Navigating through Triangle Meshes Implemented as Linear Quadtrees

MICHAEL LEE and HANAN SAMET
University of Maryland

Techniques are presented for navigating between adjacent triangles of greater or equal size in a hierarchical triangle mesh where the triangles are obtained by a recursive quadtree-like subdivision of the underlying space into four equilateral triangles. These techniques are useful in a number of applications, including finite element analysis, ray tracing, and the modeling of spherical data. The operations are implemented in a manner analogous to that used in a quadtree representation of data on the two-dimensional plane where the underlying space is tessellated into a square mesh. A new technique is described for labeling the triangles, which is useful in implementing the quadtree triangle mesh as a linear quadtree (i.e., a pointer-less quadtree); the navigation can then take place in this linear quadtree. When the neighbors are of equal size, the algorithms have a worst-case constant time complexity. The algorithms are very efficient, as they make use of just a few bit manipulation operations, and can be implemented in hardware using just a few machine language instructions. The use of these techniques when modeling spherical data by projecting it onto the faces of a regular solid whose faces are equilateral triangles, which are represented as quadtree triangle meshes, is discussed in detail. The methods are applicable to the icosahedron, octahedron, and tetrahedron. The difference lies in the way transitions are made between the faces of the polyhedron. However, regardless of the type of polyhedron, the computational complexity of the methods is the same.

Categories and Subject Descriptors: G.1.2 [**Numerical Analysis**]: Approximation—*Approximation of surfaces and contours*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite element methods*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Display algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Boundary representations*; *Curve, surface, solid, and object representations*; I.4.1 [**Image Processing**]: Digitization and Image Capture—*Hierarchical*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Data structures, finite element analysis, hierarchical methods, neighbor finding, ray tracing, spherical modeling, triangle meshes

This work was supported in part by the National Science Foundation under grant IRI-9712715, the Department of Energy under contract DEFG0295ER25237, and an AASERT fellowship DAAH04-93-G-0106.
Authors' address: Computer Science Department, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742; email: magus@umiacs.umd.edu; hjs@cs.umd.edu.

1. INTRODUCTION

The representation of spatial data is an important issue in the development of efficient algorithms for applications in computer graphics, virtual reality, visualization, image processing, and geographic information systems (GIS). There are several ways of characterizing spatial data. The most common is by the dimensionality of the underlying space from which the data is drawn. The second takes the dimensionality of the underlying space as a given, and distinguishes between different data types by the extent or amount of the underlying space that they span. This allows us to easily see the difference between points, lines, regions, surfaces, volumes, and even data of higher dimensionality. In particular, point data has zero extent, in contrast to the remaining data types which have nonzero extents—they usually span more than one point in the underlying space.

In many applications, a hierarchical representation of the data is useful as a way of recursively partitioning the underlying space from which the data is drawn into smaller regions, where the decomposition criteria are usually based on data homogeneity or data distribution. There are many ways of positioning the partition lines. The two principal methods are choosing from a limited predefined set of positions, or permitting the partition lines to lie anywhere. Methods based on the region quadtree [Hunter 1978 ; Klinger 1971] (see also Samet [1990a; 1990b]) are examples of the former, while those based on the BSP tree [Fuchs et al. 1980] are an example of the latter.

In this paper we focus on representations such as the region quadtree (referred to as a *quadtree* in the rest of this discussion), as they result in a recursive partition of the underlying space into n congruent parts. The most common quadtree representation recursively decomposes the two-dimensional plane into four squares. Thus the underlying space is said to be spanned by a square mesh where the squares are of different sizes. We are interested in the case where the two-dimensional plane is recursively decomposed into four congruent triangles, where we assume that the initial underlying space is an equilateral triangle. In this case, the underlying space is said to be spanned by a triangular mesh. We term the result a *triangle quadtree*.

Such meshes find uses in many applications such as finite element analysis (e.g., Bank et al. [1983]; Bern et al. [1990]; De Floriani et al. [1997]; Eck et al. [1995]; Eppstein [1992]; Hoppe [1997]; Kela et al. [1986]; Perucchio et al. [1989]; Saxena and Perucchio [1989]; da Silva and Duarte-Ramos [1990]; Yerry and Shephard [1983]), where, for example, the mesh serves as a discrete representation of a region over which a particular function must be evaluated. The analysis is used, for example, to improve the accuracy of solving a partial differential equation over a region by controlling the error (e.g., Bank et al. [1983]). Meshes are also used in ray tracing as a way of representing a scene. In this situation, the meshes usually consist of cubical three-dimensional elements, in which case we are dealing with an octree; but they can also be two-dimensional, e.g., triangles. Regardless of the application, many operations on the data require the ability to examine a neighbor of a triangular element of the mesh (i.e., a node or a block) as well as making a transition to it. Usually the neighbor is of equal size, but this need not always be the case. In this paper, we show how to efficiently make such transitions when using a triangular mesh resulting from the use of a quadtree-like decomposition.

Triangular meshes are not restricted to purely two-dimensional data. They are also useful in the modeling of data that lies on the surface of a sphere, as is the case, for example, in applications that involve modeling the earth (e.g., De Floriani et al. [1996]). Traditional ways of representing such data invariably resort to projections onto the plane (e.g., Tobler and Chen [1986]) using one of many possible projections (e.g., Snyder [1987]). Clearly, there is no perfect projection. Ideally, we would like the projection to facilitate a decomposition into units of equal area. The difficulty here is that units of equal area in the projection do not necessarily correspond to units of equal area on the surface of the sphere. For example, it would be ideal if the projection made use of the common concepts of latitudes and longitudes, as in the case of the Mercator projection (e.g., Snyder [1987]). Unfortunately, this leads to great distortion around the poles, thereby precluding the use of equally-spaced lines of latitude.

These problems have led to the use of an approximation of the sphere by projecting its surface onto the faces of an inscribed regular polyhedron (e.g., Dutton [1984; 1990]; Fekete [1990]; Fekete and Davis [1984]; Goodchild and Yang [1992]; Otoo and Zhu [1993]), which are subsequently recursively decomposed using conventional techniques such as region quadtrees for two-dimensional planar data. The result is that each face is a triangular mesh in the form of a triangle quadtree; the sphere is represented as a collection of n quadtrees, where n is the number of faces in the inscribed polyhedron. The decomposition criteria can vary from equal value, as is the case when attempting to distinguish between oceans and land masses, to ranges of elevations when modeling terrain-like data.

The requirement of regularity precludes the use of other space decomposition methods (e.g., bintrees [Knowlton 1980; Samet and Tamminen 1988; Tamminen 1984], which are regular decomposition variants of a k-d tree [Bentley 1975]). This requirement also limits the choice of the inscribed polyhedra to one of the five Platonic solids: tetrahedron, cube, octahedron, dodecahedron, and icosahedron. The need to be able to recursively decompose the faces eliminates the dodecahedron from consideration, as it has 12 pentagonal faces. Clearly, the greater the number of faces in the inscribed polyhedron, the better the approximation. Therefore the icosahedron, with 20 triangular faces, is the most attractive. Figure 1 shows the top-level triangular faces of an icosahedron corresponding to the surface of the earth where the continents are highlighted. A case has also been made for using the octahedron, with eight triangular faces, as a representation for spherical data such as the earth [Dutton 1984; Goodchild and Yang 1992] because it can be aligned so that the poles are at opposite vertices of the octahedron

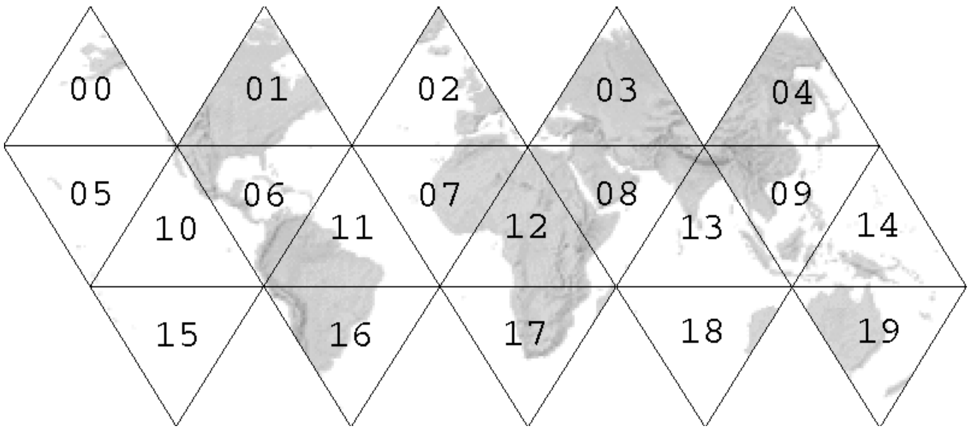

Fig. 1. Example showing the top-level triangle faces of an icosahedron corresponding to the surface of the earth; the continents are highlighted.

and the prime meridian and the equator intersect at another vertex. In addition, one subdivision line of each face is parallel to the equator. The tetrahedron with four triangular faces could also be used, but it is more difficult to find an intuitive rationale for its use.

The quadtree representation of the mesh is usually implemented as a tree with pointers from the root to its four children, which in turn contain pointers to their four children, etc. However, such an implementation can be rather wasteful of storage and has led to the development of a number of alternative quadtree representations that do not use pointers. The most common of these representations is known as the *linear quadtree* [Gargantini 1982]; here the quadtree is represented as a collection of numbers corresponding to its leaf nodes. In particular, for each quadtree corresponding to one of the faces of the inscribed polyhedron, leaf node i is represented by a unique pair of numbers known as its *location code* where the first number indicates the depth in the tree at which i is found and the second number indicates the path from the root of the tree to i. The path consists of the concatenation of the two-bit numbers corresponding to the child types of each node that is traversed on the path from the root to i. We refer to the path as the *path array component of the location code*. The quadtree is said to be *linear* because the pairs of numbers can be concatenated and the result stored in a sorted order that corresponds to a particular traversal of the underlying leaf nodes.[1]

One of the attractions of the linear quadtree when the faces are square is the ability to make use of binary arithmetic to navigate between any pair of adjacent nodes (i.e., nodes corresponding to squares of equal size) in time that is independent of the depth of the quadtree in which the nodes are found [Schrack 1992]). This enables the navigation to be performed very

[1]This traversal can be said to be postorder, preorder, or inorder, as nonleaf nodes are not visited.

efficiently, as it just requires a few bit manipulation operations that can be implemented in hardware using just a few machine language instructions. In this paper we show how to adapt the linear quadtree to triangular meshes so that such navigation can also be performed in time independent of the depth of the quadtree.

This technique has many potential applications. For example, it can be used in a ray tracer where a surface is represented by a quadtree triangle mesh instead of a quadtree square mesh [Fujimoto et al. 1986; Glassner 1984; Kaplan 1985; Samet 1989; Tamminen et al. 1984]. It can also be used in finite element analysis. For example, in many applications it is desirable to transform an arbitrary triangular mesh to a more restricted mesh with "subdivision connectivity" [Eck et al. 1995] by applying a "remeshing" process that yields a triangular hierarchy where groups of four triangles are aggregated into larger triangles (but see De Floriani et al. [1997] which uses a different approach to create a hierarchy based on the time and the location at which the mesh refinement takes place). The results of this "remeshing" process (i.e., Eck et al. [1995]) can be traversed efficiently using our techniques. In other applications (e.g., Bank et al. [1983]), the triangulation is not hierarchical, thereby causing some difficulty in performing operations such as finding ancestors, descendants, and neighbors. Our technique makes these operations much easier to perform. Others have devised special methods such as clipping the corners of the mesh elements (e.g., Kela et al. [1986]; Yerry and Shephard [1983]) to overcome the fact that the mesh elements are square (e.g., da Silva and Duarte-Ramos [1990], which has the drawback that the square mesh elements are not congruent, although they still form a hierarchy). Square mesh elements are viewed as attractive due to the ease of finding neighbors and being able to perform local refinement. With our methods, we can make use of the more natural triangular hierarchy, without the addition of special "corner" handling, while still being able to find neighbors and do local refinement efficiently.

In order to see the generality of our approach, in this paper we treat the more general navigation problem where the underlying surface is a sphere represented by a collection of quadtree triangle meshes. In particular, without loss of generality, we assume that the sphere is approximated by an icosahedron whose 20 faces are represented by quadtree triangle meshes. In our implementation, the adjacent triangles are not restricted to lie on the same face of the icosahedron, and the triangles whose neighbors are being sought can be at any depth in the tree. Thus the meshes are assumed to be continuous across the face of the sphere.

Although only our navigation solution for neighbors of equal size executes in time independent of the depth in the quadtree at which the nodes are found, we also show how to navigate between a node and its neighbor of greater or equal size with slightly greater execution time complexity. Our solution is in contrast to an existing method [Fekete 1990] which always has a worst-case execution time proportional to the maximum level of decomposition (i.e., the maximum depth of the quadtree, or resolution), regardless of whether the neighbors are of equal size. It is important to note that our approach is not restricted to the icosahedron and, as we shall show, can be used with the octahedron or a tetrahedron representation. In this case too, our methods differ from existing ones (e.g., Goodchild and Yang [1992]; Otoo and Zhu [1993]), which also have a worst-case execution time proportional to the maximum level of decomposition.

