Efficient Octree Conversion by Connectivity Labeling

Markku Tamminen
Laboratory of Information Processing Science
Helsinki University of Technology, 02150 Espoo 15, Finland

Hanan Samet
Computer Science Department
University of Maryland, College Park, MD 20742

ABSTRACT

We present an algorithm for converting from the boundary representation of a solid to the corresponding octree model. The algorithm utilizes an efficient new connected components labeling technique. A novelty of the method is the demonstration that all processing can be performed directly on linear quad- and octree encodings. We illustrate the use of the algorithm by an application to geometric mine modeling and verify its performance by analysis and practical experiments.

CR Categories and Subject Descriptors: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling -- Solid and Object Representations, Geometric Algorithms, Languages, and Systems

General Terms: Algorithms, Data Structures, Performance

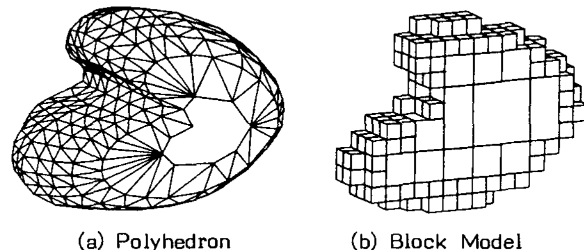
Additional Key Words and Phrases: Image Processing, Octree, Conversion

1. Introduction

A solid modeler is a system for manipulating spatially complete data on the geometric form of three-dimensional solid objects. Each modeler uses one or more solid representation schemes and conversion algorithms between representations have become increasingly important (Requicha and Voelcker 1983).

The main representation of constructive solid geometry (CSG) modelers is a tree of set operations and rigid motions applied to primitive building blocks while boundary representation modelers define a solid by a collection of faces, edges and vertices. A radically different approach, receiving increasing attention, is exemplified by the octree scheme (Meagher 1982a) of hierarchic spatial enumeration. It divides a region of space recursively into eight cubic parts until each one is simple (empty or solid) or a fixed maximal resolution is reached.

Relaxing some of the assumptions of the octree model we shall more generally consider block models or three-dimensional image trees. Figure 1-1 shows a polyhedron with 588 faces (a) and part of its block model formed at resolution $2^4 \times 2^4 \times 2^4$ (b).



(a) Polyhedron (b) Block Model

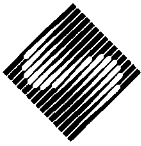
Figure 1-1
Polyhedron and Part of Corresponding Block Model

The methods used for analyzing the integral properties of solids and for converting between representations depend intimately on the underlying solid representation as discussed by Lee and Requicha (1982), whose algorithm converts efficiently from the CSG scheme into a block model. We shall present a technique for converting from a boundary representation into a block model. This topic has not been much treated in the literature: neither Meagher (1982b) nor Lee and Requicha (1982) nor Requicha and Voelcker (1983) report efficient solutions. Ourselves, we have heretofore used an algorithm reported in (Tamminen et al. 1984).

Block models, and octrees in special, are direct derivatives of the two-dimensional quadtree representation of images, originally introduced by Klinger (1971). See (Samet 1983) for a comprehensive survey. There exist many different encodings of quadtrees, octrees and similar hierarchical data structures. The explicit pointer based tree representation of a block model is not well suited for external storage and ordinarily requires 20 to 40 times as much space as the most compact linear tree representations (Kawaguchi

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0-89791-138-5/84/007/0043 $00.75



and Endo 1980, Meagher 1982a, Gargantini 1982b, Yamaguchi et al. 1983, Tamminen 1984).

The space complexity of an octree corresponding to a general polyhedron is proportional to the surface area of the polyhedron measured at the chosen resolution (Meagher 1982b). Even at moderate resolution the block model may contain hundreds of thousands of nodes. Thus, it is not always sufficient to formulate a general conversion algorithm; it may be as important that it supports a linear tree representation.

In (Samet and Tamminen 1983) we have presented a new technique of determining geometric properties, such as the perimeter, of linear quadtree encodings and in (Samet and Tamminen 1984) the method has been applied to 3-dimensional connected components labeling. Now we show how the same approach can be used in the conversion problem. The demonstration that all phases of our algorithm can operate directly on linear tree representations without utilizing explicit neighbor finding techniques (Samet 1981) is one of its main interests.

The practical framework of our research is the Geometric Workbench (Mantyla and Sulonen 1982), an experimental solid modeler constructed at the Helsinki University of Technology. The conversion problem originates from applying GWB to geometric mine modeling (Karonen et al. 1983). Through the conversion program, GWB has been connected to the octree modeler OCTGRAS (Yamaguchi et al. 1984), made available to us through co-operation with the Kunii Laboratory of the University of Tokyo.

We first describe briefly the application and previous conversion efforts. Section 3 defines linear image tree representations. In Section 4 we formulate the new conversion algorithm. Its performance is analyzed in Section 5 and finally the claims verified by experimental results.

2. Background

2.1. Why Conversions

We first briefly discuss an example application, geometric mine modeling, to demonstrate why conversions are needed.

