A Decomposable Algorithm for Contour Surface Display Generation

MICHAEL J. ZYDA
Naval Postgraduate School

We present a study of a highly decomposable algorithm useful for the parallel generation of a contour surface display. The core component of this algorithm is a two-dimensional contouring algorithm that operates on a single 2×2 subgrid of a larger grid. An intuitive procedure for the operations used to generate the contour lines for a subgrid is developed. A data structure, the contouring tree, is introduced as the basis of a new algorithm for generating the contour lines for the subgrid. The construction of the contouring tree is detailed. Space requirements for the contouring tree algorithm are described for particular implementations.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*visible line/surface algorithms*

General Terms: Algorithms

Additional Key Words and Phrases: Contouring, contouring tree, contour surface display generation

1. INTRODUCTION

Contour surface display generation from three-dimensional grid data is one of the most frequently and widely used graphics application algorithms [1–4, 6, 7, 9–13, 16]. The core component of contour surface display generation is the two-dimensional contouring of successive two-dimensional slices of the three-dimensional grid (see Figure 1). The best review of the historical development of two-dimensional contouring algorithms and their properties is found in [12]. The most striking thing about contouring literature is that the algorithms described are never complete, clean solutions to the two-dimensional contouring problem. The tendency in the literature is to rely on ad hoc treatment of special cases, with no attempt made to fit the special cases into a general algorithmic framework. An additional problem with the algorithms in the literature is that they are computationally slow. This slowness prevents the algorithms' use in real-time situations on hosts other than supercomputers. This paper attempts to

This work has been supported by the Naval Postgraduate School Foundation Research Program, the U.S. Army Combat Developments Experimentation Center, Fort Ord, California, and by the Naval Ocean Systems Center, San Diego, California.

Author's address: Code 52, Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943.

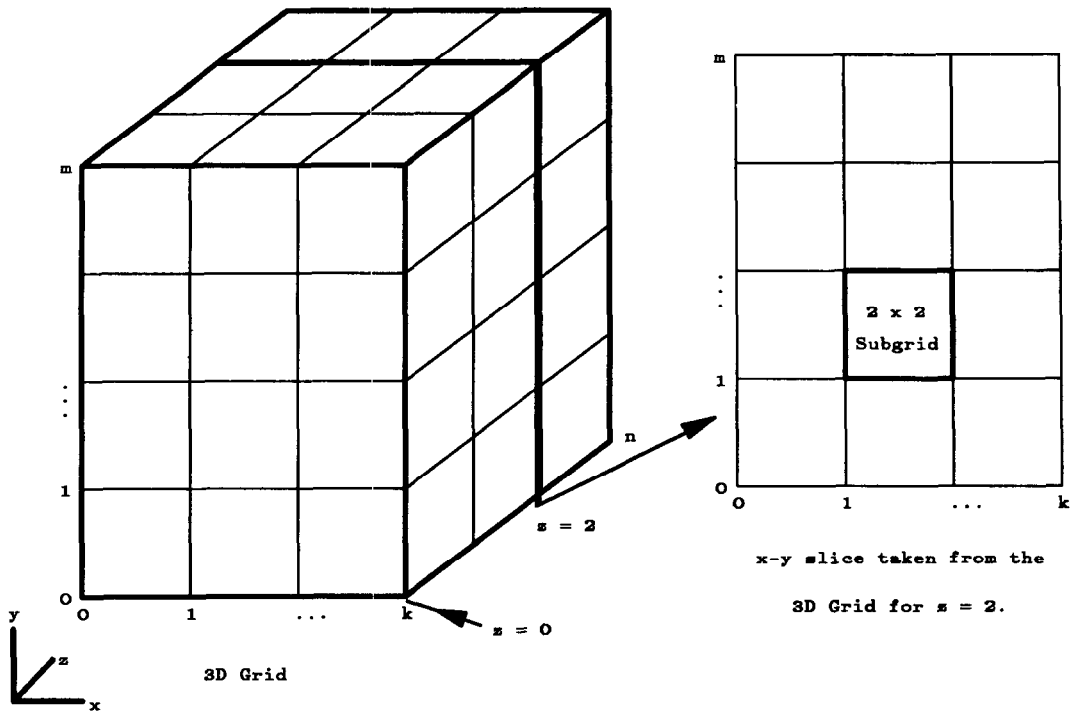


Fig. 1. Two-dimensional slice of a 3-D grid.

provide solutions for both the completeness and speed problems. The completeness problem is dealt with by presenting an algorithm for two-dimensional contouring based on a unifying data structure called the contouring tree. The speed problem is lessened by presenting that same data structure as a means for structuring subgrids of the two-dimensional grid into a constant form useful for rapidly computing successive contour levels. As an additional part of lessening the speed problem, the algorithm is shown to be eminently decomposable, that is, amenable to large-scale parallel computation.

2. DEFINITIONS

A *contour surface* is a visual display that represents all points in a particular region of three-space $\langle x, y, z \rangle$ that satisfy the relation $f(\langle x, y, z \rangle) = k$, where k is a constant known as the contour level. The function f represents a physical quantity that is defined over the three-dimensional volume of interest. The visual display created by this algorithm is the collection of lines belonging to the intersection of both the set of points satisfying the relation $f(\langle x, y, z \rangle) = k$, and a set of regularly spaced parallel planes passing through the region of three-space for which the relation is defined.

For this study, the function f is approximated by a discrete, three-dimensional grid created by sampling that function over the volume of interest. The three-dimensional grid contains a value at each of its defined points that corresponds

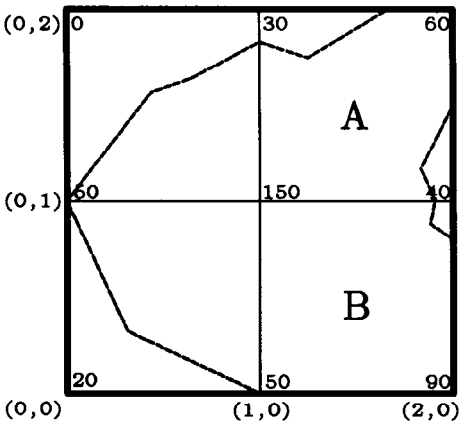


Fig. 2. Example contour grid with contours drawn for level 50.

to the physical quantity obtained from the function; that is, the value associated with point (x_0, y_0, z_0) is v_0, where $f(x_0, y_0, z_0) = v_0$. To minimize confusion, we specify the value at a particular grid point (x, y, z) by $a(x, y, z)$ and the value at a particular point (x, y, z) of the function by $f(x, y, z)$.