We achieve this by introducing a new method of labeling the elements of the triangular meshes corresponding to the faces of the icosahedron (which, we point out, is also applicable to the octahedron and tetrahedron) and showing how traditional two-dimensional neighbor-finding techniques [Samet 1982; 1990b] for quadtree square meshes (which work for both pointer-based and linear quadtrees) can be adapted to deal with quadtree triangle meshes. Neighbor-finding using these methods has a worst-case execution time proportional to the maximum level of decomposition, although the average has been shown to be constant [Samet 1982; 1990b] using techniques similar to amortized cost analysis. Next, we describe how the methods of Schrack [1992] for quadtree square meshes implemented as linear quadtrees can be adapted to quadtree triangle meshes represented as linear quadtrees, and we present our solution to handling adjacency between different faces of the icosahedron. This results in worst-case constant time algorithms for finding neighbors of equal size. We also show how to handle adjacencies between different faces of an octahedron and a tetrahedron. We conclude with a discussion of how to handle neighbors of greater size.

Our triangle labeling method is similar to that used for the octahedron [Goodchild and Yang 1992]. However, the difference is that the neighbor-finding methods used in Goodchild and Yang [1992] have a worst-case execution time proportional to the maximum level of decomposition, although they are based on the same principle as our methods (i.e., on Samet [1982; 1990b]) which have the same execution time complexity. In contrast, our triangle labeling method is very different from that used by Fekete [1990] for the icosahedron, as well as by Dutton [1990] and Otoo and Zhu [1993] for the octahedron.

The neighbor-finding methods of Dutton [1990], Fekete [1990], and Otoo and Zhu [1993] are quite different from ours. For example, Dutton [1990] and Fekete [1990] form new triangular regions for each level of decomposition by bisecting the edges of the parent triangle. This way of viewing the subdivision process results in a "floating" labeling scheme, as opposed to the fixed schemes used by us as well as by Goodchild and Yang [1992]. In other words, the labeling of children at each level varies based on the orientation of the parent triangle, which is effectively determined by the path from the root to the parent instead of being based on its global orientation. The advantage of this labeling technique [Fekete 1990] is that the path components of all location codes that correspond to the neighbors of a particular triangle differ by one directional code (at different depths of the hierarchy—that is, equivalently, at different positions in the code), at the expense of added complexity for the neighbor-finding process. Unfortunately, the fact that the transitions and node labels depend on orientation makes it difficult to adapt these methods to use binary arithmetic. Moreover, this technique [Fekete 1990] has the further disadvantage that finding neighbors that lie on different faces of the icosahedron is much harder. Otoo and Zhu [1993] use a square quadtree labeling scheme with an additional bit at the final decomposition level to distinguish between the orientations of the pair of triangles that make up each square. This makes it possible to use algorithms and techniques developed for quadtree square meshes, including a variant of the worst-case constant time neighbor-finding algorithm (not mentioned in Otoo and Zhu [1993]), but generally needlessly complicates neighbor-finding algorithms, especially when the neighbors are not of the same size. In contrast, our labeling method is considerably simpler and can be easily used to yield a neighbor-finding method that locates equal-sized neighbors in worst-case constant time.

2. TREE NODE LABELING

The icosahedron has 20 triangular faces each of which is decomposed recursively into four equilateral triangles. The result is a triangle quadtree. Every node in the tree represents a triangle. We use the terms *triangle* and *node* interchangeably. Each triangle has three edges, also termed *sides* or *boundaries*, and three vertices (also termed *corners*, e.g., Yerry and Shephard [1983]). These triangles always have one of two orientations: tip-up and tip-down. *Tip-up* means that the corresponding triangle *points* upward, and *tip-down* means that the triangle *points* downward. As tip-up triangles cover a different section of space than tip-down triangles (and cannot be made to cover the same space without some transformation such as rotation), we subdivide the two triangle types differently. We will see that using different subdivisions for the two types actually makes certain operations easier (e.g. point location). Since we plan on linearizing our tree, the discussions and algorithms all make use of a location code for each triangle consisting of two fields `LEV` and `CODE` corresponding to the depth and path arrays, respectively. As these location codes determine a triangle rather than just a point, the terms *location code* and *triangle code* are used interchangeably. We also often use the term *code* to refer to the path array. Moreover, since we decompose each triangle into four smaller equal-sized triangles, each child triangle adds two bits to the path array component of the location code of the parent. Regardless of the orientation of a triangle, we use the terms *vertical*, *left*, and *right* to refer to neighboring triangles of equal size along its horizontal, left angular, and right angular edges, respectively.

Tip-up triangles use the following bit patterns for children (see Figure 2(a)):

```
Top triangle: 00
Bottomleft triangle: 01
Center triangle: 10
Bottomright triangle: 11
```

Tip-down triangles use the following bit patterns for children (see Figure 2(b)):

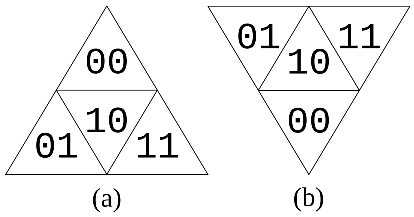

Fig. 2. Two possible orientations for a triangle: (a) tip-up, and (b) tip-down.

```
Topleft triangle: 01
Center triangle: 10
Topright triangle: 11
Bottom triangle: 00
```

Our node-labeling scheme is almost the same as that proposed by Goodchild and Yang [1992]. The difference is that they use the label 0 for the middle triangle, 2 for the left triangle, 3 for the right triangle, and 1 for the upper or lower triangles. As we see in Section 5, our labeling scheme permits us to make right and left transitions by use of addition and subtraction, which will enable us to perform the operations in constant time across the entire sphere. It is different from other methods (e.g., Dutton [1990]; Fekete [1990]), which are based on a "floating" labeling scheme (see Lee and Samet [1998] for an example).

There are several other advantages to using our node-labeling scheme. If we use the topmost or bottommost point to locate a triangle (since we only need one vertex, the orientation, and the size to determine the other two vertices), it is quite simple to traverse the tree using only local computations to determine where we are in space. The vertices of children are easy to determine relative to the positions of their parents. In particular, a child is always half the size (one quarter the area) of its parent. Child 10 always has the opposite orientation of its parent. The remaining three children always have the same orientation as the parent. See Figure 3 for an example of a tree that is encoded using this node-labeling method. This is in contrast to other methods (e.g., Dutton [1990]; Fekete [1990]; Otoo and Zhu [1993]), which lead to more complex neighbor-finding methods.

Regardless of whether a triangle is tip-up or tip-down, the triangles do not all have to be the same size. In other words, the triangles may be at different depths in the quadtree. As mentioned earlier, in the case of a linear quadtree, the depth is recorded in the LEV field. Assuming a maximum tree depth of n, the CODE field has $2n$ bits. For nodes or triangles at a depth i where $i < n$, the rightmost $n - i$ pairs of bits are 00 (i.e., the $2 \cdot (n - i)$ least significant bits are 0).

3. NEIGHBOR FINDING

In this section we describe how to find an equal-sized neighbor of a node p along an edge in the same face of the icosahedron. The algorithm is equivalent to the one described in Goodchild and Yang [1992], which is

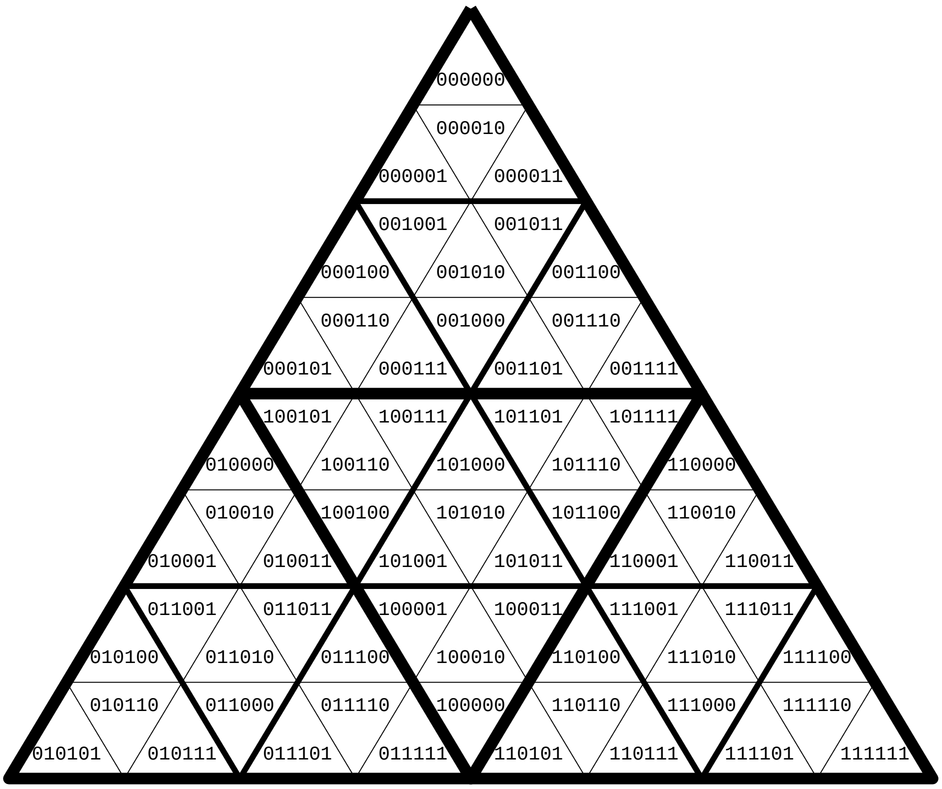

Fig. 3. Labeling of a tree that is three levels deep.

based on the approach of Samet [1982; 1990b]. We present it here because it is the basis of our extension to the entire sphere in Section 4, as well as our constant-time algorithm in Section 5. The node whose neighbor is being sought can be at any depth in the set of quadtrees corresponding to the faces; it is not restricted to being at the deepest level.

The algorithm does not need to make use of the actual coordinate values of the triangle block corresponding to p. Instead, it just processes the path array component of the location code (referenced by field name CODE, and often referred to simply as *the code* or the *bit pattern of the location code*). Elements of the path array are referenced using array notation. Assuming that the root triangle is at depth 0, given a triangle t with location code P, we say that CODE(P)[i] refers to the relative position (i.e., the child type) of the descendant at depth i of the root triangle that is also an ancestor of t. In this manner, we can effectively trace the path from the root of the tree to a given node by looking at CODE(P)[i] for successive values of i.

It is important to note that the procedures that we describe are destructive (i.e., in-place), in the sense that the location code P whose neighbor is being computed is overwritten with the location code of the neighboring triangle of equal size. If the original location code is to be preserved, it should be saved prior to being transmitted as a parameter to the neighbor-finding algorithm.

3.1 Step One: Locating the Nearest Common Ancestor

The first step is to find an ancestor of the current node which also contains the desired neighbor of that node. This node is called the *nearest common ancestor* of the two nodes. The technique used for finding the nearest common ancestor is effectively the same as that found in most standard quadtree implementations [Samet 1982; 1990b] that use trees. Of course, we aren't actually dealing with tree nodes. Instead, we want to find the location code of the nearest common ancestor within p's location code.

We now show how to find the right neighbor of p. If we start with p and work our way up (right to left in the path array corresponding to the location code), then we can stop scanning upward (leftward) when we find the ancestor of p that must contain the right neighbor of p. We stop when we encounter a node that has a right sibling (its parent contains a node that is adjacent to and to the right of p). If we look at Figure 2(a) , we see that this is true for children `01` and `10`. Also, in Figure 2(b), children `01` and `10` have right siblings. Thus, we can stop as soon as we find a `01` or `10` in the path array corresponding to the location code.

As an example, consider the location code with path array `010010110000`. Let us use `EXCODE` to refer to this path array. Therefore, `EXCODE[6]`=`00` (the last two bits). If we want to start searching for the right neighbor of the node corresponding to `EXCODE`, we need to examine the bits while looking for a `01` or `10`. `EXCODE[6]` does not equal `01` or `10`, so we continue upward. `EXCODE[5]` is the same as `EXCODE[6]`, so we continue upward. `EXCODE[4]`=`11` does not equal `01` or `10`, so we continue upward. `EXCODE[3]`=`10` means that we stop here. This sets us up for step two, described in Section 3.2. Note that the path array corresponding to the nearest common ancestor in this case is actually `0100` (all of `EXCODE` ending at `EXCODE[2]`).

A similar analysis is used to determine the nearest common ancestor when finding the left or vertical neighbor of a node. This process is encoded by procedure `STEP_ONE`. It makes use of the relation `STOPTAB` given in Table I (it is similar to the stop component in the conversion table used in Goodchild and Yang [1992]). `STOPTAB` is indexed by the bit pair corresponding to the child type and the direction of the neighbor. Entries corresponding to the end of the search for the nearest common ancestor are denoted by `TRUE` in the table.