The methods of this article were first motivated by an experimental geometric mine modeling system implemented together with a Finnish mining company, Outokumpu Oy. The system operates on boundary models representing entities, such as ore bodies, tunnels, or planned excavations. Three dimensional solids describing ore bodies are constructed by connecting two-dimensional sections by a three-dimensional boundary (Figure 1-1a).

The principal analysis task of mine design consists of intersecting a planned excavation with an ore body and determining the amount of minerals and side material thus formed (Figure 2-1). This requires a volume integration of the type:

$$\text{(I)} \qquad \int_S f(x,y,z)dV,$$

where f(x,y,z) is the (unknown) function describing mineral content at each point of space and S is the solid modeling the extracted part of the ore body.

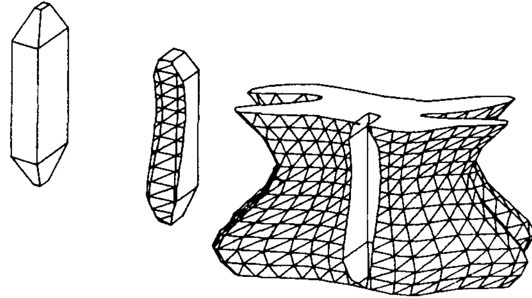


Figure 2-1
Modeling Excavation by a Boolean Set Operation

The function f is empirical, in that it must be estimated at each relevant point separately by geostatistical methods, or kriging (Journel and Huijbregts 1978). Therefore , discrete approximations (Lee and Requicha 1982) must be employed to evaluate (I). The determination of one value of f(x,y,z) necessitates a spatial search among the hundreds or thousands of drill samples and is an expensive operation.

Geostatistics is applied to the boundary representation of an ore body by first converting into a block model. Each block is estimated separately and the results summed for a total value.

2.2. Existing Algorithms

There do not exist many publications on converting a boundary representation into an octree. However, such a component is used in some practical systems (Meagher 1983, Requicha and Voelcker 1983). The algorithms utilized can be divided into two groups: those based on connectivity (Meagher 1983) and those based on explicit block classification (Tamminen et al. 1984). A third approach would be to make M sections of the polyhedron, convert each one of them into a quadtree at resolution M^2 and combine the results into an octree at resolution M^3 by the algorithm of Yau and Srihari (1983).

Conversion algorithms based on connectivity reflect the structure of the quadtree algorithm of Samet (1980). They first determine the volume elements lying on the boundary of the solid. The partial tree thus formed is then traversed and each unclassified leaf is determined to be empty or full by inspecting its neighbors. This approach can also be implemented by using a connected components labeling algorithm. For efficiency, the method has required an explicit tree representation.

In the geometric mine modeling system we have utilized an algorithm where each leaf of the block model is explicitly classified by a point-in-polyhedron test. The main computational operations of this algorithm are to determine whether a block intersects the boundary of the solid and, if so, whether it is contained in it. These operations typically have to be performed tens of thousands of times with the boundary model containing hundreds of faces. The technique has been made efficient by using a spatial in-

dex based on the EXCELL method (Mantyla and Tamminen 1983). In practice computation time is almost independent of the number of polygons defining the polyhedron.

This method has not been a main bottleneck of the mine modeling system. However, with the results of Samet and Tamminen (1983), implementing the connectivity approach has become justified.

3. Binary Image Trees

Solid modeling by spatial enumeration is closely related to three-dimensional image processing, which will be reflected in our terminology. This section pertains to both two- and three-dimensional images but, for conciseness, we present mainly the three-dimensional case.

3.1. Definitions

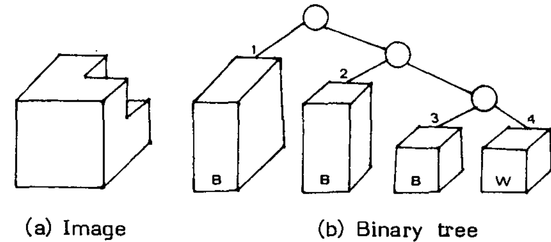
We shall consider two- (2D) and three dimensional (3D) binary images (i.e., 2- or 3-dimensional matrices of pixels, respectively voxels) and speak of the pixels and voxels as image elements. We use the same term also for the homogeneous blocks (leaves), which are the basic elements of quadtrees and octrees. Let $M = 2^n$ describe the resolution of the image so that the total number of pixels (voxels) is M^2 (M^3).

An octree is defined as a recursive 8-ary partition of a three-dimensional image into octants until homogeneous blocks (BLACK or WHITE) are reached (Srihari 1981, Meagher 1982a, Jackins and Tanimoto 1980,1983). A three-dimensional binary image tree is formed exactly analogously but by dividing only in two parts at each level of recursion. We assume the first partition to be in the x-direction with the y-, z- and x-directions alternating thereafter. Figure 3-1 illustrates this concept. In the x-partition we postulate the left subtree to correspond to the western (W-) half of the image; in the y-partition it corresponds to the S-half. Let us similarly speak of the lower (L) and upper (U) halves of the z-partition.

A node in a 3D binary image tree has six sides (W, E, S, N, L, U) and a neighbor node (of equal size), in each of these directions. In the ordering of nodes induced by a preorder traversal of the binary tree all the nodes in a W- or S- or L-neighbor of a given node come before that node. We utilize binary image trees mainly because tree traversal algorithms become somewhat simpler than for octrees.