The visual display of the contour surface is created from this three-dimensional grid by taking two-dimensional slices of the grid, and constructing the two-dimensional planar contours for each slice at the designated contour level (see Figure 1). A slice of a three-dimensional grid is a planar, two-dimensional grid assigned a constant coordinate in three-space; that is, an x–y slice of $a(\langle x, y, z\rangle)$ corresponds notationally to $a(\langle x, y\rangle)$ for a particular z coordinate. The two-dimensional contours created are the lines that satisfy the relation $a(\langle x, y, z\rangle) = k$ for a particular coordinate, either x, y, or z, where again k is the constant contour level. If we contour all x–y slices of the three-dimensional grid at contour level k, we have a stack of parallel contours approximating the contour surface, each set of contours corresponding to a particular z coordinate. We can execute a similar contouring operation on all the x–z and y–z slices. The assemblage of the three sets of parallel contours, that is, the simultaneous display of all the contours created for the x–y, x–z, and y–z slices of the three-dimensional grid, produces a "chicken-wire-like" contour surface display.

3. FOCUS ON TWO-DIMENSIONAL CONTOURING

Given that the contour surface display generation algorithm works on the two-dimensional slice of the three-dimensional grid, it is best that we start our study with an understanding of the operations performed on that slice. Figure 2 shows a two-dimensional grid, with the contours drawn corresponding to contour level 50. Figure 3 shows that same two-dimensional grid, with the contours drawn corresponding to contour level 100. The goal of the two-dimensional contouring operation for such a grid is the determination of where lines are drawn on that grid given a fixed contour level k. To develop an intuitive feel for that determination mechanism, we restrict our focus to a small portion of the complete two-dimensional grid, the 2×2 subgrid. The 2×2 subgrid is defined to be that portion of the two-dimensional grid bounded by four adjacent grid points. In the

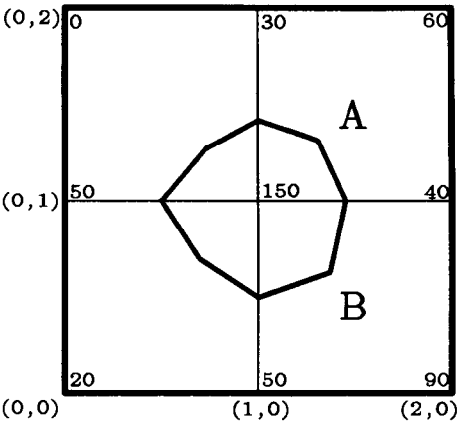


Fig. 3. Example contour grid with contours drawn for level 100.

two-dimensional grid of Figures 2 and 3, the lower left-hand 2 × 2 subgrid is bounded by points (0, 0), (1, 0), (1, 1), and (0, 1). The upper right-hand 2 × 2 subgrid of the same example is bounded by points (1, 1), (2, 1), (2, 2), and (1, 2).

4. AN INTUITIVE PROCEDURE FOR CONTOURING THE SUBGRID

The procedure used to generate the contours for a subgrid is the core part of two-dimensional contouring. If we compute the contours corresponding to contour level k for all subgrids of a two-dimensional grid, then we have determined the complete set of contours for that grid. To provide an intuitive feel for contour generation on the subgrid, we summarize that procedure to highlight potential problems.

The first step in the procedure is to determine whether any contours should be generated for the subgrid. That determination is based on whether any of the subgrid's edges contain the desired contour level k. An edge contains contour level k if the contour level is within the range of values defined by the grid points' edge.

The next part of the procedure is the computation of the contour edge intersections for any subgrid edges shown to contain the contour level. The point of intersection is computed through linear interpolation, using the grid values assigned to the endpoints of the edge and their corresponding coordinates. The point of intersection represents the location on the subgrid edge corresponding to the contour level k.

The determination of the connectivity necessary to form the appropriate contours from the list of edge intersections is the next part of the contour generation procedure. Before attempting to describe the procedure that assigns those connectivities, we first examine the subgrid's contour crossing possibilities. We accomplish that by looking at Figure 4, which shows the 12 possible ways for contours to cross or intersect a subgrid. (Note that rotations of cases 1–12 are considered as equivalent.)

Each of the cases of Figure 4 belongs to one of three subgrid crossing categories: (1) single edge crossings, (2) double edge crossings, and (3) constant edge borders

at the contour level. The 12 cases are drawn according to the following small set of rules for contour crossings:

(1) Contours are directed by the values associated with the edges and are directed toward their subgrid edge intersection points.
(2) For nonequivalued edges, if contours are indicated for a particular subgrid—that is, if there are edges in the subgrid that contain the contour level—there is only one point of intersection for each edge of the subgrid.
(3) Contours are continuous; that is, if a contour enters a subgrid, it must also leave that subgrid.
(4) Equivalued subgrid edges at the contour level are special cases and are drawn in their entirety. The only exception to this rule is that constant-valued subgrids are not drawn. This is by convention.

The first rule means that one determines the placement of contours, and hence, the connectivity of the edge intersections, by using both the values assigned to the endpoints of each edge of the subgrid and the computed intersections of the subgrid. The importance of this rule is twofold: First, it means that no outside forces or parameters direct contour placement. Second, it means that computed intersections are not the sole basis for determining the connectivity of the contours.

The second rule means that, for an edge intersected by a contour, there are no other points along that edge with contour value k. Since the range of values assigned to that edge is continuous and monotonic, the origin of this rule is clear. Note that this rule does not apply to equivalued edges.

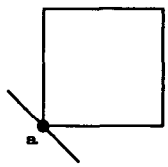
The third rule, that of contours being continuous, means that if a contour enters a subgrid it must also leave that subgrid; that is, the contour does not end inexplicably in the middle of the subgrid. A corollary of this rule, in combination with the first and second rules, states that contours entering a subgrid through an edge must leave via a different edge. Again, note that this corollary does not apply to cases with equivalued edges. This corollary holds though for contours tangent to grid points of the subgrid if we assign one edge to that grid point as the entering edge and assign the other edge as the leaving edge.

The last rule, that of equivalued subgrid edges at the contour level being special cases and being drawn in their entirety, is a rule based on visual expectations for the contour for such a subgrid edge. The only exception to this rule, that of not drawing constant-valued subgrids, is adopted by general convention.