Algorithm 1.

```
procedure STEP_ONE(CODE,NEIGHBOR_DIR,CHILD_TYPE,DEPTH);
/* Obtain the nearest common ancestor of the node at depth DEPTH
whose location code has path array CODE when seeking a neighbor
in direction NEIGHBOR_DIR. CHILD_TYPE indicates the child type
of the nearest common ancestor while the final value of DEPTH is
its depth. */
```

Table I. STOPTAB (Neighbor_Direction,Child_Type) relation indicating when to cease search for the nearest common ancestor in step 1 of the neighbor-finding algorithm

	Neighbor Direction		
Child Type (Bits)	Left	Right	Vert
00	FALSE	FALSE	TRUE
01	FALSE	TRUE	FALSE
10	TRUE	TRUE	TRUE
11	TRUE	FALSE	FALSE

```
begin
  value path_array CODE;
  value integer NEIGHBOR_DIR;
  reference integer CHILD_TYPE;
  reference integer DEPTH;
  preload Boolean array STOPTAB[0:2][0:3] with Table I;
  CHILD_TYPE  ← CODE[DEPTH];
  while not(STOPTAB[NEIGHBOR_DIR][CHILD_TYPE]) do
  begin
    DEPTH  ← DEPTH−1;
    CHILD_TYPE  ← CODE[DEPTH];
  end;
end;
```

3.2 Step Two: Updating the Path to Contain the Neighbor

Step two identifies and sets the position in the path array of the location code corresponding to the child of the nearest common ancestor (found in step one) to the appropriate child type of the neighbor. This step is simple. Say we are looking for a left neighbor q of node p. If we have the nearest common ancestor and we know what child contains p, it is easy to determine what child contains q. We move left. If child 10 contains p, child 01 must contain the neighbor node q. If we are looking for a right neighbor, we move right. The same procedure also holds for vertical neighbors.

As a concrete example, consider the location code with path array 010010110000. This is EXCODE from Section 3.1. The nearest common ancestor was 0100 and the child at EXCODE[3] was 10. Recall that we want the right neighbor, which means that the new child at this level should be on the right of 10. If we examine Figures 2(a) and 2(b) we find that 11 is to the right of 10 in both of them. Thus, in this step, we set EXCODE[3] to 11.

A similar analysis can be used to obtain the neighboring children for other child types and directions. This process is encoded by procedure STEP_TWO. It makes use of the relation NEXTTAB given in Table II (it is similar to the new address component in the conversion table used in Goodchild and Yang [1992]). NEXTTAB is indexed by the bit pair corresponding to the child type of the child of the nearest common ancestor and the direction of the neighbor that we are seeking. Its value is the child type of the neighboring child of the nearest common ancestor.

Table II. NEXTTAB (Neighbor_Direction,Child_Type) indicating child type of neighboring child of the nearest common ancestor

	Neighbor Direction		
Child Type (Bits)	Left	Right	Vert
00	11	01	10
01	00	10	01
10	01	11	00
11	10	00	11

Algorithm 2.

```
procedure STEP_TWO(CODE,NEIGHBOR_DIR,CHILD_TYPE,DEPTH);
/* Obtain the child type of the neighboring child of the nearest
common ancestor of the node and its neighbor in direction
NEIGHBOR_DIR. CODE is the path array corresponding to the
neighboring node in direction NEIGHBOR_DIR. Set the entry at
depth DEPTH of CODE to the child type of the ancestor of the
neighboring node. CHILD_TYPE indicates the child type of the
child of the nearest common ancestor that is an ancestor of the
current node whose neighbor is being sought. */
begin
    reference path_array CODE;
    value integer NEIGHBOR_DIR;
    value integer CHILD_TYPE;
    value integer DEPTH;
    preload integer array NEXTTAB[0:2][0:3] with Table II;
    CODE[DEPTH] ← NEXTTAB[NEIGHBOR_DIR][CHILD_TYPE];
end;
```

3.3 Step Three: Updating the Rest of the Path to the Neighbor

Step three finds the path from the child obtained in step two to the neighbor of p. This won't require searching, since we can exploit the fact that the path to a neighbor of a node is a reflection of the path to the node. In particular, for square quadtrees, we reflect the path to p to get the path to the neighbor q. For triangles, things work a little differently, but the layout of the children that we have chosen (see Section 2) keeps things simple. Reflection for the triangles works as follows. Keep in mind that a tip-up triangle is always adjacent to a tip-down triangle (and vice versa). This leads to three cases (one for each neighboring direction).

For left neighbors, 00 always becomes 11. Notice that 00 is always within the same y coordinate range as 01, 10, and 11 in the adjacent parent triangle. Since 11 is the closest of the three children, 11 is the appropriate "reflected" value. Child 01 always becomes 00. Only 00 in the adjacent parent triangle is within the same y coordinate range as 01, so 00 is the only candidate for the "reflected" value. Finding the left neighbors of children 10 and 11 is easy because their neighbors don't require leaving the parent node.

For right neighbors, 00 always becomes 01. Again, 00 is always within the same y coordinate range as 01, 10, and 11 in the adjacent parent triangle. Since 01 is the closest of the three children, 01 is the appropriate "reflected" value. Finding the right neighbors of children 01 and 10 is easy because their neighbors don't require leaving the parent node. Child 11 always becomes 00. Only 00 in the adjacent parent triangle is within the same y coordinate range as 11, so 00 is the only candidate for the "reflected" value.

For vertical neighbors, finding the neighbors of children 00 and 10 is easy because their neighbors don't require leaving the parent node. For both 01 and 11 the "reflected" value is equal to the original value (as Figures 2(a) and 2(b) are vertical reflections of each other).

For example, let's consider the location code 010010110000. This is EXCODE from Section 3.1. The nearest common ancestor (from step one) was 0100 and the child at EXCODE[3] was 10. In step two, we set EXCODE[3] to 11. The current (processed) portion of EXCODE is 010011. The entire code is 010011110000. Thus the remaining portion of EXCODE is 110000. This is the part that we will update in this step. We are still trying to find the right neighbor. EXCODE[4]=11 which becomes 00. EXCODE[5]=00 which becomes 01. EXCODE[6]=00 which becomes 01. The entire code 010010110000 becomes 010011000101 which gives us the right neighbor. The process is encoded by procedure STEP THREE.

Algorithm 3.

```
procedure STEP_THREE(CODE,NEIGHBOR_DIR,CHILD_TYPE,DEPTH,DEPTH_NCA);
/* Calculate the path array entries in CODE corresponding to the
NEIGHBOR_DIR neighbor of the original node at depth DEPTH.
CHILD_TYPE indicates the child type of the nearest common
ancestor while DEPTH_NCA is its depth. */
begin
  reference path_array CODE;
  value integer NEIGHBOR_DIR;
  value integer CHILD_TYPE;
  value integer DEPTH,DEPTH_NCA;
  preload integer array NEXTTAB[0:2][0:3] with Table II;
  while (DEPTH_NCA < DEPTH) do
  begin
    DEPTH_NCA ← DEPTH_NCA+1;
    CHILD_TYPE ← CODE[DEPTH_NCA];
    CODE[DEPTH_NCA] ← NEXTTAB[NEIGHBOR_DIR][CHILD_TYPE];
  end;
end;
```

3.4 Putting It All Together to Find a Neighbor

Now, if we combine the previously described steps, we can find the neighbor of any node in our tree. The only issue that remains is how to apply these techniques to the entire sphere. This is discussed in Section 4. Thus, the following routine is sufficient for finding any neighbor of equal size within one triangle quadtree.

Algorithm 4.

```
procedure FIND_NEIGHBOR(P,NEIGHBOR_DIR);
/* Return in P the location code corresponding to the neighbor
in the NEIGHBOR_DIR direction of the node corresponding to
location code P. */
begin
  value pointer location_code P;
  value integer NEIGHBOR_DIR;
  integer CHILD_TYPE;
  integer DEPTH;
  DEPTH ← LEV(P);
  STEP_ONE(CODE(P),NEIGHBOR_DIR,CHILD_TYPE,DEPTH);
  STEP_TWO(CODE(P),NEIGHBOR_DIR,CHILD_TYPE,DEPTH);
  STEP_THREE(CODE(P),NEIGHBOR_DIR,CHILD_TYPE,LEV(P),DEPTH);
end;
```

Since step one (finding the nearest common ancestor) involves examining each two-bit pair in the path array of the location code, its worst-case execution time is on the order of the length of the code (related to the height of the tree). Step two (changing two bits in the location code) always takes a constant amount of time. Step three (changing the remaining bits) requires examining the same bits as in step one, so its worst-case execution time is on the order of the length of the code. Overall, in the worst case, neighbor-finding requires time proportional to the length of the location code, which, of course, is the maximum level of decomposition.

4. EXTENSIONS TO THE ENTIRE SPHERE

Indexing the entire icosahedron (rather than just one of its faces) actually requires 20 of the previously described triangle quadtrees. This means that whenever we reach the top level (or root) of one of these trees, a bit of extra work is required. We label the 20 nodes corresponding to the roots of the quadtrees of the faces of the icosahedron using a 6-bit code ranging from `000000` (decimal 0) to `010011` (decimal 19). We could have fit the 20 values into just 5 bits, but we decided to use an even number of bits because the machine word length is always an even number of bits. The order in which the triangle faces of the icosahedron are numbered isn't important, since tables will be used most of the time. Thus we have numbered the faces using a simple left-to-right and top-to-bottom order (see Figure 1). Our numbering scheme has the property that triangles 0 to 4 are tip-up, 5 to 9 are tip-down, 10 to 14 are tip-up, and 15 to 19 are tip-down.

Neighbor finding in the entire icosahedron involves several modifications to our algorithm for a single face, but these changes are minor and have little impact on the computational complexity of the algorithms. We continue to work with the location code only. No coordinate values are used.

The only necessary modification to step one is that if we reach the top level of the spherical quadtree (or if there are no remaining bits to examine in the location code because we are at `CODE[0]`), we stop looking for the nearest common ancestor. Obviously, the entire sphere contains every

Table III. Execution Trace of Procedure EXT_STEP_ONE for the Left Neighbor of 000010010001010001

DEPTH	Child_Type	STOPTAB	CONDITION VALUE
6	01	FALSE	TRUE
5	00	FALSE	TRUE
4	01	FALSE	TRUE
3	01	FALSE	TRUE
2	00	FALSE	TRUE
1	01	FALSE	TRUE
0	000010		ALWAYS STOP AT 0

possible location, and is therefore an ancestor of every node. No additional stop tables are required. We always stop at the top level. This process is encoded by procedure EXT_STEP_ONE, not given here (see Lee and Samet [1998]), as the only change is the addition of the check for DEPTH $>$ 0 to yield a loop of the form:

```
while DEPTH > 0 and not(STOPTAB[NEIGHBOR_DIR][CHILD_TYPE]) do
```

Also, note that since Figure 1 is really a sphere, every triangle has a neighbor in every direction (the triangles on the ends wrap around), so we are well-prepared for step two.

As an example, consider the location code with path array 000010010001010001. We refer to it by EXCODE2. Our path array uses the extended format for the sphere so EXCODE2[0]=000010 (the first six bits) and EXCODE2[6]=01 (the last two bits). Let's suppose we are looking for the left neighbor of EXCODE2. Table III traces the execution of procedure EXT_STEP_ONE for this neighbor. Notice that in this case the nearest common ancestor is the entire sphere.

Step two is similar to the one described in Section 3.2 and is encoded by procedure EXT_STEP_TWO. The only modification from procedure STEP_TWO is the use of a different relation NEXTTOP (Table IV) to indicate how to update CODE[0]. It summarizes the actions for all possible neighbors from Figure 1 and replaces relation NEXTTAB in the algorithm for this case. This relation is used only when the nearest common ancestor from step one is the entire sphere. As an example, for the left neighbor of EXCODE2, from Table IV we find that the node to the left of EXCODE2[0] (000010 in binary or 2 in decimal) is 1. Thus, in step two, we set EXCODE2[0] to 000001.

Algorithm 5.

```
procedure EXT_STEP_TWO(CODE,NEIGHBOR_DIR,CHILD_TYPE,DEPTH);
/* Obtain the child type of the neighboring child of the nearest
common ancestor of the node and its neighbor in direction
NEIGHBOR_DIR. CODE is the path array corresponding to the
neighboring node in direction NEIGHBOR_DIR. Set the entry at
depth DEPTH of CODE to the child type of the ancestor of the
neighboring node. CHILD_TYPE indicates the child type of the
child of the nearest common ancestor which is an ancestor of the
current node whose neighbor is being sought. */
```

Table IV. NEXTTOP (Neighbor_Direction,Child_Type) indicating neighbors for triangles corresponding to faces of the icosahedron

	Neighbor Direction		
Child Type	Left	Right	Vert
0	4	1	5
1	0	2	6
2	1	3	7
3	2	4	8
4	3	0	9
5	14	10	0
6	10	11	1
7	11	12	2
8	12	13	3
9	13	14	4
10	5	6	15
11	6	7	16
12	7	8	17
13	8	9	18
14	9	5	19
15	19	16	10
16	15	17	11
17	16	18	12
18	17	19	13
19	18	15	14

```
begin
  reference path_array CODE;
  value integer NEIGHBOR_DIR;
  value integer CHILD_TYPE;
  value integer DEPTH;
  preload integer array NEXTTAB[0:2][0:3] with Table II;
  preload integer array NEXTTOP[0:2][0:19] with Table IV;
  if DEPTH>0 then CODE[DEPTH] ← NEXTTAB[NEIGHBOR_DIR][CHILD_TYPE]
  else CODE[0] ← NEXTTOP[NEIGHBOR_DIR][CHILD_TYPE];
end;
```

Step three requires one more relation called REFLTOP, given in Table V, to deal with the special case of reflection needed for nodes 0 to 4 and nodes 15 to 19. All other nodes still use the NEXTTAB relation from Table II in Section 3.3. The rationale for this additional relation is as follows. If we consider the left neighbor case and use a standard "mirror reflection," we see that 00 stays 00 and 01 reflects to 11. 10 and 11 cannot occur along the left edge of a node. Similarly, if we consider the right neighbor case, we see that 00 stays 00 and 11 reflects to 01. 01 and 10 cannot occur along the right edge of a node. The vertical case doesn't need to be updated. The algorithm in Section 3.3 works for the entire sphere if we use the reflection relation REFLTOP instead of NEXTTAB. Note that the vertical neighbor entries are identical to those in relation NEXTTAB given in Table II, since no special treatment is required for the vertical case.