3.2. Representations

We use a linear tree representation that is based on the preorder traversal of the binary image tree. The traversal yields a string over the alphabet "(", "B", "W" corresponding respectively to internal nodes (GRAY), BLACK leaves, and WHITE leaves. We call this string a DF-expression as Kawaguchi and Endo (1980) do in the case of quadtrees. A different but related representation is the linear octree of Gargantini (1982). For the image of Figure 3-1 the DF-expression becomes (B(B(BW. Its most straightforward bit-encoding requires two bits per node both for octrees and binary image trees. Explicit pointer



(a) Image (b) Binary tree

Figure 3-1
Three-dimensional Binary Image Tree

based representations ordinarily require at least one computer word per node (Meagher 1982b).

In (Tamminen 1984) we have reported methods of compacting the DF-expression. First of all, encode "(" by "1" and "B" and "W" by "01" and "00", respectively. Further, at the lowest level of a condensed tree there may exist only two types of node pairs, "BW" and "WB". Thus these pairs may be encoded by "0" and "1", respectively. In practice the above method has required about one bit per node of a three-dimensional binary image tree.

A binary image tree always contains at most as many leaves (but often more nodes) than the corresponding octree. For instance, at resolution $M = 2^6$ the condensed binary tree of the surface of a unit sphere contains 25600 leaves while the corresponding octree has 43800 leaves.

4. Conversion Algorithm

4.1. General Outline

In Figure 4-1 we give the outline of an algorithm for converting from a boundary representation to a 3D image tree. The method supports multiple solids without interior voids, but the 3D outside of the solids must be connected.

First, in procedure COMBINE3(), each face is separately converted into a linear image tree representation. The trees are recursively OVERLAY'ed in pairs to give the tree of the whole boundary. In the second phase - FILL3() - the image tree is traversed and its WHITE components, which are not connected to the outside of the image are extracted and changed to BLACK as described in the next section. As there is not enough space for detailed algorithms of all the (simple) subroutines of Figure 4-1 we only present their outlines.

OVERLAY() forms the boolean union of two (linear) binary image trees by traversing them synchronously according to the following rules:

(1) If either of the nodes is BLACK the resulting node is BLACK. The other subtree is skipped (by sequential traversal).

```
procedure BR_TO_BLOCKS3();
/* Convert boundary representation defined by face-
array FACES into binary image tree at resolution M
= 2^n.  */
begin
    global value integer M,NFACES;
    global pointer face array FACES[0:NFACES-1];
    global pointer nodelist DF; /* DF-expression */
    DF <- COMBINE3(0,NFACES-1);
    FILL3(); /* see Section 4.2 */
end;

pointer nodelist procedure COMBINE3(N1,N2);
/* Convert separately faces with indices between N1
and N2 to image trees and combine results into a
tree of the corresponding part of the boundary.  */
begin
    global value integer NFACES;
    global pointer face array FACES[0:NFACES-1];
    value integer N1,N2;
    if N2 - N1 > 1 then
        return(OVERLAY(COMBINE3(N1,(N1+N2)/2),
                       COMBINE3((N1+N2)/2+1,N2)));
    else if N2 - N1 = 1 then
        return(OVERLAY(CONVERT3(FACES[N1]),
                       CONVERT3(FACES[N2])));
    /* CONVERT3() converts one face */
    else return(CONVERT3(FACES[N1]));
end
```

Figure 4-1
Conversion Algorithm Outline

(2) If either of the nodes is WHITE the other subtree is copied to the result (by sequential traversal).

(3) If both nodes are GRAY the result is also GRAY.

(4) Replace recursively (BB by B and (WW by W.

CONVERT3() converts one face with plane equation

$$P(x,y,z) = ax + by + cz + d = 0$$

into a binary image tree as follows:

(1) Choose a projection plane, say xy, so that the remaining coefficient (c) has maximal absolute value.

(2) Form the 2D binary image tree TWODT of the projection of the face on the xy-plane by procedure BR_TO_BLOCKS2().

(3) The rest of the conversion is performed similarly to forming the image tree of the whole plane P(x,y,z) = 0, except that nodes, whose xy-projection is WHITE in TWODT, become WHITE in the result. The universe is halved recursively by planes alternatingly perpendicular to the x-, y-, and z-axes while keeping track of the minimum and maximum values of P(x,y,z) in each block thus formed. To each block corresponds a node N2 of TWODT so that the block can be classified as WHITE, BLACK, or GRAY as follows:

- if N2 is WHITE, the block is WHITE
- if zero does not lie between the minimum and maximum of P(x,y,z) in the block, the block is WHITE and N2 is skipped
- if the block is at voxel level and N2 is BLACK then the block is BLACK (division continues to voxel level on a face)
- otherwise the block is GRAY and is further subdivided.

(4) Replace recursively (BB by B and (WW by W.

The recursive halving directly produces the desired DF-expression.

BR_TO_BLOCKS2() forms the 2D image tree of a polygon. For simplicity we have implemented it completely analogously to BR_TO_BLOCKS3():

(1) Each edge of the face is converted into a 2D image tree by CONVERT2() similarly to the method applied in CONVERT3().

