed in [11]. To refine the center point C of a 3-face triangle to a corner, we first refine C to a point P on the edge curve of two corresponding face surfaces, say S_1 and S_2, and compute the intersection point Q between the third surface S_3 with the tangent line of the edge curve at P. If all distances between Q and face surfaces S_1, S_2, and S_3 are within a user-specified tolerance, then we have found the corner Q; otherwise we set $C = Q$ and repeat the computation steps.

4. IMPLEMENTATION AND EXAMPLES

The proposed polygonizer has been implemented in C++ on a Silicon Graphics Indy workstation with a MIPS R4600PC CPU and 32MB RAM. Several examples have been tested, including six solids shown in Figs 12–14. For comparison purpose, we also implement Constructive cube method proposed by Breen [6]. Table 2 shows the distribution of inside, outside, and overlapping voxels results from our space subdivision hierarchy. Table 3 compares the performance of the Constructive cube method and the proposed method. The distance evaluation time for Constructive cube method increases very fast as the number of primitives increases while the space subdivision time for the proposed method remains negligible. Although in the proposed method only cells transversal to the solid's boundary are processed, its polygonization time is slightly larger than marching cube process in the Constructive cube method. This is because in the proposed method intersections between solid's boundary and cell's

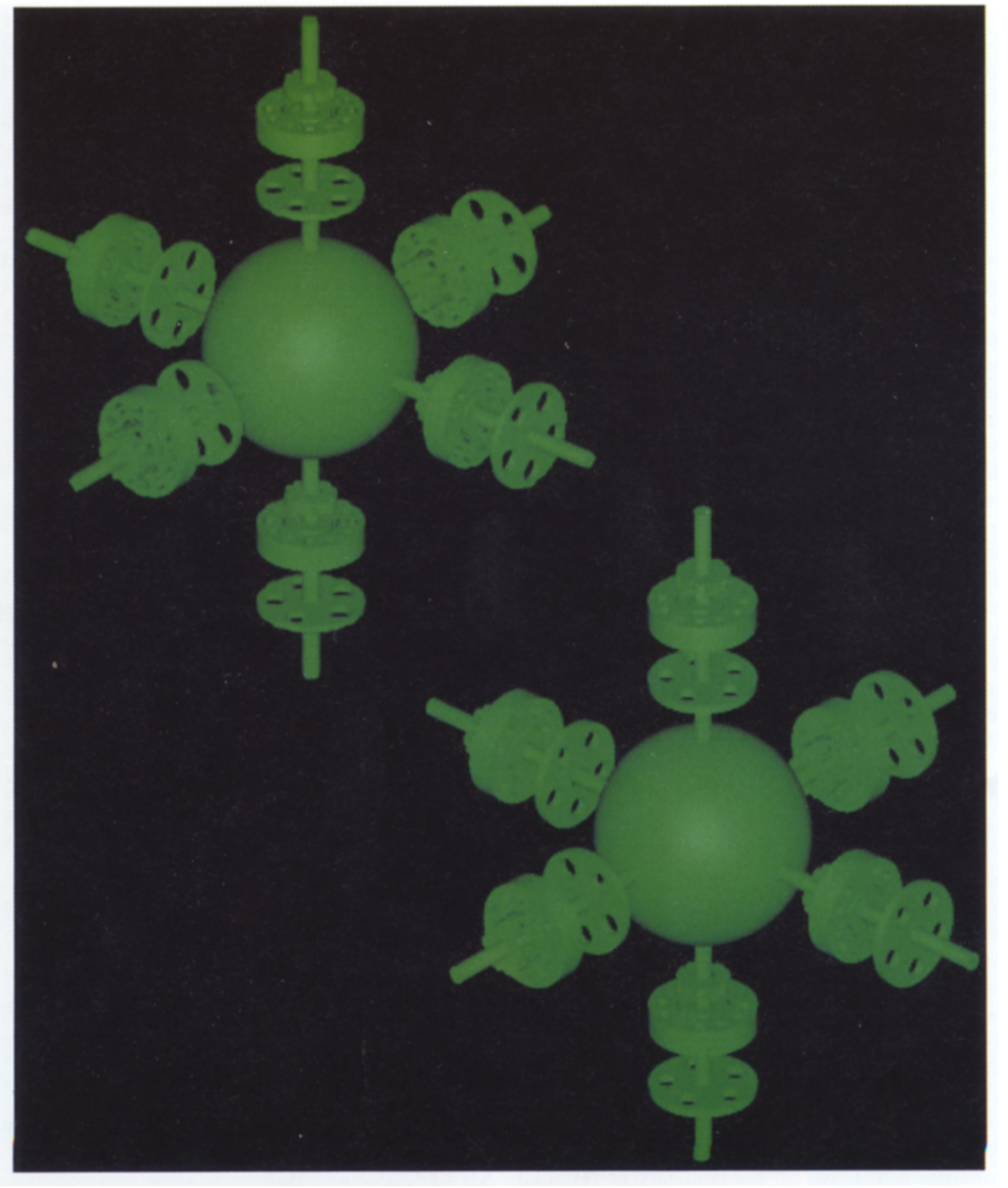


Fig. 14. Images of Model-6.