As an example, consider again the location code with path array 000010010001010001, which was previously labeled as EXCODE2. The

Table V. REFLTOP(Neighbor_Direction,Child_Type) indicating the child type when finding neighbors across the top five and bottom five triangle faces of the icosahedron taking reflection into account

	Neighbor Direction		
Child Type (Bits)	Left	Right	Vert
00	00	00	10
01	11	–	01
10	–	–	00
11	–	01	11

current (processed) portion of EXCODE2 is 000001. The entire path array (after the previously mentioned example steps) is 000001010001010001. Thus the remaining portion of EXCODE2 is 010001010001. This is the part that we update in step three. Once again, we want to find the left neighbor. Using Table V, EXCODE2[1] (01) becomes 11, EXCODE2[2] (00) stays 00, EXCODE2[3] (01) becomes 11, EXCODE2[4] (01) becomes 11, EXCODE2[5] (00) stays 00, and EXCODE2[6] (01) becomes 11. Thus, 010001010001 becomes 110011110011. The final path array is 000001110011110011, which is the left neighbor that we desired.

Algorithm 6.

```
procedure EXT_STEP_THREE(CODE,NEIGHBOR_DIR,CHILD_TYPE,DEPTH,DEPTH_NCA);
/* Calculate the path array entries in CODE corresponding to the
NEIGHBOR_DIR neighbor of the original node at depth DEPTH.
CHILD_TYPE indicates the child type of the nearest common
ancestor while DEPTH_NCA is its depth. */
begin
  reference path_array CODE;
  value integer NEIGHBOR_DIR;
  value integer CHILD_TYPE;
  value integer DEPTH,DEPTH_NCA;
  preload integer array NEXTTAB[0:2][0:3] with Table II;
  preload integer array REFLTOP[0:2][0:3] with Table V;
  if DEPTH_NCA>0 or (4 < CHILD_TYPE and CHILD_TYPE < 15) then
  begin
    while (DEPTH_NCA<DEPTH) do
    begin
      DEPTH_NCA ← DEPTH_NCA+1;
      CHILD_TYPE ← CODE[DEPTH_NCA];
      CODE[DEPTH_NCA] ← NEXTTAB[NEIGHBOR_DIR][CHILD_TYPE];
    end;
  end
  else
  begin
    while (DEPTH_NCA<DEPTH) do
    begin
      DEPTH_NCA ← DEPTH_NCA+1;
      CHILD_TYPE ← CODE[DEPTH_NCA];
      CODE[DEPTH_NCA] ← REFLTOP[NEIGHBOR_DIR][CHILD_TYPE];
    end;
  end;
end;
```

The procedure for finding the neighbor that combines the three steps (i.e., FIND_NEIGHBOR given in Section 3.4) does not need to be modified, except for changing the names of the three procedures that it invokes by prepending "EXT_" to them. The result is encoded by procedure EXT_FIND_NEIGHBOR, not given here (see Lee and Samet [1998]).

5. CONSTANT-TIME NEIGHBOR-FINDING ALGORITHM

In this section we describe how neighbor finding can be accomplished in worst-case constant time. The algorithms presented here make use of the carry (borrow) property of addition (subtraction) to quickly find a neighbor without specifically searching for a nearest common ancestor and reflecting the path to the neighbor. We replace the iteration in steps one and three of the algorithm presented in Sections 3 and 4 by an arithmetic operation that takes constant time instead of as much as the depth of the tree as in the worst case of the iterative process. The resulting algorithms make use of just a few bit manipulation operations that can be implemented in hardware using just a few machine language instructions. Of course, the constant time bound arises because the entire path array for each location code can fit in one computer word. If more than one word is needed, then the algorithms are a bit slower but still take constant time. Our algorithms are based on the method devised by Schrack [1992] for square quadtrees implemented using pointer-less quadtrees represented by the location codes of the leaf nodes. Our contribution is twofold:

(1) Its adaptation to triangle quadtrees and the formulation of the appropriate triangle quadtree node-labeling technique.

(2) Its adaptation to the icosahedron in the sense that we make it work for neighboring triangles that are in different base triangles of the icosahedron.

Our algorithms also work for the octahedron and the tetrahedron. The only modification that is needed is to include a mechanism to handle the case where the neighboring triangles are in different base triangles of the solid (i.e., tetrahedron or octahedron). This is discussed in Section 6.

5.1 Square Quadtrees

In order to gain a better understanding of the basic idea, let us see how simple addition can be used with square quadtrees to find right neighbors of equal size. We make use of the following two definitions in our algorithms:

(1) ODDBITMASK is defined as an alternating bit pattern starting with a 1 at the leftmost bit position, so ODDBITMASK=10101010 . . . .

(2) EVENBITMASK is defined as an alternating bit pattern starting with a 0 at the leftmost bit position, so EVENBITMASK=01010101 . . . .

Both masks should be a full code length. For example, if we store the path array part of the location code in a long integer (4 bytes), then both masks would contain 32 bits. Our algorithms also make use of the following six bitwise operators:

(1) `COMPLEMENT(param1)` returns the complement of `param1`.

(2) `AND(param1,param2)` returns the result of a bitwise "and" between `param1` and `param2`.

(3) `OR(param1,param2)` returns the result of a bitwise "or" between `param1` and `param2`.

(4) `XOR(param1,param2)` returns the result of a bitwise "exclusive or" between `param1` and `param2`.

(5) `SHIFT_LEFT(param1)` returns the result of shifting `param1` to the left by one bit. A bit value of `0` is shifted into the bit string at the extreme right.

(6) `SHIFT_RIGHT(param1)` returns the result of shifting `param1` to the right by one bit. A bit value of `0` is shifted into the bit string at the extreme left.

Neighbor finding in square quadtrees is achieved in worst-case constant time by using the equivalence between the path array of the location code and the result of interleaving the bits that comprise the binary representation of the x and y coordinates of one of the corners (e.g., the upper-leftmost corner), chosen in a consistent manner, of the blocks corresponding to the leaf nodes. The result of bit interleaving is also known as a *Morton code* [Morton 1966; Samet 1990b]. For example, the Morton code for coordinates x and y has the form $y_{n-1}x_{n-1}\cdots y_1x_1y_0x_0$, where the y coordinate is the most significant. The right neighbor of equal size is obtained by incrementing the x coordinate value of the corner of the block by one. Assuming that we work with the Morton code of the block, instead of the individual coordinate values, we start this process by incrementing x_0 by one. If there is a carry, we add one to x_1. If there is another carry, we add one to x_2, and so on. This process is iterative, in the sense that the carries are propagated one bit at a time. Ideally, we want to accomplish the propagation of the carry using one operation. The problem is that when the addition operation is applied directly to the Morton code value, we need to skip the values of the corresponding y coordinates.

Schrack [1992] achieves the propagation of the carries in constant time by saving the values of all of the y bits, replacing their corresponding bit positions with `1`s, performing the addition, and then restoring the y bits to their original values. This technique is shown in the procedure `SCHRACK_RIGHT` given below.

Algorithm 7.

```
procedure SCHRACK_RIGHT(P);
/* Determine the location code of the right neighbor of equal
size of the square quadtree node with location code P. This
involves setting the CODE field of P. */
begin
  value pointer location_code P;
  path_array SAVED_BITS;
  /* Save all the $y$ bits */
  SAVED_BITS $\leftarrow$ AND(CODE(P),ODDBITMASK);
  /* Load the $y$ bit positions with 1s */
  CODE(P) $\leftarrow$ OR(CODE(P),ODDBITMASK);
  /* Add one (move right) */
  CODE(P) $\leftarrow$ CODE(P)+1;
  /* Clear the $y$ bit positions */
  CODE(P) $\leftarrow$ AND(CODE(P),EVENBITMASK);
  /* Restore the original $y$ bits */
  CODE(P) $\leftarrow$ OR(CODE(P),SAVED_BITS);
end;
```

In order to see how this algorithm works, consider the following example where $x = 11$ and $y = 6$. The Morton code is 01101101. The values of the odd bits are saved in SAVED_BITS, which for this example is 00101000. The result of the first OR with ODDBITMASK changes our Morton code to 11101111. Adding one yields 11110000. The second AND with EVENBITMASK changes our Morton code to 01010000. The last OR with SAVED_BITS restores our original y value, thereby making our final Morton code 01111000. We can easily see that this corresponds to a block with $x = 12$ and $y = 6$, which means that our algorithm did indeed obtain the proper answer.

Procedures SCHRACK_LEFT, SCHRACK_UP, and SCHRACK_DOWN, not given here (see Lee and Samet [1998]), use a similar technique to SCHRACK_RIGHT to calculate the left, up, and down neighbors of equal size. In particular, SCHRACK_LEFT differs from SCHRACK_RIGHT by loading the y positions with 0s instead of 1s (using EVENBITMASK instead of ODDBITMASK), and by using subtraction instead of addition. The only difference between procedures SCHRACK_DOWN and SCHRACK_UP, and procedures SCHRACK_RIGHT and SCHRACK_LEFT, respectively, is the replacement of EVENBITMASK by ODDBITMASK and ODDBITMASK by EVENBITMASK.

Using standard Morton codes for square quadtrees, we see that we can find a neighbor by addition if we just skip every other bit in the Morton code. This method does not work directly in the case of the triangle quadtree, although something similar can be made to work. One problem is the lack of a direct correlation between the coordinate system of the decomposition induced by the triangle quadtree and the path array values of the locational codes. Nevertheless, the values of the path array of the location code in a triangle quadtree can be manipulated in an analogous manner to the values of the path array of the location code in a square

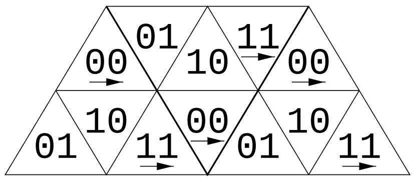

Fig. 4. Examples of rightward transitions that generate a carry (denoted by a rightward pointing arrow) as the neighboring triangles are not siblings.

quadtree, as shown in the next three subsections. For the sake of simplicity, our presentation assumes that the nodes whose neighbors are being sought are at the deepest level in the quadtrees corresponding to the faces. The only modification needed to handle a node at depth i is to add or subtract 2^i instead of 1 when calculating the path array component (i.e., the Morton code) of the location code.

5.2 Rightward Transitions

In this section we consider a transition from a triangle to its right neighbor. Below, we look at the transitions from the different children. Transitions from a `01` child to a `10` child or from a `10` to a `11` child are achieved by adding one when the neighboring triangles are siblings. On the other hand, the triangle quadtree analog of a carry in the square quadtree arises when we make a transition from a `00` child to a `01` child or when we move from a `11` child to a `00` child (see Figure 4). This is the case when the neighboring triangles are not siblings. Making a transition from a `11` child to a `00` child is not a problem because this is handled easily by the use of addition. Basically, we add one to the bit string represented by the path array of the input and the carry automatically updates the parent node. However, moving from a `00` child to a `01` child doesn't work so simply. We want a carry but we don't naturally get one. One way to obtain the carry is to locate and replace all occurrences of `00`s with `11`s, so that either of the following two situations is handled properly :

(1) A carry will be generated if necessary (i.e., the `00` is at the extreme right of the path array of the input).

(2) A carry will be properly propagated (i.e., the `00` is the recipient of a carry).

In both of these situations, we can use simple addition to find the neighbor. Since we have replaced all `00`s with `11`, once the addition has taken place, any `00`s that became `00` (i.e., were affected by the addition) must be set to their proper value, which is `01`, while all `00`s that remained `11` (i.e., were unaffected by the addition) must be reset to their original value, which is `00`.

In order to specifically deal with the `00` case, we introduce the concept of an *idmask*. From a general standpoint, the idmask has two roles:

Table VI. Result of Applying Idmask ABIDXY to an Example Input Value (so that all occurrences of the two-bit pattern with value "AB"are replaced by the two-bit pattern with value "XY")

Idmask	Input=00011011	Input=10000100	Input=11010100
00ID11	11000000	00110011	00000011
01ID11	00110000	00001100	00111100
?0ID11	11001100	11110011	00000011
?1ID11	00110011	00001100	11111100
01ID01	00010000	00000100	00010100
?0ID10	10001000	10100010	00000010
?1ID10	00100010	00001000	10101000

(1) to identify the bit positions that have particular values, and

(2) to aid in marking these bit positions with specific values, not necessarily the same, while leaving the values of the remaining bit positions unchanged.