(2) The trees of the edges are recursively OVERLAY'ed in pairs by COMBINE2().

(3) The WHITE components of the 2D image not connected with the outside are changed to BLACK by FILL2(). (If necessary, holes within a face are treated by dividing the face into simply connected parts.)

The main virtue of CONVERT3() is that, to classify a block, we do not have to perform any point-in-polygon test. Also, the P(x,y,z)-range within each block can be efficiently computed during the recursive subdivision and no sorting is required to arrive at the correct DF-order of the blocks.

As a result of providing all xy-information in the 2D image tree, some spurious BLACK leaves may result when compared to the exact face/voxel -intersection tests. This is not serious considering the overall nature of the block model approximation. The choice of the projection plane minimizes the occurrencies of this event while guaranteeing that the inside of a solid is never connected with the outside.

The boundary conversion method described above has been satisfactory, even though we chose it mainly for its simplicity. We do not want to emphasize it because other, potentially more efficient, techniques can be imagined and combined with the core of our approach, described in the next section.

4.2. Connectivity Labeling

As discussed in Section 2.2 a variant of connected components labeling can be utilized in block model conversion. We show how it can be applied to linear tree representations.

Two elements of a 3D image are called (face-) connected to each other if they share a boundary (called adjacency) with non-zero area. Labeling the connected components of a binary image is ordinarily defined as transforming it into a symbolic image in which every maximally connected subset of BLACK elements is labeled by a distinct positive integer. However, in our case the image elements intersecting the boundary of the solid are BLACK and we want to extract and change to BLACK the WHITE components not connected to the outside of the image.

Connected components labeling can be performed by the union-find algorithm (Tarjan 1975). At the start each image element is assumed to form a separate component. The final components are determined by processing once each adjacency between image elements. For each relevant adjacency we must determine the putative components of the two elements (find). If they differ, they are combined (union).

The above algorithm, applied to a DF-expression, must be able to determine adjacencies as the tree is traversed in the fixed order. When processing a node we know that its W-, S-, and L-neighbors have been processed. Therefore there must exist data structures to record the information of respectively the E-, N-, and U-sides of the processed part. Let us call these data structures the active yz-, xz-, and xy-borders. They consist of active face elements. (In the rest of this section "face" means an active face of the above borders, not a face of the solid.)

The main change compared to the two-dimensional connected components algorithm reported in (Samet and Tamminen 1983) is that there are now three active borders, instead of two and that the size of a border element is defined as its area, not width. Further, the active borders can be represented as linked lists (instead of arrays), which is most important in the three-dimensional case. See (Samet and Tamminen 1984) for more details on connectivity labeling.

We give the filling algorithm in three parts. In the main program (Fig. 4-2) the three face element lists are first initialized so that each contains one WHITE face element of size M^2. The solid can be imagined as situated in the positive octant of coordinate space with all the other octants having been processed and WHITE. This mirrors the state of the active borders at the start of processing any node: its W-, S-, and L-neighbors have been processed and their color and component information is contained in the active border. Then procedure TRAVERSE() (Fig. 4-4) is called to traverse the DF-expression of the binary image tree. Finally PHASEII() traverses the tree once more. For each WHITE leaf it checks whether the leaf is in the component of the outside. If not, its color is changed to BLACK.

The main function of TRAVERSE() is to provide, at each call to its sub-procedures, a pointer to the parts of the active face element lists bordering that subtree. It calls itself twice recursively at each internal node. At each leaf node it calls procedure INCREMENT() three times to perform the actual updating of the active borders and the connected components. If a WHITE leaf is not identified with any existing component then a putative new component is formed. Labels of WHITE leaves are stored for processing by PHASEII(). After processing a leaf each list of active face elements is advanced to the element following it.

To illustrate the working of the algorithm we show, in Figure 4-3, the state of the active xy-border and the start of the sublist XYL when entering and leaving TRAVERSE() at each of its calls corresponding to leaf nodes of Figure 3-1.

```
procedure FILL3();
/* First compute connected WHITE components of a
binary tree of an M by M by M (M = 2^n) three-
dimensional image represented by preorder traversal
DF. Then change components not connected with out-
side of image to BLACK in PHASEII(). Each active
border surface xy, xz and yz is represented as a
linked list of records of type facelist, which contain
pointers to the active faces comprising the border.
Each active face is represented as a record of type
face with four fields SIZ, LAB, COL, and CRD,
which give respectively, the size (area), the com-
ponent label, the color, and the value of the third
coordinate (z for an xy border) of a face. A record
of type facelist has two fields, DATA and NEXT,
containing respectively, a pointer to a face and a
pointer to the next element in the list. */
begin
  global value integer M;
  global value pointer nodelist DF;
  pointer facelist XYL, XZL, YZL; /* borders */
  pointer face XY,XZ,YZ;
  XYL <- create(facelist); XZL <- create(facelist);
  YZL <- create(facelist);
  DATA(XYL) <- XY <- create(face);
  DATA(XZL) <- XZ <- create(face);
  DATA(YZL) <- YZ <- create(face);
  SIZ(XY) <- SIZ(XZ) <- SIZ(YZ) <- M*M;
  LAB(XY) <- LAB(XZ) <- LAB(YZ) <- outside;
  COL(XY) <- COL(XZ) <- COL(YZ) <- WHITE;
  CRD(XY) <- CRD(XZ) <- CRD(YZ) <- 0;
  if not empty(DF) then begin
        TRAVERSE(M,M,M,XYL,XZL,YZL);
        PHASEII() /* change inside to BLACK */
  end
end;
```