Once we have an idea of the types of contour crossings possible for a subgrid, and once we have an outline of the rules used in composing those possibilities, we can then address the problem of forming a procedure for assigning connectivities or drawing commands to the computed edge intersections. Starting with the simplest cases of Figure 4, the equivalued edge cases, we clearly see that the connectivity generation procedure for subgrids containing such edges at the contour level is simple once those equivalued edges have been detected. If we find that we have a "constant subgrid," we do not need to issue any coordinates or drawing commands because by convention we have decided not to draw that case. Two of the other three possibilities, the "contour along one edge" and

Single Contour Crossings

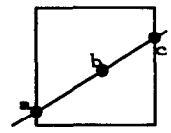
Case 1: Contour Tangent to the Subgrid



Expected Picture:

Drawpoint a

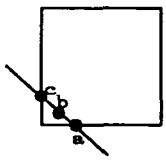
Case 3: One Contour Through Parallel Edges



Expected Picture:

Moveto a
Drawto b
Drawto c

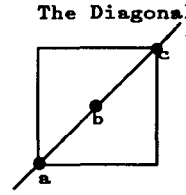
Case 2: One Contour Through Adjacent Edges



Expected Picture:

Moveto a
Drawto b
Drawto c

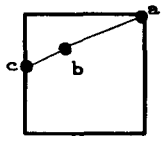
Case 4: Contour Across The Diagonal



Expected Picture:

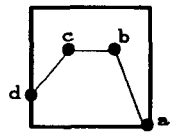
Moveto a
Drawto b
Drawto c

Case 5: Contour Through One Corner and Through One Edge



Expected Picture:

Moveto a
Drawto b
Drawto c



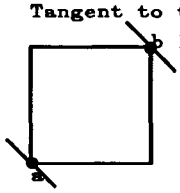
Expected Picture:

Moveto a
Drawto b
Drawto c
Drawto d

Figure 4a

Double Contour Crossings

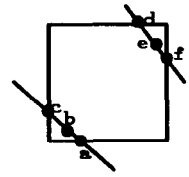
Case 6: Two Contours Tangent to the Subgrid



Expected Picture:

Drawpoint a
Drawpoint b

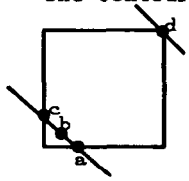
Case 8: Two Contours Through Adjacent Edges



Expected Picture:

Moveto a
Drawto b
Drawto c
Moveto d
Drawto e
Drawto f

Case 7: One Contour Tangent, One Contour through Adjacent Edges



Expected Picture:

Moveto a
Drawto b
Drawto c
Drawpoint d

Figure 4b

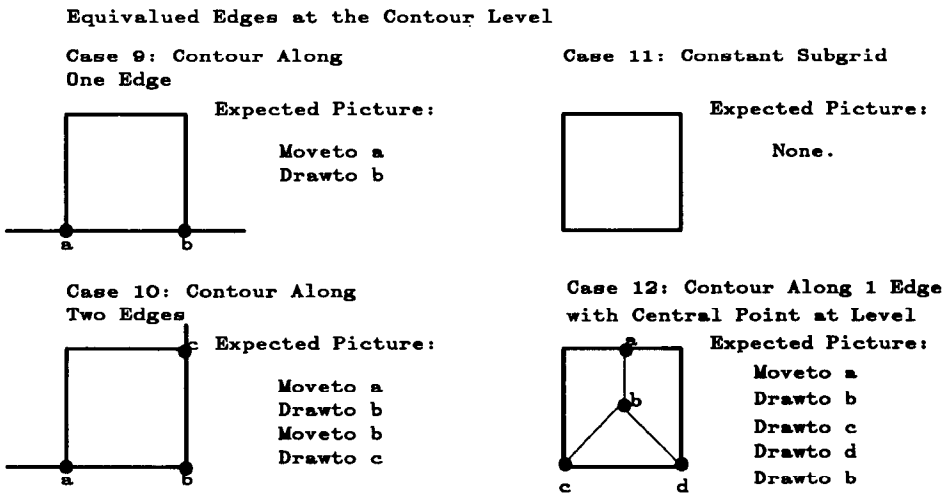


Figure 4c

Fig. 4. All possible contour crossings of a subgrid.

"contours along two edges" cases, are equally simple. The only operation necessary once such cases have been detected is to issue coordinates and drawing commands corresponding to the detected edges. The "contour along one edge with central point at level" case is a bit more difficult. In this case we note that the region *b–c–d* is an isovalued region. From our rules above, we know that we must have an additional edge leading to the isovalued region (contours do not just end inexplicably in the middle of a subgrid). We also know that we have left only one edge that can be intersected (the other three edges have intersections). Hence, we have the line *a–b* in case 12.

At first glance, given the edge intersections for a subgrid, the connectivity generation procedure for the single-contour cases of Figure 4 seems easy. It appears as if the only operation to be done is the issuing of coordinates and drawing commands corresponding to the straight line between the two points of edge intersection. Such a procedure works well if we know that we have a single contour crossing the subgrid. The only single-contour crossing case for which this does not work is the "contour tangent to the subgrid" case, which is an even simpler case for connectivity generation.

It is not until we consider the two contours crossing the subgrid cases of Figure 4 that we realize the potential for problems with the single-contour crossing procedure. A procedure based only on connecting edge intersections cannot differentiate between such cases as the "two contours tangent to the subgrid" and the "contour across the diagonal." There are other similar connectivity generation problems evident for the two-contours crossing cases. The "two contours through adjacent edges" case has four edge intersections. For that case we need to know which of these possible intersection pairs should be connected.

Now that we have established a background for the connectivity problem for contour crossings of the subgrid, we can describe the procedure used to contour the subgrid. This procedure uses as a data structure the contouring tree, which provides a coherent framework for subgrid contouring.

5. THE CONTOURING TREE

A *contouring tree* is a data structure that maintains the edges and associated grid values of a subgrid in a form that permits the rapid generation of the contour display for any contour level contained within the subgrid (see Figure 5). The formulation of the contouring tree is based on the observation that for any two-dimensional grid a continuous series of contour displays can be created for contour levels within the range of the minimum and maximum grid values [15]. By continuous, we mean that for any picture at "level" there is a picture at "level plus delta," where delta is small, as long as "level plus delta" is within the minimum and maximum grid values.

The use of the contouring tree is outlined best with an example of a small two-dimensional grid. Figures 2 and 3 depict the contours generated for contour levels 50 and 100. Figures 5 and 6 present the contouring trees created for two subgrids of the larger grid, subgrids A and B. The edges of the contouring trees correspond to the directed downhill edges inscribed on the subgrids of the figures. There are eight directed edges on each subgrid, four for the boundary edges and four for the edges to the subgrid's center point. The value used for the center point is the average of the four values comprising the corners of the subgrid. (For a reference on the usefulness of the center point average value in generating smooth contours, see [12].) The edges of the contouring trees are ordered, maintaining the same counterclockwise ordering as in the original subgrids. An "M" under a node means that a **moveto** display command should be generated for any coordinate that is created along an edge that has that drawing command on its lower valued node. A "D" means a **drawto** display command should be issued, and a "P" means a **drawpoint**.