We use the naming convention ABIDXY for the idmasks where AB denotes the values of the bit pattern pair whose positions of occurrence we seek to identify, and XY denotes the values of the bit pattern pair that we use to mark these positions of occurrence. The idmasks are formed by invoking a procedure MAKE_IDMASK(INPUT,AB,XY) that sets all pairs of bits in *idmask* for which the corresponding bit pairs in the path array component of INPUT have value AB to XY, while the bits corresponding to the other bit pairs are set to 00. The actual idmasks are built by calls to specialized routines of the form MAKE_IDMASK_ABIDXY.[2]

In our example of a rightward movement, we use the idmask 00ID11. In particular, the idmask is used to identify the bit positions where we need to modify the path array value of the input before and after performing the addition. Once these bit positions have been identified (i.e., the bit positions in the path array of the input that have the bit pattern pair value 00), they are marked with the bit pattern pair 11, while the remaining bit positions are left alone. We use the marking pattern 11 because taking its exclusive or with any input sequence ensures that all pairs of bits with value 00 are changed to 11, and all pairs of bits with other bit patterns are left alone, since the exclusive or of any bit value i with 0 is i.

Note that virtually any pattern of bit pairs can be identified by forming the appropriate idmask in constant time. For example, Table VI shows the effect of some example idmasks on a bit string. The idmasks 00ID11, 01ID11, ?0ID11, and ?1ID11 use the marking pair 11 to identify the bit pairs 00, 01, a don't care followed by 0, and a don't care followed by 1, respectively. Of course, other marking pairs can be used as well. In particular, we show 01ID01 which uses the pair 01 to mark the pair 01,

[2]Procedure MAKE_IDMASK can be implemented using a table lookup method that uses the values of the parameters AB and XY to invoke the appropriate routine MAKE_IDMASK_ABIDXY. We do not give the code for MAKE_IDMASK here.

Table VII. Example of Steps in Generation of Idmask 00ID11 for Input Values

Step	Operation	Results		
0	Example Input	00011011	10000100	11010100
1	SHIFT_RIGHT 0	00001101	01000010	01101010
2	1 OR 0 (i.e., Input)	00011111	11000110	11111110
3	2 XOR EVENBITMASK	01001010	10010011	10101011
4	3 AND EVENBITMASK	01000000	00010001	00000001
5	SHIFT_LEFT 4	10000000	00100010	00000010
6	5 OR 4	11000000	00110011	00000011

?0ID10 which uses the pair 10 to mark the pair ?0, and ?1ID10 which uses the pair 10 to mark the pair ?1. These idmasks are used in the remaining sections for leftward and vertical transitions, as well as transitions between neighboring triangles that are in different base triangles of the icosahedron.

In order to gain an understanding of how an idmask is generated, let us examine the generation of 00ID11. Identifying 00 within a given child bit pair whose left and right bits are labeled leftbit and rightbit, respectively, requires a Boolean expression such as NOT(leftbit OR rightbit). Notice that this expression returns TRUE only when both bits are 0 (i.e., FALSE). The sequence of operations given in Table VII shows how this idmask is generated for a given path array. The SHIFT_RIGHT operation aligns every leftbit with every rightbit. Step 2 performs the OR part of our Boolean expression. The XOR in step 3 performs the NOT part of our Boolean expression (EVENBITMASK is used because only the values of the even bits starting at the leftmost position are relevant at this point). The AND removes any “noise” left in the odd bits. This completes the Boolean expression (i.e., step 4 in Table VII), but doesn't give us the pair of 1s that we wanted. In particular, at this point, our marking pattern is 01, which we wish to change to 11. This is done by applying two more operations, as follows: A SHIFT_LEFT moves all the right bits into the left bit position. A final OR combines our unshifted bits (i.e., the result of step 4) with our shifted bits to yield the marking pattern we want (i.e., 00ID11). This process is implemented by procedure MAKE_IDMASK_00ID11.

Algorithm 8.

```
path_array procedure MAKE_IDMASK_00ID11(P);
/* Return the 00ID11 idmask corresponding to the path array
component of location code P. */
begin
  value pointer location_code P;
  path_array 00ID11;
  /* Identify the location of all 00s */
  00ID11 ← OR(SHIFT_RIGHT(CODE(P)),CODE(P));
  00ID11 ← XOR(00ID11,EVENBITMASK);
  00ID11 ← AND(00ID11,EVENBITMASK);
  /* Duplicate bits in 00ID11 */
  00ID11 ← OR(00ID11,SHIFT_LEFT(00ID11));
  return(00ID11);
end;
```

Now let us return to our task of finding a right neighbor of equal size. This is achieved using the following strategy, which is implemented by procedure `CONSTANT_RIGHT` given below. We first compute idmask `00ID11` by invoking procedure `MAKE_IDMASK_00ID11`. Next, we prepare for the addition step by taking the `XOR` of idmask `00ID11` with the input path array. When the adjacent triangles are siblings, the neighbor is obtained by simple addition, and there is no carry. When the adjacent triangles are not siblings, the carry that is generated by the addition process is used to obtain the correct path array values for the location code of the adjacent triangle. This situation arises whenever the current child is either `00` or `11` (recall Figure 4). When the current child is `11`, the necessary carry is generated or propagated by the addition process. However, when the current child is `00`, no carry is generated or propagated by the addition process, and thus we have to artificially create a situation where a carry is generated or propagated.

This situation is created by using idmask `00ID11` to identify all `00`s in the path array of the input and to replace them with `11`s before performing the addition of `1`, so that the carry will be generated or propagated if necessary. After the addition, we must take care of the following two special cases:

(1) `11`s not affected by the addition (which were originally `00`s) must be changed back to `00`; and

(2) `11`s affected by the addition, and which thus became `00`s (again, only the ones that were originally `00`s) must be changed to `01` (because `00` plus one is `01`).

The handling of these special cases is also facilitated by use of the idmask `00ID11`. In particular, once the addition has taken place, `CONSTANT_RIGHT` must perform the following two tasks in order to work correctly:

(1) identify the occurrences of `11` in the result that were not affected by the addition or the propagation of a carry (they must be reset to `00`); and

(2) identify the occurrences of `00` in the result that were generated by an addition or a propagation of a carry (they are set to `01`).

THEOREM 1. *Tasks 1 and 2 are correctly performed by procedure `CONSTANT_RIGHT`.*

PROOF. The first task is performed by taking the `XOR` of the idmask with the result of the addition, thereby creating a bit pattern that we term t. This has the effect of leaving all pairs of bits that were not originally `00` alone, since the exclusive or of any bit value i with `0` is i. This also has the effect of resetting to `00` all `11`s at positions in the path array of the input that originally contained `00` (which is desired) and resetting to `11` all `00`s at positions in the path array of the input that originally contained `00`.

Once the first task has been completed, perform the second task. In particular, all 11s that were affected by the addition and thus became 00s (again, only the ones that were originally 00s) must be changed to 01 (because 00 plus one is 01). This is achieved by constructing a mask that has a 11 at every pair of positions in the path array of the input that did not contain 00 (obtained by taking the complement of idmask 00ID11). Next, we OR this mask with EVENBITMASK (an alternating bit pattern starting with 0 at its left end; that is, 010101. . .), which results in marking the even positions, starting at the leftmost position, in the path array of the input that were part of the original 00 pair with a 01. Taking the AND of the resulting mask with t yields the desired result. □

Algorithm 9.

```
procedure CONSTANT_RIGHT(P);
/* Determine the location code of the right neighbor of equal
size of the triangle quadtree node with location code P. This
involves setting the CODE field of P. */
begin
  value pointer location_code P;
  path_array 00ID11;
  00ID11 ← MAKE_IDMASK(P,00,11);
  /* Change 00s to 11s while leaving the rest alone */
  CODE(P) ← XOR(CODE(P),00ID11);
  /* Add one (move right) */
  CODE(P) ← CODE(P)+1;
  /* Restore unchanged 00s */
  CODE(P) ← XOR(CODE(P),00ID11);
  /* Fix-up 00s that got hit with a carry */
  CODE(P) ← AND(CODE(P),OR(COMPLEMENT(00ID11),EVENBITMASK));
end;
```

Table VIII shows the effects of procedure CONSTANT_RIGHT on various bit pattern pairs. The four columns under the heading Without Carry show what happens to each of the four possible child bit pattern pairs when these bits are not involved in a carry. It is important that the final bit pattern pair values match the initial bit pattern pair values for these four columns. For example, suppose we want to know what happens to the bit pattern pair 01 in the location code with path array value ??0110?? during the execution of procedure CONSTANT_RIGHT. Since 01 is followed by 10, it is impossible for the addition of one to the path array value of the input to have any effect on 01. In other words, 01 cannot be the recipient of an incoming carry (from the right). Therefore, the effect of procedure CONSTANT_RIGHT on it and any other bit pattern pair values that are followed by a bit pattern pair that does not generate a carry are found in the second column of Table VIII (titled Without Carry).

As another example, suppose we want to know what happens to the bit pattern pair 10 in the location code with path array value ??101111 during the execution of procedure CONSTANT_RIGHT. Since 10 is followed by all 1s, adding one to the path array value will change the 10 to 11. In other words,

Table VIII. Effect of procedure CONSTANT_RIGHT on Different Bit Pattern Pair Values (depending on whether there is an incoming carry from the right)

Action	Without Carry				With Carry			
Initial Bits	00	01	10	11	00	01	10	11
XOR 00ID11	11	01	10	11	11	01	10	11
Add One	11	01	10	11	00	10	11	00
XOR 00ID11	00	01	10	11	11	10	11	00
Fixup 00s	00	01	10	11	01	10	11	00

`10` is the recipient of an incoming carry (from the right). Therefore, the effect of procedure `CONSTANT_RIGHT` on it and any other bit pattern pair values that are followed by a bit pattern pair that does generate a carry are found in the third column of Table VIII (titled `With Carry`). Notice that in this case the final bit pattern pair values are one greater than the initial bit pattern pair values (`11` becomes `00`).

As an example of the action of procedure `CONSTANT_RIGHT`, let us find the right neighbor of the triangle whose location code has path array value `00011100`. Let `RCODE` refer to this path array value. `00ID11` is `11000011`, since both `RCODE[1]` and `RCODE[4]` have value `00`. The first `XOR` changes `RCODE` to `11011111`. Adding one changes `RCODE` to `11100000`. The second `XOR` changes `RCODE` to `00100011`. The operation `OR(COMPLEMENT(00ID11),EVENBITMASK)` yields `01111101`. The final `AND` changes `RCODE` to `00100001`. This example is illustrated in Figure 5(a).

5.3 Leftward Transitions

In this section we consider a transition from a triangle to its left neighbor. This transition differs from a rightward transition in that instead of adding `1` to the path array value of the location code and propagating a carry when moving between triangles that are not siblings, we subtract `1` from the path array value of the location code and propagate a borrow when moving between triangles that are not siblings.[3]

Below, we look at leftward transitions from the different children. The cases corresponding to a transition from a `10` child to a `01` child or from a `11` to a `10` child are simple, as they are achieved by subtracting one when the neighboring triangles are siblings. On the other hand, the leftward movement analog of a carry for the rightward movement arises when we make a transition from a `01` child to a `00` child or when we move from a `00` child to a `11` child (see Figure 6). This is the case when the neighboring triangles are not siblings. Making a transition from a `00` child to a `11` child is not a problem because this is handled easily by the use of subtraction. Basically, we subtract one from the bit string represented by the path array and the borrow automatically updates the parent node. However, moving

[3]We could also implement the subtraction by adding -1, using twos complement arithmetic, in which case the discussion would be in terms of additions and carries rather than subtractions and borrows. In the interest of clarity, we use the latter.

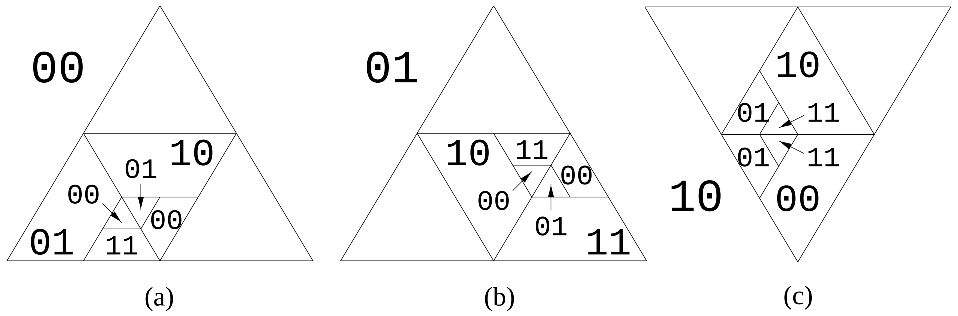

Fig. 5. Examples showing how to find neighbors of equal size: (a) right neighbor of 00011100; (b) left neighbor of 01110001; (c) vertical neighbor of 10100111.

from a 01 child to a 00 child doesn't work so simply. We want a borrow but we don't naturally get one. One way to obtain the borrow is to locate and replace all occurrences of 01s with 00s, so that either of the following two situations is handled properly :

(1) A borrow will be generated if necessary (i.e., the 01 is at the extreme right of the path array of the input).

(2) A borrow will be properly propagated (i.e., the 01 is the recipient of a borrow).

In both of these situations, we can use simple subtraction to find the neighbor. Since we replaced all 01s with 00, once the subtraction has taken place, any 01s that became 11 (i.e., were affected by the subtraction) must be set to their proper value, which is 00, while all 01s that remained 00 (i.e., were unaffected by the subtraction) must be reset to their original value, which is 01.