Figure 4-2
Main Procedure for Filling Inside

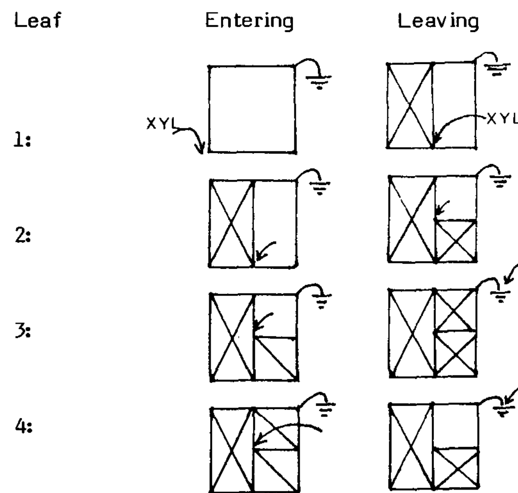


Figure 4-3
State of XYL at Each Call to TRAVERSE()

The purpose of procedure INCREMENT() (Figure 4-5) is to process all the active face elements bordering a face of a new leaf. Whenever an adjacency between WHITE faces is encountered, the connected components information is updated. Processing divides into three cases. In each of them INCREMENT() per-

```
procedure TRAVERSE(SX,SY,SZ,XYL,XZL,YZL);
/* Process SX by SY by SZ segment of image where
DF presents the preorder traversal of its binary tree.
XYL, XZL, and YZL are pointers to the lists of ac-
tive faces on the xy, xz-, and yz-borders of this
part of the image. Once the three faces of a leaf
that are adjacent to the active borders have been
processed, XYL, XZL, and YZL are advanced to point
to the portion of the active border that is adjacent
to the image element to be processed next. The list
LL stores the putative labels of WHITE nodes for
PHASEII(). */
begin
  value integer SX,SY,SZ;
  reference pointer facelist XYL,XZL,YZL;
  global pointer nodelist DF;
  global pointer labellist LL;
  pointer facelist T; /* auxiliary */
  pointer node L;
  L <- create(node);
  COL(L) <- next_node(DF);
  if COL(L) = GRAY then begin
    if SX = SZ then begin /* partition on x */
      T <- YZL; /* save start of yz border */
      TRAVERSE(SX/2,SY,SZ,XYL,XZL,YZL);
      TRAVERSE(SX/2,SY,SZ,XYL,XZL,T)
    end
    else if SZ = SY then begin /* on y */
      T <- XZL; /* save start of xz border */
      TRAVERSE(SX,SY/2,SZ,XYL,XZL,YZL);
      TRAVERSE(SX,SY/2,SZ,XYL,T,YZL)
    end
    else begin /* partition on z */
      T <- XYL; /* save start of xy border */
      TRAVERSE(SX,SY,SZ/2,XYL,XZL,YZL);
      TRAVERSE(SX,SY,SZ/2,T,XZL,YZL)
    end
  end
  else begin /* A leaf node. */
    LAB(L) <- unknown;
    INCREMENT(L,XYL,SX*SY,SZ); /* xy- border */
    INCREMENT(L,XZL,SX*SZ,SY); /* xz- border */
    INCREMENT(L,YZL,SY*SZ,SX); /* yz- border */
    if COL(L) = WHITE then begin
      if LAB(L) = unknown then /* new label */
          LAB(L) <- create(label);
          /* update active borders with label: */
      LAB(DATA(XYL)) <- LAB(L);
      LAB(DATA(XZL)) <- LAB(L);
      LAB(DATA(YZL)) <- LAB(L);
      add_to_list(LL,LAB(L)) /* for PHASEII() */
    end
    XYL <- NEXT(XYL); /* advance lists */
    XZL <- NEXT(XZL); YZL <- NEXT(YZL)
  end
end;
```

Figure 4-4
Tree Traversal

forms the necessary union operations and updates the active border as follows with the face of the new leaf:

(1) The entering face is larger than the corresponding first element of the active border. Neighboring face elements are determined from the size (area) of the new face. The new face replaces the last neighboring element and all others are disposed of.

(2) The entering face is equal in size with the first border element, which it replaces.

(3) The entering face is smaller than the first border element, which it replaces. A new active face is created to account for the rest of the old border element.