Display generation from a contouring tree is performed by a preorder traversal of the contouring tree, producing a coordinate and drawing instruction whenever the desired contour level is found to be within the range of an edge of the contouring tree. A preorder traversal visits the root, the left subtree, the middle subtree, and then the right subtree. An edge's range is defined to be the set of values between those associated with the nodes on either end of the edge. More precisely, we say a contour level is within an edge if the following condition holds:

$$\text{lower\_node's\_value} \leq \text{contour\_level} < \text{higher\_node's\_value}.$$

For example, in Figure 5 at contour level 100, we issue coordinates and drawing instructions for the edges 1–2, 1–5, and 1–4. (Note that edges are specified by subgrid node numbers.) The drawing command issued for each of these edges is the one associated with the lower valued node of the edge. The coordinate for each of these edges is generated by a linear interpolation of the edge's endpoint coordinates according to the decrease in contour level along the edge.

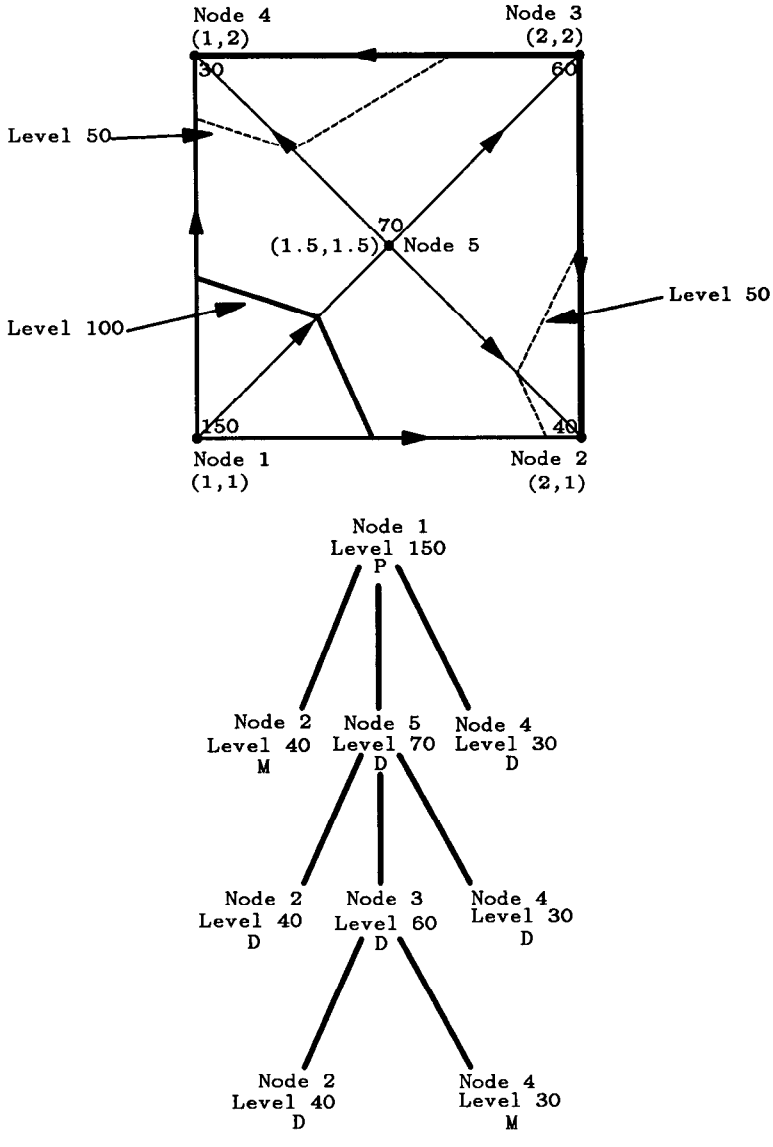


Fig. 5. Contouring tree for subgrid A.

5.1 Traversing the Contouring Tree

There are some subtleties not evident from the above that are best detailed using a pseudocode description of the traversal algorithm. Figure 7 shows the traversal procedure for the contouring tree assuming a particular data organization. The pointers to the descendant nodes of NODE are LEFT(NODE), MIDDLE(NODE), and RIGHT(NODE). For each node of the contouring tree, there are three pieces of information: the value associated with the node,

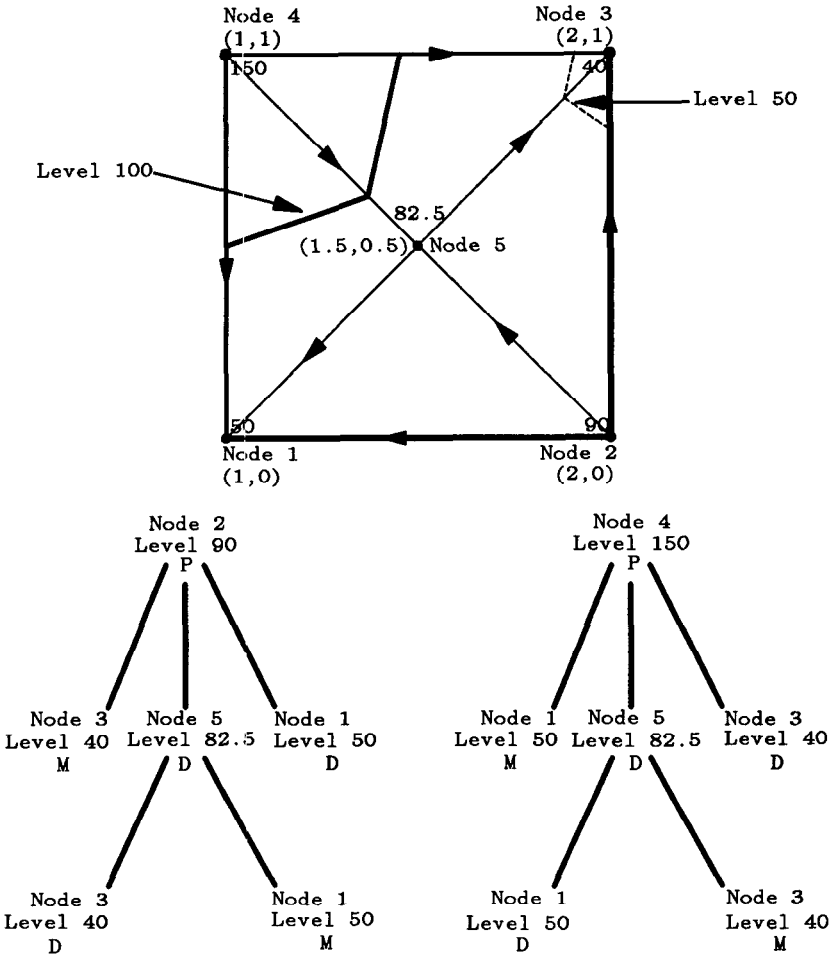


Fig. 6. Contouring trees for subgrid B.

VALUE(NODE); the coordinate associated with the node, *XYZ*(NODE); and the connectivity associated with the node, CONN(NODE).