In order to specifically deal with the 01 case, we once again make use of the concept of an *idmask*. As in the case of the rightward movement, the idmask identifies the bit positions where we need to modify the path array value of the input before and after performing the subtraction. However, unlike the rightward movement, we must identify the bit positions in the path array of the input that have value 01 and change them to 00 prior to the subtraction, while leaving all other bit pattern pairs alone. This is not done easily if we use the marking pattern of 11, as we did in the case of a rightward movement, since now our goal is to change a bit pattern pair whose two values are not the same. The task is more easily accomplished by observing that the result of taking the exclusive or of bit pattern pair 01 with bit pattern pair 01 is 00, while the result of taking the exclusive or of all other bit pattern pairs with bit pattern pair 00 leaves them unchanged. Thus, for leftward transitions we use an idmask called 01ID01 with a marking pattern of 01 for all occurrences of 01 in the path array of the input. It is formed by a call to MAKE_IDMASK_01ID01, given below.

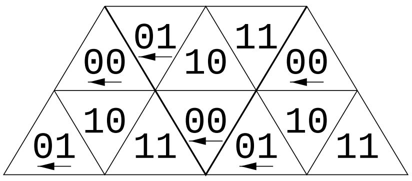

Fig. 6. Examples of leftward transitions that generate a borrow (denoted by a leftward pointing arrow), as the neighboring triangles are not siblings.

Algorithm 10.

```
path_array procedure MAKE IDMASK 01ID01(P);
/* Return the 01ID01 idmask corresponding to the path array
component of location code P. */
begin
  value pointer location code P;
  path_array 01ID01;
  /* Identify the location of all 01s */
  01ID01 ← AND(COMPLEMENT(CODE(P)),ODDBITMASK);
  01ID01 ← SHIFT_RIGHT(01ID01);
  01ID01 ← AND(01ID01,CODE(P));
  return(01ID01);
end;
```

Finding the left neighbor of equal size is achieved using procedure CONSTANT_LEFT, given below. The difference from procedure CONSTANT_RIGHT is the use of idmask 01ID01 instead of 00ID11 and subtraction instead of addition. After subtraction, CONSTANT_LEFT must perform the following two tasks in order to work correctly:

(1) identify the occurrences of 00 in the result that were not affected by the subtraction or the propagation of a borrow (they must be reset to 01); and

(2) identify the occurrences of 11 in the result that were generated by a subtraction or a propagation of a borrow (they are set to 00).

THEOREM 2. *Tasks 1 and 2 are performed correctly by procedure CONSTANT_LEFT.*

PROOF. The first task is performed by taking the XOR of the idmask with the result of the subtraction, thereby creating a bit pattern that we term t. This has the effect of leaving all pairs of bits that were not originally 01 alone, since the exclusive or of any bit value i with 0 is i. This also has the effect of resetting to 01 all 00s at positions in the path array of the input that originally contained 01 (which is desired) and resetting to 10 all 11s at positions in the path array of the input that originally contained 01.

Once the first task has been completed, perform the second task. In particular, all 00s that were affected by the subtraction and thus became 11s (again, only the ones that were originally 01s) must be changed to 00 (because 01 minus one is 00). This is achieved by constructing a mask that has a 11 at every pair of positions in the path array of the input that did not contain 01, and a 01 in the positions that did contain 01 (obtained by taking the COMPLEMENT of the result of applying SHIFT_LEFT by one bit position to idmask 01ID01). Taking the AND of the resulting mask with *t* yields the desired result. □

Algorithm 11.

```
procedure CONSTANT_LEFT(P);
/* Determine the location code of the left neighbor of equal
size of the triangle quadtree node with location code P. This
involves setting the CODE field of P. */
begin
  value pointer location_code P;
  path_array 01ID01;
  01ID01 ← MAKE_IDMASK(P,01,01);
  /* Change 01s to 00s while leaving the rest alone */
  CODE(P) ← XOR(CODE(P),01ID01);
  /* Subtract one (move left) */
  CODE(P) ← CODE(P)-1;
  /* Restore unchanged 01s */
  CODE(P) ← XOR(CODE(P),01ID01);
  /* Fix-up 01s that got hit with a borrow */
  CODE(P) ← AND(CODE(P),COMPLEMENT(SHIFT_LEFT(01ID01)));
end;
```

As an example of the action of procedure CONSTANT_LEFT, let us find the left neighbor of the triangle whose location code has path array value 01110001. Let LCODE refer to this path array value. 01ID01 is 01000001, since both both LCODE[1] and RCODE[4] have value 01. The first XOR changes LCODE to 00110000. Subtracting one changes LCODE to 00101111. The second XOR changes LCODE to 01101110. The operation COMPLEMENT(SHIFT_LEFT(01ID01)) yields 01111101. The final AND changes LCODE to 01101100. This example is illustrated in Figure 5(b).

5.4 Vertical Transitions

In this section we consider a transition from a triangle to its vertical neighbor. This is a very simple transition because once we locate the nearest common ancestor (i.e., the parent of the smallest containing sibling triangles of the neighboring triangles), the reflection process for finding the neighbor results in no change in any of the other elements of the path array of the input. In particular, recall from Figures 2(a) and 2(b) that with exception of the path array component corresponding to the sibling triangles that contain the two neighbors, the path array value of the neighbor is the same as the path array value of the triangle whose vertical neighbor is being sought. We made use of this property when we calculated vertical neighbors in Section 3.

The vertical transition differs from the rightward and leftward transitions in that the path array values of the inputs do not change, except for the transition between sibling triangles. In particular, we need to make one, and only one, transition from the least significant 00 child (i.e., rightmost in the path array of the input) to the least significant 10 child or vice versa (i.e., from the least significant 10 child to the least significant 00 child). From an implementation standpoint, making a vertical transition is quite simple. All we need to do is identify the rightmost ?0 child and complement the left bit of its bit pattern pair value. All remaining bit pattern pairs are left alone.

In order to facilitate the identification of the rightmost ?0 case, we once again make use of the concept of an *idmask*. In this case, we use the idmask ?0ID10, which identifies the bit positions in the path array of the input with value ?0 and marks them with 10. We use the marking pattern 10 because we want to complement the left bit of a bit pattern pair value, and this is easily done with the aid of an exclusive or operation as the exclusive or of any bit value i with 1 the complement of i. Idmask ?0ID10 is formed by a call to MAKE_IDMASK_?0ID10, given below.

Algorithm 12.

```
path_array procedure MAKE_IDMASK_?0ID10(P);
/* Return the ?0ID10 idmask corresponding to the path array
component of location code P. */
begin
  value pointer location_code P;
  path_array ?0ID10;
  /* Identify the location of all 00s and 10s */
  ?0ID10 ← AND(COMPLEMENT(CODE(P)),EVENBITMASK);
  ?0ID10 ← SHIFT_LEFT(?0ID10);
  return(?0ID10);
end;
```

Finding the vertical neighbor of equal size is achieved using procedure CONSTANT_VERTICAL, given below. It must complement the left bit of the rightmost ?0 in the original input.

THEOREM 3. *Procedure CONSTANT_VERTICAL complements the left bit of the rightmost ?0 in the original input.*

PROOF. CONSTANT_VERTICAL first computes idmask ?0ID10 by invoking procedure MAKE_IDMASK_?0ID10. Next, it creates a new mask m from ?0ID10 that is zero at all bit positions with the exception of the rightmost 10. This is achieved by taking the COMPLEMENT of ?0ID10. The result is a mask n that contains 11 in all bit-pair positions to the right of the rightmost 10 of ?0ID10, which itself has become 01 in n. Adding 1 to n, thereby resulting in p, means that all 11s to the right of the rightmost 01 have become 00s, while the rightmost 01 has become a 10. All other bit-pair positions in n are unchanged by the addition. The desired mask m is now obtained by taking the AND of p and ?0ID10. This works because all items to the left of the rightmost 10 in p are the complements of the corresponding items in ?0ID10, while all items to the right of the rightmost 10 in p are 0. The final step is to take the XOR of m with the original input value. This has the correct effect of complementing the left bit of the rightmost ?0 in the original input value, since the exclusive or of any bit value i with 1 is the complement of i. This process yields the same effect as the column labeled "Vert" in Table II in Section 3.2. □

Algorithm 13.

```
procedure CONSTANT_VERTICAL(P);
/* Determine the location code of the vertical neighbor of equal
size of the triangle quadtree node with location code P. This
involves setting the CODE field of P. */
begin
  value pointer location_code P;
  path_array ?0ID10,MASK;
  ?0ID10 ← MAKE IDMASK(P,?0,10);
  MASK ← COMPLEMENT(?0ID10);
  /* Use carry to find what to update */
  MASK ← MASK+1;
  /* Clear out everything but carry */
  MASK ← AND(MASK,?0ID10);
  /* Update the path array */
  CODE(P) ← XOR(CODE(P),MASK);
end;
```

As an example of the action of procedure CONSTANT_VERTICAL, find the vertical neighbor of the triangle whose location code has path array value 10100111. Let VCODE refer to this path array value. ?0ID10 is 10100000, since both VCODE[1] and VCODE[2] have value ?0. The operation COMPLEMENT(?0ID10) yields 01011111 stored in variable MASK. Adding one to MASK yields 01100000. Applying AND(MASK,?0ID10) changes MASK to 00100000. The final XOR of MASK with VCODE changes VCODE to 10000111. This example is illustrated in Figure 5(c).

5.5 Transitions Across Different Faces of the Icosahedron

Transitions between different base triangles of the icosahedron are relatively simple. This situation arises if the addition steps in procedures CONSTANT_RIGHT and CONSTANT_VERTICAL generated a carry past the leftmost end of the path array of the input or if the subtraction step in procedure CONSTANT_LEFT generated a borrow past the leftmost end of the path array of the input. In this case, some sort of carry (borrow) or overflow indicator will be set. Testing this flag is achieved by a simple one-cycle machine instruction on most computer architectures. Alternatively, we could allocate one additional bit at the extreme left of the path array of the input to indicate when an 'overflow' condition has occurred. For example, consider the location code 0011110011. If we reserve an overflow bit, the code becomes 00011110011. The result of applying CONSTANT_RIGHT to 00011110011 yields 10100000100. Since the overflow bit is 1, we need to update the identity of the base triangle for this example. This is achieved in constant time by making use of Table IV, which is described in Section 4.

Vertical transitions between different faces of the icosahedron as well as left and right transitions between nodes corresponding to the faces of the icosahedron labeled 05 to 14 as shown in Figure 1 are straightforward, in the sense that there is no change in the algorithms. However, special care must be taken when making left and right transitions between nodes corresponding to the faces of the icosahedron labeled 00 to 04 and 15 to 19. In Section 4, we solved this problem by making use of Table V. We now want to obtain the same result in worst-case constant time. The issue here is that the left and right neighbors are “mirror reflections.” In particular, recall that in the case of a right neighbor, 00 stays 00, while 11 reflects to 01, 10 and 01 cannot occur along the right edge of a node. Similarly, in the case of a left neighbor, 00 stays 00, while 01 reflects to 11, 10 and 11 cannot occur along the left edge of a node.

These situations are handled in a similar manner to vertical transition, in the sense that we make use of reflection. The difference is that we must perform the reflection for all occurrences of 11 in the case of right neighbors and all occurrences of 01 in the case of left neighbors. These situations are identified by complementing the left bit of the bit pattern value of each ?1 child. All remaining bit pattern pairs are left alone.

In order to facilitate the identification of all occurrences of ?1, we once again make use of the concept of an *idmask*. In this case, we use the idmask ?1ID10 that identifies the bit positions in the path array of the input with value ?1 and marks them with 10. Idmask ?1ID10 is formed by a call to MAKE_IDMASK_?1ID10, given below. Note the similarity to idmask ?0ID10 used in finding vertical neighbors (procedure CONSTANT_VERTICAL).

Algorithm 14.

```
path_array procedure MAKE IDMASK ?1ID10(P);
/* Return the ?1ID10 idmask corresponding to the path array
component of location code P. */
begin
  value pointer location_code P;
  path_array ?1ID10;
  /* Identify the location of all 01s and 11s */
  ?1ID10 ← AND(CODE(P),EVENBITMASK);
  ?1ID10 ← SHIFT_LEFT(?1ID10);
  return(?1ID10);
end;
```

The reflection is implemented by procedure CONSTANT_REFLECTION, given below. It is important to note that we only use procedure CONSTANT_REFLECTION when the overflow bit is 1, which indicates that the nearest common ancestor is actually the entire sphere. The correctness of CONSTANT_REFLECTION depends on its proper handling of both left and right neighbors.