Finally the data of the border element corresponding to the new face are updated. For simplicity we have omitted the disposal of active face elements touching the outside of the image.

```
procedure INCREMENT(L,FL,S,W);
/* Process a leaf L of area S in the present direc-
tion (xy, xz, or yz) and width W in the perpendicular
direction. The leaf is adjacent to the first element
of the border represented by FL, pointer to a list of
active faces. See (Sedgewick 1983) for the imple-
mentation of union(), a combined find and union
operation. */
begin
 value pointer node L;
 value pointer facelist FL;
 value integer S,W;
 global value integer M;
 integer I; /* auxiliary */
 pointer facelist P,Q; /* auxiliary */
 if S > SIZ(DATA(FL)) then begin /* case 1 */
   I <- 0; P <- FL;
   while I < S do begin /* all bordering elements */
     if COL(L) = WHITE
       and COL(DATA(P)) = WHITE then
           LAB(L) <- union(LAB(L),LAB(DATA(P)));
     I <- I + SIZ(DATA(P));
     P <- NEXT(P)
   end;
   Q <- NEXT(FL); NEXT(FL) <- P; /* delete and */
   facelist_dispose(Q,P) /* reclaim storage for
   elements from Q up to but not including P */
 end
 else begin /* cases 2 and 3 */
   if COL(L) = WHITE
     and COL(DATA(FL)) = WHITE then
       LAB(L) <- union(LAB(L),LAB(DATA(FL)));
   if S < SIZ(DATA(FL)) then begin /* case 3 */
     P <- create(facelist); /* new element = */
     DATA(P) <- create(face); /* rest of old one */
     SIZ(DATA(P)) <- SIZ(DATA(FL)) - S;
     COL(DATA(P)) <- COL(DATA(FL));
     LAB(DATA(P)) <- LAB(DATA(FL));
     CRD(DATA(P)) <- CRD(DATA(FL));
     NEXT(P) <- NEXT(FL);
     NEXT(FL) <- P; /* insert into list */
   end
 end;
 SIZ(DATA(FL)) <- S; /* update first element */
 COL(DATA(FL)) <- COL(L);
 CRD(DATA(FL)) <- CRD(DATA(FL)) + W;
 if CRD(DATA(FL)) = M /* touches outside */
   and COL(L) = WHITE then
     LAB(L) <- union(LAB(L),outside)
end;
```

Figure 4-5
Processing one Side of a Leaf

5. Analysis

Let us analyze separately the procedures OVERLAY(), CONVERT2(), COMBINE2(), FILL2(), CONVERT3(), COMBINE3(), and FILL3() focusing on the effect of using linear tree representations.

With linear tree representations OVERLAY() clearly inspects once each node of both trees and its complexity is thus proportional to the total number of input nodes, which is also a bound on the number of output nodes. With explicit tree representations the complexity of OVERLAY() is at most proportional to the size of the smaller input tree.

CONVERT2() performs a fixed amount of computation for each node of the 2D image tree of an edge segment and directly outputs the DF-expression. The analysis of COMBINE2() and FILL2() for each face corresponds closely to that of COMBINE3() and FILL3() given below.

CONVERT3() also performs a fixed amount of computation for each node of the output tree, except for the case where an output leaf is WHITE and the corresponding portion of the 2D tree must be skipped. (In this case the brother of the leaf will not be WHITE.) Because of the choice of the projection plane, the amount of skipping can be at most proportional to the number of output nodes. With explicit tree representations the skipping could be performed more efficiently. However, its contribution to processing time is minor.

When there are N faces, COMBINE3() calls OVERLAY() N − 1 times. Each node resulting from CONVERT3() passes through OVERLAY() at most $\lceil \log(N) \rceil$ times (logarithms are to base 2). This follows from the remark above on the size of the output of OVERLAY(). Thus for a total of I input nodes in the image trees of the faces the complexity of all the OVERLAY's is at most $I\lceil \log(N) \rceil$. Of course, the elementary operations are very simple. COMBINE3() requires at most twice the amount of space needed for storing the image trees of the faces. Using techniques similar to external sorting, disk storage may be used for this purpose.

FILL3() performs a fixed amount of work for each node, except for the contribution of the union-find -algorithm. Tarjan (1975) has shown that this contribution is almost linear in the number of operations performed. Thus the complexity of FILL3() is very nearly linear in the number of nodes. The worst case complexity of FILL3() is better than that of the connected components algorithm of (Samet 1981). However, the boundary determination method of Jackins and Tanimoto (1983) could be applied to achieve equal performance with explicit tree structures.

The worst case space complexity of the connectivity labeling algorithm is about $3M^2$ face elements: It is easy to construct 3D checkerboard-like images, which would require the active borders to contain only face elements at voxel level. The union-find -algorithm requires a label array with size determined by the highest label used. In practice, central memory requirements are somewhat difficult to determine à priori. In the following section we report some experiences.

6. Experiences

We have programmed the conversion algorithm in C language and run it on a VAX 11/750 (without a floating point accelerator), under Unix to determine its practical efficiency. Even though processing costs depend heavily on implementation details we report below various cost components to give an indication of their relative magnitudes.

Detailed performance testing is based on the following solids:

(1) B(100) - ball approximated by 100 faces

(2) B(400) - ball approximated by 400 faces

(3) Ore - the ore body of Figure 1-1a (588 faces)

(4) Exc. - an excavation (Figure 6-1, 40 faces).