The generation of coordinates and drawing commands from a contouring tree begins with routine CONTOUR_SUBGRID of Figure 7. That routine receives a pointer to the root node of the contouring tree. It then starts the traversal by calling routine VISIT with that root node. Routine VISIT checks to see whether the edge defined by the passed-in node and that node's ancestor, NODE and ANCESTOR, contains the contour level. If the edge does contain the contour level, the edge intersection coordinate is computed using linear interpolation and issued to the display along with the drawing command associated with that node, CONN(NODE). If we issue a coordinate and drawing command for a node, we need to check the subtree under that node for equivalued edges. If an equivalued edge at the contour level is found, a coordinate and drawing command are issued for that equivalued edge (routine VISIT_SUBTREE). Once a coordinate and drawing command have been issued for an edge, and once the subtree beneath

that edge has been investigated for equivalued edges, further traversal of the subtree is terminated. If an edge is found not to contain the contour level, the traversal continues as depicted at the bottom of routine VISIT.

The preorder traversal procedure described above generates the coordinates and drawing commands for the represented subgrid. To generate the coordinates for a larger two-dimensional grid, we generate the contouring trees for each subgrid of the grid and then apply the traversal procedure to those trees. We note here that no ordering is required in the generation of coordinates for the subgrids. The coordinate and drawing commands generated for each subgrid are complete and independent of the picture generated for any neighboring subgrid.

5.2 Contouring Tree Limitations

Having presented the use of the contouring tree, we must look back and discuss its capabilities and limitations. The contouring tree provides a uniform framework for generating the coordinates and drawing commands appropriate to the subgrid. The algorithm takes care of the single-contour crossing cases readily, as well as the two-contours crossing case for the subgrid. The algorithm correctly handles subgrids containing equivalued lines at the contour level and subgrids containing a single grid point at the contour level.

The core problems with this algorithm all concern issues of picture efficiency. Since the display generated for each subgrid is generated independently of any neighboring subgrids, equivalued lines at the contour level on the border of a subgrid are duplicated. A similar problem occurs for subgrid corner values that equal the contour level. If we display either of the above cases on a calligraphic display device, we see a bright line for the equivalued edge and a bright point for the grid value equal to the contour level. Another problem, also caused by the independent computation of each subgrid, is that no ordering is provided for coordinates that come out of this algorithm. For calligraphic displays this is a problem because for such devices electron beam movement is expensive. A contour display that causes the maximum movement of the electron beam every other subgrid greatly decreases the vector drawing capacity of the calligraphic display device.

There are three possible solutions to the problem of duplicate vectors. The easiest solution is to choose an output display device for which such picture inefficiencies do not matter, that is, a raster display. Vector ordering is also eliminated as a problem with this solution. The second solution is to set aside at the contour level points and lines that correspond to subgrid boundaries. A final pass at the end of the computation for a complete two-dimensional plane could readily cull the duplicates. This second solution does nothing for the vector ordering problem. If this solution to the vector duplicates problem is used, one either does not worry about the vector ordering, or one performs a sort on the vectors. The third solution, and the most expensive of the three, is to merge the set of trees generated for the two-dimensional grid such that duplicate edges in separate trees are eliminated. This solution has the added benefit that the resultant contours are generated in an order that solves the beam movement problem. This solution is not described in detail here, and the reader is referred to [14] for further detail. For this study, the first and simplest solution is assumed, and consequently, the expected output display device is the raster display.

Contouring Tree Structure

Pointers to Descendent Nodes:

LEFT(NODE)
MIDDLE(NODE)
RIGHT(NODE)

Values Associated with Each Node:

VALUE(NODE) : grid value.
XYZ(NODE) : coordinate of that grid value.
CONN(NODE) : drawing command (The M, D and P commands.).

Procedure CONTOUR_SUBGRID(ROOT)

```
VISIT(ROOT,ROOT)  # begin the traversal of the pointed at contouring tree.

end.
```

Procedure VISIT(NODE,ANCESTOR)

```
if(NODE == NULL)
{
        return
}

if((VALUE(NODE) <= CONTOUR_LEVEL < VALUE(ANCESTOR))
                                        OR
(VALUE(NODE) == CONTOUR_LEVEL AND NODE == ANCESTOR))
{

        # Edge contains the contour level.

        Issue a coordinate computed via linear interpolation along the edge.

        Issue CONN(NODE) as the drawing command.

        # Check subtrees of this node for equivalued edges.
        VISIT_SUBTREE(LEFT(NODE),NODE)
        VISIT_SUBTREE(MIDDLE(NODE),NODE)
        VISIT_SUBTREE(RIGHT(NODE),NODE)

        return   # no need to examine the subtree further.

}  # endif coordinates were generated for an edge.


VISIT(LEFT(NODE),NODE)          # visit left subtree.
VISIT(MIDDLE(NODE),NODE)        # visit middle subtree.
VISIT(RIGHT(NODE),NODE)         # visit right subtree.

return

end
```

Fig. 7. Pseudocode of the traversal algorithm for the contouring tree. (Continued on next page.)

```
Procedure VISIT_SUBTREE(SUBNODE,SUBANCESTOR)

    if(SUBNODE == NULL)
    {
        return
    }

    if(VALUE(SUBNODE) == CONTOUR_LEVEL)
    {

        Issue coordinates for the equivalued edge.
        Moveto XYZ(SUBANCESTOR).
        Drawto XYZ(SUBNODE).

    }


    VISIT_SUBTREE(LEFT(SUBNODE),SUBNODE)
    VISIT_SUBTREE(MIDDLE(SUBNODE),SUBNODE)
    VISIT_SUBTREE(RIGHT(SUBNODE),SUBNODE)

    return

end
```

Figure 7 continued.

6. CONTOURING TREE CONSTRUCTION

Contouring tree construction is best understood if we describe the procedure in graph-theoretic terms. We begin by assuming that we have a graph of five nodes, each node being one of the subgrid definition nodes or the center node of average value. The eight edges on that graph are the *subgrid boundary edges*, and the edges from each subgrid definition point to the center point of average value. We can readily assign directions to each edge of this graph using the values assigned to each node. Equivalued edges can be assigned an arbitrary direction. (One such assignment is to make equivalued edges along the border point in a counterclockwise fashion and equivalued edges from the center point in toward the center.) With these assumptions, each subgrid is perceived as a directed graph. The question then becomes, how do we obtain the contouring tree, or trees, from this directed graph? We can put this question in graph-theoretic terms if we notice that a contouring tree is a directed tree. The problem then becomes one of obtaining the directed tree, or trees, from the directed graph such that the ordering of edge attachment in the tree corresponds to the ordering in the directed graph. From graph theory, we have the requirement that a directed tree has the indegree of its root node equal to 0, and the indegree of every other node equal to 1 [5, p. 35]. To examine the indegree of each node of the directed graph, we must construct the indegree matrix D for that graph. The indegree matrix D of a directed graph G is defined in [5, p. 35] as

$$D(i, j) = \text{indegree}(i), \quad \text{if } i = j;$$

$$D(i, j) = -k, \quad \text{if } i \neq j, \text{ where } k \text{ is the number of edges in } G \text{ from } i \text{ to } j \text{ (i.e., } -1 \text{ for all our graphs).}$$

Figures 8 and 9 show the indegree matrices for our example subgrids.