THEOREM 4. *Procedure CONSTANT_REFLECTION works correctly for both left and right neighbors.*

PROOF. In the left neighbor case, since we are on the extreme left edge of one of the triangles of the faces of the icosahedron, the path array value can only contain bit pattern pairs with values 00 and 01. Thus all 01s are "marked" by ?1ID10 (with the pattern 10). Therefore, one application of XOR to the input with ?1ID10 changes all 01s to 11s, as desired. Similarly, in the right neighbor case, since we are on the extreme right edge of one of the triangles of the faces of the icosahedron, the path array value can only contain bit pattern pairs with values 00 and 11. Thus all 11s are "marked" by ?1ID10 (with the pattern 10). Therefore, one application of XOR to the input with ?1ID10 changes all 11s to 01s, as desired. □

Algorithm 15.

```
procedure CONSTANT_REFLECTION(P);
/* Determine the location code of the right or left neighbor of
equal size of the triangle quadtree node corresponding to a face
of the icosahedron labeled 00 to 04 and 15 to 19 with location
code P. This involves setting the CODE field of P. */
begin
  value pointer location_code P;
  path_array ?1ID10;
  ?1ID10 ← MAKE IDMASK(P,?1,10);
  /* Update the path array */
  CODE(P) ← XOR(CODE(P),?1ID10);
end;
```

We now present the complete algorithms for finding right, left, and vertical neighbors. They work regardless of whether the neighbors are in the same or different faces of the icosahedron. The algorithms are encoded by procedures EXT_CONSTANT_LEFT, EXT_CONSTANT_RIGHT, and EXT_CONSTANT_VERTICAL. Procedure EXT_CONSTANT_RIGHT first invokes procedure CONSTANT_RIGHT. If the overflow bit is 1, we need to update the child type of the root; otherwise we are done. We can update the root value using Table IV. If the current child of the root corresponds to a face of the icosahedron labeled 00 to 04 or 15 to 19, we discard the result of CONSTANT_RIGHT (only the overflow condition is significant) and invoke procedure CONSTANT_REFLECTION with our original input location code. At this point we are done, as we have found the right neighbor of the input location code. Procedure EXT_CONSTANT_LEFT, not given here (see Lee and Samet [1998]), is equivalent to procedure EXT_CONSTANT_RIGHT once we replace the call to CONSTANT_RIGHT by a call to CONSTANT_LEFT, as well as the constant RIGHT by LEFT. Procedure EXT_CONSTANT_VERTICAL just needs to call procedure CONSTANT_VERTICAL, and then update the root value using Table IV if overflow occurs.

Algorithm 16.

```
procedure EXT_CONSTANT_RIGHT(P);
/* Determine the location code of the right neighbor of equal
size of the triangle quadtree node with location code P. The
routine works regardless of whether or not the neighbor is on
the same face of the icosahedron. This involves setting the CODE
field of P. */
begin
```

```
    value pointer location_code P;
    pointer location_code NEWP;
    preload integer array NEXTTOP[0:2][0:19] with Table IV;
    NEWP ← create(location code);
    CODE(NEWP) ← CODE(P);
    LEV(NEWP) ← LEV(P);
    /* Use top of CODE(NEWP) as overflow space */
    CODE(NEWP)[0] ← 0;
    /* Find standard right neighbor */
    CONSTANT_RIGHT(NEWP);
    /* Check for overflow */
    if CODE(NEWP)[0]=0 then
      /* Restore root position to original value */
      CODE(NEWP)[0] ← CODE(P)[0]
    else
    begin
      /* Check for nodes 0 to 4 and 15 to 19 */
      if not(4<CODE(P)[0] and CODE(P)[0]<15) then
      begin
        /* Get a new copy of the original path array value */
        CODE(NEWP) ← CODE(P);
        /* Use reflection to get the neighbor */
        CONSTANT_REFLECTION(NEWP);
      end;
      /* Set root position to appropriate neighbor */
      CODE(NEWP)[0] ← NEXTTOP[ 'RIGHT'][CODE(P)[0]];
    end;
    /* Set CODE(P) to the new path array value */
    CODE(P) ← CODE(NEWP);
  end;
```

Algorithm 17.

```
  procedure EXT_CONSTANT_VERTICAL(P);
  /* Determine the location code of the vertical neighbor of equal
  size of the triangle quadtree node with location code P. The
  routine works regardless of whether or not the neighbor is on
  the same face of the icosahedron. This involves setting the CODE
  field of P. */
  begin
    value pointer location_code P;
    pointer location_code NEWP;
    preload integer array NEXTTOP[0:2][0:19] with Table IV;
    NEWP ← create(location_code);
    CODE(NEWP) ← CODE(P);
    LEV(NEWP) ← LEV(P);
    /* Use top of CODE(NEWP) as overflow space */
    CODE(NEWP)[0] ← 0;
    /* Find standard vertical neighbor */
    CONSTANT_VERTICAL(NEWP);
    /* Check for overflow */
    if CODE(NEWP)[0]=0 then
      /* Restore root position to original value */
      CODE(NEWP)[0] ← CODE(P)[0]
```

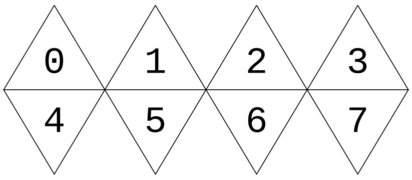

Fig. 7. Example showing the top-level triangle faces of an octahedron.

```
    else
      /* Set root position to appropriate neighbor */
      CODE(NEWP)[0] ← NEXTTOP[ ‘VERTICAL’][CODE(P)[0]];
    /* Set CODE(P) to the new path array value */
    CODE(P) ← CODE(NEWP);
  end;
```

6. NEIGHBOR FINDING USING OCTAHEDRA AND TETRAHEDRA

In this section we briefly describe how to perform neighbor finding when the sphere is approximated by other Platonic solids with triangular faces. Section 6.1 describes the modifications to the algorithms for the icosahedron needed for the octahedron while Section 6.2 deals with the tetrahedron.

6.1 Octahedron

Approximating a sphere by an octahedron requires eight of our triangle quadtrees. We label the eight nodes corresponding to the roots of the quadtrees of the faces of the octahedron using a 4-bit code ranging from 0000 (decimal 0) to 0111 (decimal 7). We could have fit the eight values into just 3 bits, but we decided to use an even number of bits because the machine word length is always an even number of bits. The order in which the faces of the octahedron are numbered isn't important since tables will be used. Thus we have numbered the faces using a simple left-to-right and top-to-bottom order (see Figure 7). Our numbering scheme has the property that triangles 0 to 3 are tip-up and 4 to 7 are tip-down.

The only modification with respect to step two of the algorithm in Section 3 is the use of a different relation NEXTOCT (Table IX) to indicate how to update CODE[0]. It summarizes the actions for all possible neighbors from Figure 7 and replaces relation NEXTTAB in the algorithm for this case. This relation is used only when the nearest common ancestor from step one is the entire sphere.

We now present the complete constant time algorithms for finding right, left, and vertical neighbors, analogous to those in Section 5.5 for the icosahedron, in the sense that they work regardless of whether the neighbors are on the same or different faces of the octahedron. The algorithms are encoded by procedure OCT_CONSTANT_LEFT, OCT_CONSTANT_RIGHT, and OCT_CONSTANT_VERTICAL. Procedure OCT_CONSTANT_RIGHT first invokes procedure CONSTANT_RIGHT. If the overflow bit is 1, we need to update the child type of the root; otherwise we are done. We update the root

Table IX. NEXTOCT(Neighbor_Direction,Child_Type) indicating neighbors of triangles corresponding to faces of the octahedron

	Neighbor Direction		
Child Type	Left	Right	Vert
0	3	1	4
1	0	2	5
2	1	3	6
3	2	0	7
4	7	5	0
5	4	6	1
6	5	7	2
7	6	4	3

value using Table IX. Also, we throw away the result of CONSTANT_RIGHT (only the overflow condition is significant) and invoke procedure CONSTANT_REFLECTION with our original input location code. We are now done, as we have found the right neighbor of the input location code. Procedure OCT_CONSTANT_LEFT, not given here (see Lee and Samet [1998]), is equivalent to procedure OCT_CONSTANT_RIGHT once we replace the call to CONSTANT_RIGHT by a call to CONSTANT_LEFT, as well as the constant RIGHT by LEFT. Procedure OCT_CONSTANT_VERTICAL, not given here (see Lee and Samet [1998]), just needs to call procedure CONSTANT_VERTICAL and then update the root value using Table IX if overflow occurs. It is identical to procedure EXT_CONSTANT_VERTICAL once we replace table NEXTTOP (Table IV) by NEXTOCT (Table IX).

Algorithm 18.

```
procedure OCT_CONSTANT_RIGHT(P);
/* Determine the location code of the right neighbor of equal
size of the triangle quadtree node with location code P. The
routine works regardless of whether or not the neighbor is on
the same face of the octahedron. This involves setting the CODE
field of P. */
begin
  value pointer location_code P;
  pointer location_code NEWP;
  preload integer array NEXTOCT[0:2][0:7] with Table IX;
  NEWP ← create(location code);
  CODE(NEWP) ← CODE(P);
  LEV(NEWP) ← LEV(P);
  /* Use top of CODE(NEWP) as overflow space */
  CODE(NEWP)[0] ← 0;
  /* Find standard right neighbor */
  CONSTANT_RIGHT(NEWP);
  /* Check for overflow */
  if CODE(NEWP)[0]=0 then
    /* Restore root position to original value */
    CODE(NEWP)[0] ← CODE(P)[0]
  else
  begin
```

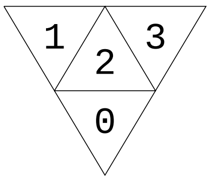

Fig. 8. Example showing the top-level triangle faces of a tetrahedron.

```
      /* Get a new copy of the original path array value */
      CODE(NEWP) ← CODE(P);
      /* Use reflection to get the neighbor */
      CONSTANT_REFLECTION(NEWP);
      /* Set root position to appropriate neighbor */
      CODE(NEWP)[0] ← NEXTOCT[ ‘RIGHT’][CODE(P)[0]];
    end;
    /* Set CODE(P) to the new path array value */
    CODE(P) ← CODE(NEWP);
  end;
```

6.2 Tetrahedron

Approximating a sphere by a tetrahedron requires four of our triangle quadtrees. We label the four nodes corresponding to the roots of the quadtrees of the faces of the tetrahedron using a 2-bit code ranging from 00 (decimal 0) to 11 (decimal 3). The order in which the triangle faces of the tetrahedron are numbered isn’t important, since tables will be used. Thus we have numbered the faces using the numbering scheme of Figure 2 (see Figure 8).

The only modification with respect to step two of the algorithm in Section 3 is the use of a different relation NEXTTET (Table X) to indicate how to update CODE[0]. It summarizes the actions for all possible neighbors from Figure 8 and replaces relation NEXTTAB in the algorithm for this case. This relation is used only when the nearest common ancestor from step one is the entire sphere.

If we examine the triangle adjacencies for the tetrahedron (see Figure 9), we notice that some of the transitions result in a “flipped” result. Compensating for this “flipped” result isn’t a significant problem because procedure CONSTANT_REFLECTION already does the required work. We just need to make sure that we call CONSTANT_REFLECTION whenever we make a transition between faces 0 and 1, 0 and 3, or 1 and 3.

We now present the complete constant time algorithms for finding right, left, and vertical neighbors, analogous to those in Section 5.5 for the icosahedron, in the sense that they work regardless of whether the neighbors are on the same or different faces of the tetrahedron. The algorithms are encoded by procedure TET_CONSTANT_LEFT, TET_CONSTANT_RIGHT, and TET_CONSTANT_VERTICAL. Procedure TET_CONSTANT_RIGHT first invokes procedure CONSTANT_RIGHT. If the overflow bit is 1, we need to update the child type of the root; otherwise we are done. We can update the root value using Table X. If the current child of the root corresponds to the

Table X. NEXTTET(Neighbor_Direction,Child_Type) indicating neighbors for triangles corresponding to faces of the tetrahedron

	Neighbor Direction		
Child Type	Left	Right	Vert
0	1	3	2
1	0	2	3
2	1	3	0
3	2	0	1

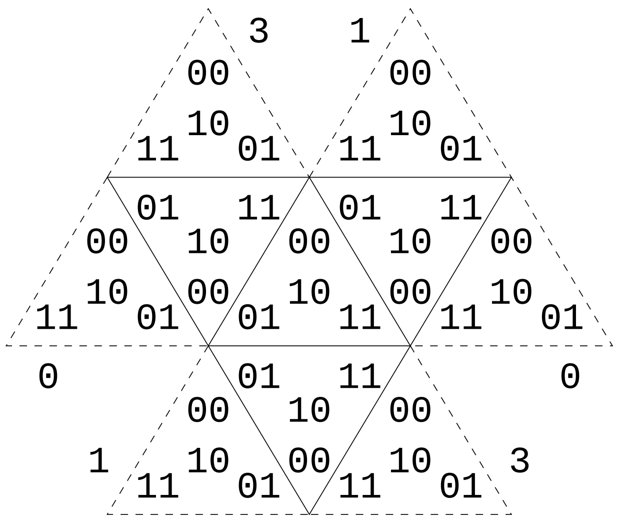

Fig. 9. Example showing triangle adjacencies of the tetrahedron.

faces of the tetrahedron labeled 0 or 3,we invoke procedure CONSTANT_REFLECTION with the current location code. We are now done, as we have found the right neighbor of the input location code. Procedure TET_CONSTANT_LEFT, not given here (see Lee and Samet [1998]), is equivalent to procedure TET_CONSTANT_RIGHT once we replace the call to CONSTANT-_RIGHT by a call to CONSTANT_LEFT, as well as the constant RIGHT by LEFT. We also check for children 0 and 1 instead of 0 and 3. Procedure TET_CONSTANT_VERTICAL, not given here (see Lee and Samet [1998]), is also equivalent to procedure TET_CONSTANT_RIGHT once we replace the call to CONSTANT_RIGHT by a call to CONSTANT_VERTICAL, as well as the constant RIGHT by VERTICAL. We also check for children 1 and 3 instead of 0 and 3.