edges are computed exactly using primitive geometry, rather than interpolation-based computation as in Constructive cube method. In consequence, the proposed polygonizer produces finer meshes than the Constructive cube method, as depicted in Figs 12–14, in which the left are images of meshes result from the Constructive cube method, while the middle and the right images are produced by the proposed polygonizer without and with edge/corner refinement, respectively. Model-6 shown in Fig. 14 is composed of a sphere and six objects of Model-5 in different positions and orientations. The image of Model-6 produced by Constructive cube method is not shown due to the space problem. We have found in the proposed polygonizer more than 80% of the polygonization time is devoted to the boundary–edge intersection computation. The gain factor for mesh generation time increases from 0.99 to 15.77 as the number of primitives increases from 2 to 265. The polygonization time for the proposed method seems to increase very slowly as the number of primitives increases. One thing worth mentioning is that in order to reduce the possibility and time for sequential search for a seed in some overlapping

voxels, once a seed is found in the first overlapping voxel we propagate the voxel traversal to adjacent overlapping voxels in a recursive manner. By doing this, only one sequential search for a seed is necessary for a connected object. The proposed edge/corner refinement seems very effective in both the computational efficiency and the quality, as we can see from the timing table and images.

When CSG solids are polygonized based on vertex evaluation, small features may be lost. Such a loss of small features is in general extremely difficult to detect. Due to the properties of the space subdivision we use, the boundary of a CSG solid is completely bounded by the overlapping voxels [2]. Hence with our polygonizer a lost small feature can be detected and recovered, since it will be contained in a voxel with a size smaller than the cell's size in some axis directions. This is depicted in Fig. 11, in which the vertices of the cell are considered to be *out* with respect to the sub-CSG tree associated with the voxel. A regular grid with a smaller cell size will improve this problem, but at considerable additional cost. A feasible approach is to maintain the grid with a reasonable cell size and adoptively subdivide a cell when its size is larger than the size of an intersecting voxel in some axis directions. Within the subdivided cells, ordinary surface tracking can be performed. Care must be taken to remove cracks due to different levels of subdivision in adjacent cells.

5. CONCLUSION

We have proposed and implemented an efficient polygonizer for CSG solids. The method directly triangulates the boundary surface of CSG solids without converting the CSG representation to a boundary representation. For a given CSG tree, a regular grid is constructed that is embedded on the S-bounds of the solid, which is subdivided based on the S-bounds of internal nodes and primitives of the CSG tree. Such space subdivision greatly reduces the localized CSG tree associated with each leaf voxel and in turns speeds up point classification process. In the polygonization process, only voxels of the space subdivision that overlap the boundary of the solid are traversed and within such voxels only cells transversal to the solid's boundary are tracked. The surface-edge intersection and normal, which are usually interpolated in the standard marching cube algorithm, are computed directly from the exact boundary surface of the CSG solid. In consequence, edges or corners, which are generally rounded by the vertex-evaluation based polygonizers, can be easily recovered by the proposed method. According to our experience, the computing time seems to increase very slowly when the number of primitives increases. The proposed structure is capable of detecting and recovering small features that are generally difficult to be dealt with by polygonizers based on vertex evaluation.

REFERENCES

1. Glassner, A. S., Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 1984, **4**(10), 15–22.
2. Chuang, J. H. and Hwang, W. J., A new space subdivision for ray tracing CSG solids. *IEEE Computer Graphics and Applications*, 1995, **15**(6), 56–63.
3. Kay, T. L. and Kajiya, J. T., Ray tracing complex scenes. *Computer Graphics*, 1986, **20**(4), 269–278.
4. Krishnan, S. and Manocha, D., Efficient representations and techniques for computing B-reps of CSG models with NURBS primitives. In *CSG '96 Proceedings*, Information Geometers Ltd., Winchester, UK, April 1996, pp. 101–122.
5. Hoffmann, C. M., *Geometric and Solid Modeling, An Introduction*. Morgan Kaufmann, San Francisco, CA, 1989.
6. Breen, D. E., Constructive cubes: CSG evaluation for display using discrete 3-D scalar data sets. In *Proceedings of Eurographics*. North-Holland, Amsterdam, 1991, pp. 127–142.
7. Tobler, R. F., Galla, T. M. and Purgathofer, W., ACSGM—an adaptive CSG meshing algorithm. In *CSG '96 Proceedings*. Information Geometers Ltd., Winchester, UK, April 1996, pp. 17–31.
8. Lorensen, W. E. and Cline, H. E., Marching cubes: a high resolution 3-D surface construction algorithm. *Computer Graphics (Proceedings of Siggraph)*, 1987, **22**(4), 163–169.
9. Wyvill, G., Kunii, T. L. and Shirai, Y., Space division for ray tracing in CSG. *IEEE Computer Graphics and Applications*, 1986, **6**(4), 28–34.
10. Cameron, S., Efficient bounds in constructive solid geometry. *IEEE Computer Graphics and Applications*, 1991, **21**(4), 68–74.
11. Barnhill, R. E. and Kersey, S. N., A marching method for parametric surface/surface intersection. *Computer Aided Geometric Design*, 1990, **7**(1-4), 257–280.