D(i,j)	1	2	3	4	5
1	0	-1	0	-1	-1
2	0	3	0	0	0
3	0	-1	1	-1	0
4	0	0	0	3	0
5	0	-1	-1	-1	1

Fig. 8. Indegree matrix for the directed graph of subgrid A.

D(i,j)	1	2	3	4	5
1	3	0	0	0	0
2	-1	0	-1	0	-1
3	0	0	3	0	0
4	-1	0	-1	0	-1
5	-1	0	-1	0	2

D(i,j)	1	3	4	5
150				
1	2	0	0	0
3	0	2	0	0
4	-1	-1	0	-1
5	-1	-1	0	1

D(i,j)	1	2	3	5
90				
1	2	0	0	0
2	-1	0	-1	-1
3	0	0	2	0
5	-1	0	-1	1

Fig. 9. Indegree matrices for the directed graphs of subgrid B.

From the figures, we note that the roots of the contouring trees are recognizable from D as $D(i, i) = 0$. This matches the first part of the directed-tree requirement. Further examination of the diagonal of the indegree matrices introduces two difficulties in our attempts to convert the directed graphs into directed trees: (1) multiple roots ($D(i, i) = 0$ for more than one node) and (2) $D(i, i) > 1$ for some nodes. (Note: We have assumed that we have the structure represented by the directed graph and that we can manipulate it.)

The first problem, that of multiple roots, is handled by producing multiple sets of vertices and multiple indegree matrices such that there is only one root per indegree matrix. For our case, the maximum number of roots for a subgrid is two. We eliminate this problem by alternately removing each root node from the complete set of vertices and all edges attached to that deleted node, and then making separate indegree matrices. The second problem in the conversion of the directed graph into a directed tree, that of $D(i, i) > 1$ for some nodes, is resolved by node duplication. For each diagonal entry $D(i, i) = n$, where $n > 1$, we create $n - 1$ duplicates of the node, for a total of n, taking care to copy the appropriate values, coordinates, etc. We then reassign the original edges that went to the single node, such that each edge receives its own copy of the duplicated node. The edges that are reassigned are those between each node of column i of D that has a -1 and each of the n duplicate nodes. When performed for each indegree matrix created, this operation forms a new directed graph that is the desired directed tree.

6.1 Drawing-Command Placement

The preceding section details the creation of the contouring trees for the subgrid using nothing more than basic graph definitions and simple construction procedures. The only thing remaining for completing this construction is the placement of the drawing commands into the contouring trees.

Drawing commands are placed in the contouring tree to indicate when a line enters the region represented by the contouring tree either from a neighboring subgrid or from a location off the grid. If we look at the structure of the contouring tree, such as that exhibited in Figure 5, and consider that during the traversal the edges are examined in a counterclockwise, downward ordering from the root, we note that we need to place **moveto** (M) drawing commands on the lower valued node of each edge that presents a new lowest value for the tree. We insert these drawing commands by way of a preorder traversal of the directed tree, placing a **moveto** command on each node that is a new lowest value for the tree. This drawing-command placement strategy is based on the knowledge that, if we have a contour level for which we desire a picture, the first drawing command we generate for any contouring tree is a **moveto**. Although effective, this procedure does not provide a complete solution to drawing-command insertion. Some neighboring edges in the contouring tree, that is, edges sharing an ancestor node, have a "split" between them; that is, the edges are not immediate counterclockwise neighbors in the original grid. For this case we must mark the discontinuity in the contouring tree. We register the discontinuity on the lower valued node of the edge where the discontinuity occurs. For example, in Figure 5 the edges 3–2 and 3–4 are neighbors in the contouring tree but are not immediate neighbors in the original grid. We mark this split by placing an "M" on the lower valued node of edge 3–4.

To recognize the nodes that require a drawing command marking a split edge in the contouring tree, we must first examine where split edges occur in the subgrid. These occur wherever the subgrid has edge directions and tree edge configurations similar to those in Figure 10, that is, where there are neighboring tree edges not directly corresponding to neighboring subgrid edges. There are two cases in the figure.

For Case 1 the split edge is edge b–d. This edge neighbors edge b–c in the contouring tree but is not the immediate counterclockwise neighbor edge of b–c. The lower valued node of edge b–d, d, receives a **moveto** (M) drawing command. From the figure, we see that node d has a possibility of being a "sink," that is, a node with all incoming edges. This depends on the direction of edge a–d. There are two cases to consider: (1a) $a \rightarrow d$ or (1b) $d \rightarrow a$. Case (1a), $a \rightarrow d$, is easy to show as possible because it may be seen in Figure 5. Case (1b), $d \rightarrow a$, is somewhat more difficult, because we must show that it cannot possibly occur within the relations specified on Figure 10. We begin by compiling a small set of the given relations on the directed graph shown in the figure:

$$r \geq c,$$
$$r \geq a,$$
$$r \geq d,$$
$$b \geq d,$$
$$a \geq d.$$

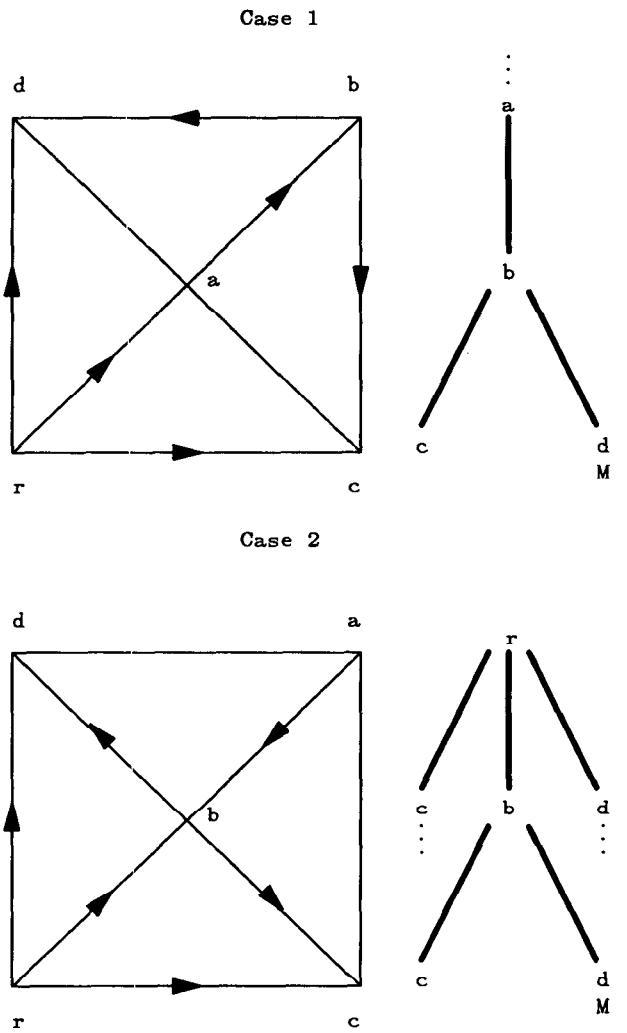


Fig. 10. All split-edge possibilities for the subgrid.