Algorithm 19.

```
procedure TET_CONSTANT_RIGHT(P);
/* Determine the location code of the right neighbor of equal
size of the triangle quadtree node with location code P. The
routine works regardless of whether or not the neighbor is on
the same face of the tetrahedron. This involves setting the CODE
field of P. */
```

```
begin
  value pointer location_code P;
  pointer location_code NEWP;
  preload integer array NEXTTET[0:2][0:3] with Table X;
  NEWP ← create(location code);
  CODE(NEWP) ← CODE(P);
  LEV(NEWP) ← LEV(P);
  /* Use top of CODE(NEWP) as overflow space */
  CODE(NEWP)[0] ← 0;
  /* Find standard right neighbor */
  CONSTANT_RIGHT(NEWP);
  /* Check for overflow */
  if CODE(NEWP)[0]=0 then
    /* Restore root position to original value */
    CODE(NEWP)[0] ← CODE(P)[0]
  else
  begin
    /* Check for nodes 0 or 3 */
    if CODE(P)[0]=0 or CODE(P)[0]=3 then
      /* Use reflection to get the neighbor */
      CONSTANT_REFLECTION(NEWP);
    /* Set root position to appropriate neighbor */
    CODE(NEWP)[0] ← NEXTTET[ 'RIGHT'][CODE(P)[0]];
  end;
  /* Set CODE(P) to the new path array value */
  CODE(P) ← CODE(NEWP);
end;
```

7. FINDING NEIGHBORS OF GREATER OR EQUAL SIZE

The algorithms in Sections 3–6 assume that the neighbors are of equal size. When the neighbors are not of equal size, we need to do a bit more work. In essence, given node P, our algorithms calculate the address of a neighbor Q in direction D of equal size. This is not a problem if all of the nodes of the quadtrees are of equal size. In general, however, there is no guarantee that such a neighbor Q actually exists if nodes can be of differing sizes. As mentioned in Section 1, the nodes are usually kept in a list L that is sorted by numbers formed by concatenating the base triangle number with the path array value and the depth from left to right.

If Q is not a member of L, there are two possibilities. The first is that the actual neighboring node of P in direction D is greater in size than P. In this case, we find it by returning the node associated with the largest value in L that is less than or equal to the value associated with Q. The second possibility arises when there are many nodes adjacent to P in direction D. In this case, there is no single neighboring node, and we return the analog of a nonleaf node in a conventional quadtree at the same depth as P with the same path array value as Q.

It is important to note that the calculation of the neighbor of equal size is achieved in worst-case constant time. The calculation of the neighboring node when all sizes are permitted requires a search through the list L. This search is speeded up by maintaining L using an index such as a B-tree [Comer 1979]. In fact, this is how the list is usually implemented (e.g., Abel [1984]). In this case, the search takes time logarithmic in the size of L, which is the total number of nodes in the triangle hierarchy.

8. CONCLUSIONS AND FUTURE WORK

We have described a triangle coding scheme that provides a new and worst-case constant time in which to navigate between adjacent triangles in a hierarchical triangle mesh, where the triangles are obtained by a recursive quadtree-like subdivision of the underlying space into four equilateral triangles. We did not address other operations such as determining whether two triangles are adjacent, but this can be accomplished in constant time using our coding scheme. Also, our method is well suited to operations such as finding all triangles that connect any two points of the sphere [Goodchild and Yang 1992].

Our navigation algorithms are given in the context of a sphere approximated by an icosahedron, octahedron, or tetrahedron represented by a collection of quadtree triangle meshes. The only difference is the mechanism to handle the case where the neighboring triangles are in the meshes of different faces of the polyhedron. The algorithms are very efficient, as they only require a few bit manipulation operations, which can be implemented in hardware using just a few machine language instructions.

It is interesting to observe that Schrack's method could also be used to reduce the execution time of the algorithm presented by Otoo and Zhu [1993], as they treat triangle quadtrees as two special subcases of a square quadtree. However, the implementation would require quite a few special cases, thereby complicating the process. Transitions between different faces of the polyhedron would also be more complex.

Our neighbor-finding technique has a natural application [Junkins 1999] in performing subdivision surface computations over triangular meshes. Subdivision surface algorithms are popular computer graphics algorithms for generating visually rich and smooth surfaces from a coarse base mesh of control polygons. They provide a powerful alternative to more traditional polygon or NURBS modeling, and enable developers to create scalable 3D applications that boast multiresolution surface capability [Dyn et al. 1990; Hoppe et al. 1994]. Our triangle encoding method alleviates the need to maintain an explicit, pointer-based triangle quadtree while enabling worst-case constant time neighbor-finding. Furthermore, our neighbor-finding methods are extremely cache and register friendly, thereby lending themselves well to highly optimized implementations on widely available consumer hardware [Junkins 1999].

Besides being useful in finite element analysis, as well as applications involving the modeling of spherical data, our algorithms could also be used in a ray tracer where a surface is represented by a quadtree hierarchy composed of triangle meshes instead of square meshes. The algorithms are also useful in the implementation of an interpolation process between neighboring triangles to give a smoother appearance [Fekete 1990]. This would remove the "cracks" that sometimes arise when storing elevation data in the current system. A direction for future research is the extension of our methods to three-dimensional data, where the basic shapes are now tetrahedra instead of triangles (e.g., Perucchio et al. [1989]; Saxena and Perucchio [1989]). Our current methods are applicable to the surface of the three-dimensional data only.

REFERENCES

ABEL, D. J. 1984. A B+ tree structure for large quadtrees. *Comput. Vision Graph. Image Process. 27*, 1, 19–31.

BANK, R. E., SHERMAN, A. H., AND WEISER, A. 1983. Refinement algorithms and data structures for regular local mesh refinement. In *Scientific Computing* IMACS Transactions on Scientific Computation, vol. 1. North-Holland Publishing Co., Amsterdam, The Netherlands, 3–17.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept.), 509–517.

BERN, M., EPPSTEIN, D., AND GILBERT, J. 1994. Provably good mesh generation. *J. Comput. Syst. Sci. 48*, 3 (June 1994), 384–409.

COMER, D. 1979. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June), 121–137.

DA SILVA, A. AND DUARTE-RAMOS, H. 1990. A progressive trimming approach to space decomposition. In *Proceedings of the Fourth International Symposium on Spatial Data Handling* (Zurich, Switzerland, July 1990),

DE FLORIANI, L., MAGILLO, P., AND PUPPO, E. 1997. Building and traversing a surface at variable resolution. In *Proceedings of the conference on Visualization '97* (Phoenix, AZ, Oct. 19–24, 1997), R. Yagel, H. Hagen, R. Moorhead, N. Johnston, T.-M. Rhyne, and C. Hansen, Eds. ACM Press, New York, NY, 103ff.

DE FLORIANI, L., MARZANO, P., AND PUPPO, E. 1996. Multiresolution models for topographic surface description. *Visual Comput. 12*, 7 (Aug.), 317–345.

DUTTON, G. 1984. Geodesic modelling of planetary relief. *Cartographica 21*, 2-3, 188–207.

DUTTON, G. 1990. Locational properties of quaternary triangular meshes. In *Proceedings of the Fourth International Symposium on Spatial Data Handling* (Zurich, Switzerland, July 1990), 901–910.

DYN, N., LEVINE, D., AND GREGORY, J. A. 1990. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Trans. Graph. 9*, 2 (Apr. 1990), 160–169.

ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. 1995. Multiresolution analysis of arbitrary meshes. In *Proceedings of the 22nd Annual ACM Conference on Computer Graphics* (SIGGRAPH '95, Los Angeles, CA, Aug. 9–11), S. G. Mair and R. Cook, Eds. Annual conference series ACM Press, New York, NY, 173–182.

EPPSTEIN, D. 1992. Approximating the minimum weight triangulation. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA '92, Orlando, FL, Jan. 27–29), G. Frederickson, Ed. ACM Press, New York, NY, 48–57.

FEKETE, G. 1990. Rendering and managing spherical data with sphere quadtrees. In *Proceedings of the Conference on Visualization '90* (San Francisco, CA, Oct. 1990),

FEKETE, G. AND DAVIS, L. S. 1984. A new representation for 3-d object recognition. In *Proceedings of the Workshop on Computer Vision: Representation and Control* (Annapolis, MD, Apr. 1984), 192–201.

FUCHS, H., KEDEM, Z. M., AND NAYLOR, B. F. 1980. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph. 14*, 3 (July), 124–133.

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. ARTS: Accelerated ray-tracing system. *IEEE Comput. Graph. Appl. 6*, 4 (Apr. 1986), 16–26.

GARGANTINI, I. 1982. An effective way to represent quadtrees. *Commun. ACM 25*, 12 (Dec.), 905–910.

GLASSNER, A. S. 1984. Space subdivision for fast ray tracing. *IEEE Comput. Graph. Appl. 4*, 10, 15–22.

GOODCHILD, M. F. AND SHIREN, Y. 1992. A hierarchical spatial data structure for global geographic information systems. *CVGIP: Graph. Models Image Process. 54*, 1 (Jan. 1992), 31–44.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (SIGGRAPH '97, Los Angeles, CA, Aug. 3–8), G. S. Owen, T. Whitted, and B. Mones-Hattal, Eds. ACM Press/Addison-Wesley Publ. Co., New York, NY, 189–198.

HOPPE, H., DEROSE, T., DUCHAMP, T., HALSTEAD, M., JIN, H., MCDONALD, J., SCHWEITZER, J., AND STUETZLE, W. 1994. Piecewise smooth surface reconstruction. In *Proceedings of the ACM Conference on Computer Graphics* (SIGGRAPH '94, Orlando, FL, July 24–29, 1994), D. Schweitzer, A. Glassner, and M. Keeler, Eds. ACM Press, New York, NY, 295–302.

HUNTER, G. M. 1978. Efficient computation and data structures for graphics. Ph.D. Dissertation. Department of Computer Science, Princeton Univ., Princeton, NJ.

JUNKINS, S. 1999. Constant time neighbor finding for subdivision surfaces. Tech. Rep.. Intel Corp., Santa Clara, CA.

KAPLAN, M. R. 1985. Space-tracing: A constant time ray-tracer. In *Proceedings of the ACM Conference on Computer Graphics* (SIGGRAPH '85, San Francisco, CA), ACM, New York, NY.

KELA, A., PERUCCHIO, R., AND VOELCKER, H. 1986. Toward automatic finite element analysis. *Comput. Mech. Eng. 5*, 1 (July), 57–71.

KLINGER, A. 1971. Patterns and search statistics. In *Optimizing Methods in Statistics*, J. S. Rustagi, Ed. Academic Press, Inc., New York, NY, 303–337.

KNOWLTON, K. 1980. Progressive transmission of grey-scale and binary pictures by simple efficient, and lossless encoding schemes. *Proc. IEEE 68*, 7 (July), 885–896.

LEE, M. AND SAMET, H. 1998. Navigating through triangle meshes implemented as linear quadtrees. TR-3900. Department of Computer Science, University of Maryland, College Park, MD.

MORTON, G. M. 1966. A computer oriented geodetic data base and a new technique in file sequencing. Tech. Rep.. IBM, Ltd., Ottawa, Canada.

OTOO, E. J. AND ZHU, H. 1993. Indexing on spherical surfaces using semi-quadcodes. In *Proceedings of the Third International Symposium on Advances in Spatial Databases* (SSD'93, Singapore, June 1993), D. Abel and B. C. Ooi, Eds. Springer-Verlag, New York, 510–529.

PERUCCHIO, R., SAXENA, M., AND KELA, A. 1989. Automatic mesh generation from solid models based on recursive spatial decompositions. *Int. J. Num. Methods Eng. 28*, 11 (Nov.), 2469–2501.

SAMET, H. 1982. Neighbor finding techniques for images represented by quadtrees. *Comput. Vision Graph. Image Process. 18*, 1 (Jan.), 37–57.

SAMET, H. 1989. Implementing ray tracing with octrees and neighbor finding. *Comput. Graph. 13*, 4, 445–460.

SAMET, H. 1990a. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Series in Computer Science. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

SAMET, H. 1990b. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley Series in Computer Science. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

SAMET, H. AND TAMMINEN, M. 1988. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Trans. Pattern Anal. Mach. Intell. 10*, 4 (July 1988), 579–586.

SAXENA, M. AND PERUCCHIO, R. 1989. Element extraction for automatic meshing based on recursive spatial decompositions. Tech. Rep.. University of Rochester, Rochester, NY.

SCHRACK, G. 1991. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Underst. 55*, 3 (May 1992), 221–230.

SNYDER, J. P. 1987. Map Projection; A Working Manual. U.S. Government Printing Office, Washington, DC.

TAMMINEN, M. 1984. Comment on quad- and oct-trees. *Commun. ACM 27*, 3 (Mar.), 248–249.

TAMMINEN, M., KARONEN, O., AND MANTYLA, M. 1984. Ray-casting and block model conversion using a spatial index. *Comput. Aided Des. 16*, 4 (July 1984), 203–208.

TOBLER, W. AND CHEN, Z. T. 1986. A quadtree for global information storage. *Geographical Anal. 18*, 4 (Oct.), 360–371.

YERRY, M. A. AND SHEPHARD, M. S. 1983. A modified quadtree approach to finite element mesh generation. *IEEE Comput. Graph. Appl. 3*, 1 (Feb.), 39–46.

Received: July 1998; revised: September 1999; accepted: October 1999