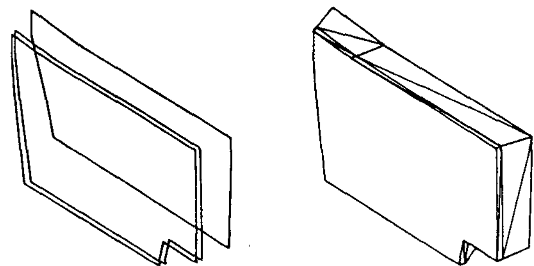


Figure 6-1
Test Solid, an Excavation

The effect of the theoretical non-linearity of the union-find -algorithm is so small that we can combine the experimental results into the following overall average costs per node.

(1) OVERLAY() requires about 17 microseconds per input node. Thus to COMBINE N trees containing a total of I nodes, the summed OVERLAY'ing time is at most $17I\lceil \log(N) \rceil$ microseconds.

(2) CONVERT2() requires about 170 microseconds per output node.

(3) CONVERT3() requires about 130 microseconds per output node. (The subroutine has been optimized further than CONVERT2().)

(4) FILL3() requires about 380 microseconds per input node and the resource requirements for FILL2() are a bit smaller.

The implementations, save that of OVERLAY(), use recursion so that subroutine calls account for much of the above costs.

To help appreciate the unit costs we note that the condensed binary image tree of a unit sphere at resolution $M = 2^7$ contains 208000 nodes. (A corresponding condensed completely BLACK ball only contains 117000 nodes!)

Tables 1 and 2 compare the run times (in VAX 11/750 CPU seconds) of the new method and the old one reported in (Tamminen et al. 1984). Alas, we have not found other publications to compare to.

	B(100)	B(400)	Ore(588)	Exc.(40)
New method	165	237	245	22
Old method	1350	1400	1600	324

Table 1. Processing Time at Resolution 128

	B(100)	B(400)	Ore(588)	Exc.(40)
New method	53	100	100	8
Old method	380	400	410	79

Table 2. Processing Time at Resolution 64

Tables 3 and 4 help in a detailed evaluation of the choices made in constructing the algorithm. Table 3 shows the contribution of each phase (in CPU seconds) to total processing time. Table 4 gives summed sizes (number of nodes) of the various kinds of image trees: Output is the final result, Boundary (3D) is the boundary of the final result, Faces denotes the trees of all faces taken separately, Proj. faces the 2D projections of faces, Boundary (2D) the trees of polygon boundaries, Segments the trees of polygon edges taken separately, and Overlay the number of nodes passing through the various invocations of OVERLAY().

	B(100)	B(400)	Ore(588)	Exc.(40)
FILL3()	79.6	84.0	62.2	7.7
CONVERT3()	32.3	46.3	47.4	4.9
OVERLAY()	29.7	45.1	40.7	2.8
FILL2()	16.9	32.1	41.6	3.2
CONVERT2()	9.2	23.0	29.2	2.6

Table 3. Processing Costs at Resolution 128

	B(100)	B(400)	Ore(588)	Exc.(40)
Output	116711	122485	98947	12331
Boundary (3D)	201668	207794	169108	20880
Faces	259758	346778	360062	34403
Proj. faces	29022	68692	92664	7825
Boundary (2D)	41370	90350	117034	10299
Segments	54088	142552	179604	15090
Overlay	1650000	2580000	2390000	180000

Table 4. Summed Sizes of Trees at Resolution 128

From Table 3 we see that the main part of the time is taken by determining the image tree of the boundary of the polyhedron. Our approach to this task was chosen for its uniformity (2D and 3D phases are almost identical) and robustness. However, there is much room for improvement by using different techniques.

The only part of our algorithm, whose efficiency is seriously affected by the use of linear tree representations, is OVERLAY(). Its contribution to the total run time is generally less than 20%. Also, a more efficient (in the expected case) FILL3() is conceivably possible with an explicit tree structure. This is because we need not form exact connected components but only extract the part connected to the outside. This can be performed using depth first search for leaves lying on the image border and recursive neighbor finding, starting from each unlabeled one of them.

The central memory requirements of FILL3() for B(400) at resolution 128 consist of about 3500 records for active faces. This compares favorably to the worst case of about 50000 records. Further, about 1500 tentative labels are formed. As resolution is increased by a factor of two the size of the output tree grows by a factor of four. The same holds for processing time of the connectivity labeling phase and for the number of putative labels. However, the number of active faces seems to grow only linearly with resolution.

The number of nodes in the two-dimensional image trees depends on the summed length of edges measured at the chosen resolution and to a lesser extent on the number of edges of the polyhedron. The length of edges grows linearly with resolution. Similarly, the summed size of the three-dimensional image trees depends on the surface area of the polyhedron and its number of faces. The surface area grows with the square of resolution. The processing time of the new algorithm is affected by both the above factors and thus grows somewhat more slowly than that of the old one, whose cost depends almost exclusively on the number of leaves output.

Our connected components labeling technique seems to outperform that of Lumia (1983), based on the voxel matrix representation, by orders of magnitude, in cases typical of the conversion problem (Samet and Tamminen 1984). This is mainly explained by the lesser amount of image elements in our representation.

The constituent parts of our algorithm can be connected in various ways. We recommend keeping the conversion of the boundary and the final connectivity labeling as separate programs communicating through a Unix pipe. With this structure the first phase can be easily replaced by another one, say, for processing curved surfaces.

7. Conclusions

We have presented an algorithm, efficient in practice, for converting a polyhedron into an octree-like block model. A characteristic of the algorithm is that all its phases operate directly on linear tree representations.