Case (1b) then becomes the question, is it possible to get $d \geq a$? Assume that $d \geq a$ is possible. Now we have $b \geq d$, so we can replace d with b. We get $b \geq a$, which is a contradiction of our original given, $a \geq b$, unless $a = b$. So we cannot have $d \rightarrow a$ unless $a = b$. Can we eliminate the possibility of the edge from a to b pointing from the center outward in the case in which $a = b$? We can do this when we set up the original directed graph, by biasing the edge selection of constant-valued edges so that they always point toward the center. Since the direction assigned to a constant edge is arbitrary, we can assume that we never see the condition $d \rightarrow a$, given the set of fixed directions assigned to Case 1. Hence, for Case 1 of Figure 10, a split-edge node is required during preorder traversal of the directed tree whenever we encounter, for the first time, a node representing a sink grid point. We now show a similar finding for Case 2.

In Case 2 of Figure 10, the split edge is edge b–d. This edge neighbors edge b–c in the contouring tree, but is not the immediate counterclockwise neighbor edge of b–c. The lower valued node of edge b–d, d, receives a **moveto** (M) drawing command. From the figure, we see that node d is possibly a sink. This depends on the direction of edge a–d. There are two cases to consider: (2a) $a \rightarrow d$ or (2b) $d \rightarrow a$. Case (2a), $a \rightarrow d$, occurs in Figure 6, so we know this case is possible. Case (2b), $d \rightarrow a$, is more difficult, because we must show that it cannot occur. We list a few relations describing the partial directed graph of the figure:

$$r \geq c,$$
$$r \geq b,$$
$$r \geq d,$$
$$b \geq c,$$
$$a \geq b,$$
$$b \geq d.$$

The question arises whether it is possible to get $d \geq a$. Assume $d \geq a$ is possible. Now we have $b \geq d$, so we can replace d with b. This gives $b \geq a$, which is a contradiction of our original given, $a \geq b$, unless $a = b$. So we cannot have $d \rightarrow a$ unless $a = b$. Clearly, we already have eliminated the possibility of the edge pointing from b to a by our initial configuration and by our biased edge selection for constant edges. Hence, for Case 2 of Figure 10, a split-edge node is required during preorder traversal of the directed tree whenever we encounter, for the first time, a node representing a sink grid point.

There are no other split-edge configurations possible in the subgrid. If we add a procedure that checks not only the new lowest value in the directed tree, but also the first encounter with a sink grid node during the same preorder traversal, we correctly place the drawing commands in the directed tree, converting it into the desired contouring tree.

7. IMPLEMENTATION NOTES[1]

Once we have an algorithm for contour generation from the subgrid, we then need to discuss its implementation. From our discussion, there are some obvious concerns: the time and memory costs incurred for building and using contouring trees. The answer to the contouring tree construction concern is that we can build all possible contouring trees ahead of time. From our prior discussion, we note that there are only 128 different subgrid possibilities or 128 possible contouring tree sets. (Note that there are really 256 possibilities for the eight edges of the subgrid. From our algorithm, we know that the center point of average value does not occur as the root of a contouring tree. Hence, we only have 128 subgrid possibilities or 128 contouring tree sets.) The only problem with building the contouring trees ahead of time is that it takes some memory space. The entire set of contouring trees can be constructed with 1104 nodes of the format shown in Figure 7. This requires about 10 kbytes of storage for the complete set of trees, given 2 bytes for the grid value, 3 bytes for the pointers to

[1] The interested reader is referred to a recent publication by Lorensen and Cline [8], which has a 3-D surface construction algorithm whose implementation has similarities to the one proposed in this paper.

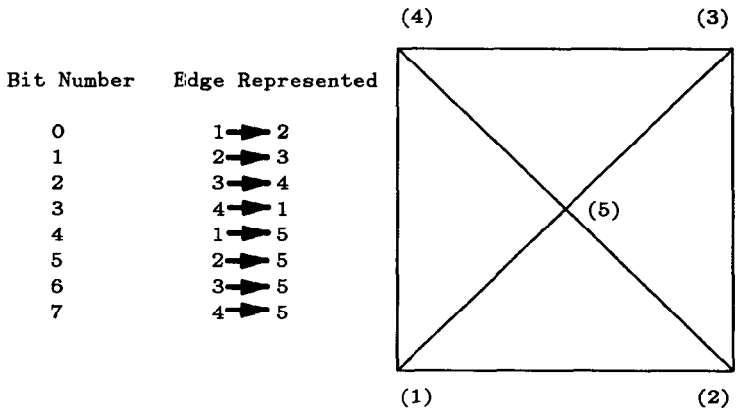


Fig. 11. Index number edge assignments for the subgrid. A "1" in the bit position means the edge exists. A "0" in the bit position means the edge of opposite direction exists.

the descendant nodes, 3 bytes for the grid coordinates, and 1 byte for the drawing command. On a graphics workstation with virtual memory, this is not a large concern.

Once we have the complete set of all possible contouring trees, we then need to consider how those trees are used, that is, how we go from a subgrid to a particular contouring tree set, and then how we generate coordinates from the contouring tree set. The first part, that of selecting a contouring tree set from an input subgrid, is simple. We can build an 8-bit index, corresponding to the adjacency matrix for the directed graph of the subgrid, by evaluating the edge directions (see Figure 11). This index can be used to select the corresponding contouring tree set from our precomputed contouring trees. The only computation involved in this operation is the computation of the edge directions, and the formation of the index.

The final step necessary, the generation of coordinates and drawing commands from the selected contouring tree set, is also simple. On each of the possible trees of the set (there may be more than one), we execute the preorder traversal algorithm of Figure 7. Figures for the number of memory references required for the complete algorithm are reported in [16]. In [16] the pretraversal part of the algorithm is shown to require 602 32-bit memory references to (1) determine whether the subgrid contains the current contour level (177 references), (2) compute the center point of average value (263 references), and (3) compute the index number (162 references). The maximum references for any subgrid's traversal is 2048 32-bit references. The maximum memory references then for the computation of the coordinates and drawing commands for any subgrid is 2650. In [16] we see that for a typical problem, a $30 \times 30 \times 30$ grid containing 75,690 subgrids, only 9900 of those subgrids generate any coordinates and drawing commands. If we multiply this out, we find that the $30 \times 30 \times 30$ grid computation requires some 38 million memory references. If we assume 50 nanoseconds per reference, this is about 1.9 seconds. At 250 nanoseconds per reference, this is 9.5 seconds. For the complete model used to derive these figures, see [16].

8. SPACE AND TIME COMPLEXITY

If we propose a new algorithm for contouring, we must somehow compare that algorithm to other algorithms in terms of space and time complexity. Unfortunately, this is hard without specific implementations of both sets of algorithms. There are, however, some generalities we can discuss.

With respect to the contouring tree algorithm and space–time complexity, there are two views, one for a software and the other for a hardware implementation. For a software implementation, we have a requirement for an additional 10 kbytes of space for the precomputed contouring trees. This space is not required in traditional contouring algorithms. In software, for each subgrid, we can precompute the contouring tree indices as already discussed and thereby add a byte of information to each subgrid. For a $30 \times 30 \times 30$ array, we have 75,690 subgrids, or a requirement for an additional 76 kbytes of information not required by a traditional contouring algorithm. Clearly, we buy time with this space, but in what way? The traversal of the contouring tree is at least as fast if not faster than traditional contouring algorithms. We can say this, because the three edge comparisons found in traditional algorithms are complete in the contouring tree algorithm once we have the tree's index. Computing the tree index requires eight edge comparisons. This means that using contouring trees is a good strategy for situations in which we compute the contours for a set of subgrids several times for different contour levels, computing and saving the tree indices only once per subgrid.

For a hardware implementation, we can do away with precomputing the contouring tree indices. The contouring tree index can be computed in the time it takes for one comparison if we have eight comparators working in parallel. We can have the precomputed contouring trees in ROM and can have multiple units performing tree traversals in parallel. The hardware solutions are many. Such solutions are more readily visible with the contouring tree algorithm than with traditional formulations. For a discussion of one particular VLSI implementation, the reader is referred to [17].

9. CONCLUSIONS

At the start of this paper, we expressed two motives for our work: (1) a desire for a complete algorithm for contour surface display generation, and (2) a desire for an algorithm that can increase the speed at which the contouring operation is performed. With respect to the first motive, we have shown that a subgrid contouring algorithm can be built around a unifying data structure—the contouring tree. The contouring tree moves the subgrid contouring operation beyond the ad hoc methods previously found in the literature. From our discussion, we know that the application of the subgrid contouring algorithm on each subgrid of all the two-dimensional slices (x–y, y–z, and x–z) of a three-dimensional grid generates the desired contour surface display.

Our second motive, a desire for an increase in speed for the contouring operation, is potentially solved by our algorithm. By showing that the computations on the subgrid are independent of those required for any other subgrid, we have shown that our surface display generation algorithm is highly decomposable. The importance of this decomposability is striking if we look at our typical $30 \times$

30 × 30 grid that contains 75,690 subgrids. For such a case, there is a potential for 75,690 concurrent operations. Given a technology that allows such parallelism, and if we can both load and unload the data from that system, it appears as if this algorithm has the potential for speeding up one of the most frequently performed graphics computations.

ACKNOWLEDGMENTS

The author wishes to acknowledge the hard work of the reviewers and the associate editor of *ACM Transactions on Graphics* in the production of this paper.

REFERENCES

1. BARRY, C. D., AND SUCHER, J. H. Interactive real-time contouring of density maps. In *Proceedings of the American Crystallographic Association Winter Meeting* (poster session) (Honolulu, Hawaii, Mar. 23–25). American Crystallographic Association, New York, 1979, pp. 233–234.
2. COTTAFAVA, G., AND LE MOLI, G. Automatic contour map. *Commun. ACM 12*, 7 (July 1969), 386–391.
3. DAYHOFF, M. O. A contour-map program for X-ray crystallography. *Commun. ACM 6*, 10 (Oct. 1963), 620–622.
4. DUTTON, G. An extensible approach to imagery of gridded data. In *Computer Graphics: Proceedings of SIGGRAPH 77* (San Jose, Calif., July 20–22). ACM, New York, 1977, p. 159.
5. EVEN, S. *Graph Algorithms*. Computer Science Press, Potomac, Md., 1979.
6. FABER, D. H., RUTTEN-KEULEMANS, E. W. M., AND ALTONA, C. Computer plotting of contour maps: An improved method. *Comput. Chem. 3* (1979), 51–55.
7. GOLD, G. M. Automated contour mapping using triangular element data structures and an interpolant over each irregular triangular domain. In *Computer Graphics: Proceedings of SIGGRAPH 77* (San Jose, Calif., July 20–22). ACM, New York, 1977, p. 170.
8. LORENSEN, W. E., AND CLINE, H. E. Marching cubes: A high-resolution 3D surface construction algorithm. In *Computer Graphics: Proceedings of SIGGRAPH 87* (Anaheim, Calif., July 27–31). ACM, New York, 1987, pp. 163–169.
9. MCLAIN, D. H. Drawing contours from arbitrary data points. *Comput. J. 17*, 4 (Dec. 1974), 318.
10. SUTCLIFFE, D. C. An algorithm for drawing the curve $f(x, y) = 0$. *Comput. J. 19*, 3 (Sept. 1976), 246.
11. SUTCLIFFE, D. C. A remark on a contouring algorithm. *Comput. J. 19*, 4 (Dec. 1976), 333.
12. SUTCLIFFE, D. C. Contouring over rectangular and skewed rectangular grids—An introduction. In *Mathematical Methods in Computer Graphics and Design*, K. W. Brodlie, Ed. Academic Press, New York, 1980, pp. 39–62.
13. WRIGHT, T., AND HUMBRECHT, J. ISOSRF—An algorithm for plotting iso-valued surfaces of a function of three variables. In *Computer Graphics: Proceedings of SIGGRAPH 79* (Chicago, Ill., Aug. 8–10). ACM, New York, 1979, pp. 182–189.
14. ZYDA, M. J. Multiprocessor considerations in the design of a real-time contour display generator. Tech. Memo. 42, Dept. of Computer Science, Washington Univ., St. Louis, Mo., Dec. 1981.
15. ZYDA, M. J. Algorithm directed architectures for real-time surface display generation. D.S. dissertation, Dept. of Computer Science, Washington Univ., St. Louis, Mo., Jan. 1984.
16. ZYDA, M. J. Real-time contour surface display generation. Tech. Memo. NPS52-84-013, Dept. of Computer Science, Naval Postgraduate School, Monterey, Calif., Sept. 1984.
17. ZYDA, M. J., AND WALKER, R. A. Design notes on a single board multiprocessor for real-time contour surface display generation. *Comput. Graph. 12*, 1 (Jan. 1988). In press.

Received September 1984; revised July and October 1987; accepted November 1987