We believe that the method presented can be applied as a general conversion tool in boundary representation modelers. Up to the present conversion seems to have been possible in practice only for basic building block solids, which have then been combined on the octree side by using boolean set operations. The conversion program links our modeler (GWB) with that of (Yamaguchi et al. 1984). An interesting practical research problem is to find the optimal division of labor between boundary representations and octrees in similar combined systems.

Our experiences on applying the solid modeling techniques described here to mine modeling have been very encouraging but will be reported in more detail elsewhere.

ACKNOWLEDGEMENTS

The work has been supported by the Academy of Finland and by the National Science Foundation under grant MCS-83-02118. We thank Reijo Sulonen for his comments and Olli Karonen for help with the mine modeling data and figures.

REFERENCES

1. Gargantini, I., Linear octtrees for fast processing of three dimensional objects. CVGIP **20**(1982)4, pp. 363-374.

2. Jackins, C.L. and Tanimoto, S.L., Oct-trees and their use in representing three-dimensional objects. CGIP **14**(1980), pp. 249-270.

3. Jackins, C.L. and Tanimoto, S.L., Quad-trees, oct-trees and K-trees: a generalized approach to recursive decomposition of Euclidean space. IEEE **PAMI**-5(1983)5, pp. 533-539.

4. Journel, A.G. and Huijbregts, Ch, J., Mining Geostatistics. Academic Press, 1978.

5. Karonen, O., Tamminen, M., Kerola, P., Mitjonen, M., and Orivuori, E., A geometric mine modeling system. Proc. Autocarto Six Conference, Ottawa, 1983 pp. 374-383.

6. Kawaguchi, E. and Endo, T., On a method of binary picture representation and its application to data compression. IEEE PAMI 5(1980)1, pp. 27-35.

7. Klinger, A., Patterns and search statistics. In Optimizing Methods in Statistics, Rustagi, J.S. (Ed.), Academic Press, New York, 1971, pp. 303-337.

8. Lee, Y.T. and Requicha, A.A.G., Algorithms for computing the volume and other integral properties of solids. II. A family of algorithms based on representation conversion and cellular approximation. CACM **25**(1982)9, pp. 642-650.

9. Lumia, R., A new three-dimensional connected components algorithm. CVGIP **23**(1983), pp. 207-217.

10. Meagher, D., Geometric modeling using octree encoding. CGIP **19**(1982a), pp. 129-147.

11. Meagher, D., Octree generation, analysis and manipulation Report IPL-TR-027, Rensselaer Polytechnic Institute, Troy, New York, 1982b.

12. Meagher, D., Personal communication. 1983.

13. Mantyla, M. and Sulonen, R., GWB - A Solid Modeler With Euler Operators. IEEE Computer Graphics & Applications **2**(1982)7, pp. 17-31

14. Mantyla, M. and Tamminen, M., Localized set operations for solid modeling. Computer Graphics**17**(1983)3, pp. 279-288.

15. Requicha, A.A.G. and Voelcker, H.B., Solid modeling: current status and research directions. IEEE Computer Graphics and Applications 3(1983)7, pp. 25 - 37.

16. Requicha, A.A.G., Representations of rigid solids: theory, methods and systems. ACM Comp. Surv. **12**(1980), pp. 437-464.

17. Samet, H., Region representation: quadtrees from boundary codes. CACM **23**(1980)3, pp. 163-170.

18. Samet, H., Connected component labeling using quadtrees. JACM **28**(1981)3, pp. 487-501.

19. Samet, H., The quadtree and related hierarchical data structures. To appear in ACM Comp. Surv. Also TR-1329, Computer Science Department, University of Maryland, College Park, MD, 1983.

20. Samet, H. and Tamminen, M., Computing geometric properties of images represented by linear quadtrees. Report TR-1359, Computer Science Department, University of Maryland, College Park, MD, 1983.

21. Samet, H. and Tamminen, M., Efficient image component labeling. Report TR-1420, Computer Science Department, University of Maryland, College Park, MD, 1984.

22. Sedgewick, R., Algorithms. Addison-Wesley, Reading, 1983.

23. Srihari, S.N., Representation of three-dimensional digital images. ACM Comp. Surv. **13**(1981)4, pp. 399-423.

24. Tamminen, M., Encoding pixel trees. To be published in CVGIP, 1984.

25. Tamminen, M., Karonen, O., and Mantyla, M., Block model conversion using an efficient spatial index. To be published in CAD Journal, 1984.

26. Tarjan, R.E., On the efficiency of a good but not linear set union algorithm. JACM **22**(1975), pp. 215-225.

27. Yamaguchi, K., and Kunii, T.L., A layered string data structure for an octree model. Techn. Rep. 83-15, Dept. of Information Science, Univ. of Tokyo, 1983.

28. Yamaguchi, K., Kunii, T.L., Fujimura, K. and Toriya, H., Octree-related data structures and algorithms. IEEE Computer Graphics and Applications 4(1984)1, pp. 53-59.

29. Yau, M-M and Srihari, S.N., A hierarchical data structure for multidimensional images. CACM **26**(1983)7, pp. 504-515.