Constant-Time Neighbor Finding in Hierarchical Tetrahedral Meshes*

Michael Lee†
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
magus@cs.umd.edu

Leila De Floriani‡
Dipartimento di Informatica
e Scienze dell'Informazione
Universita' di Genova
Via Dodecaneso, 35
16146 Genova (Italy)
deflo@disi.unige.it

Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
hjs@cs.umd.edu

Abstract

Techniques are presented for moving between adjacent tetrahedra in a tetrahedral mesh. The tetrahedra result from a recursive decomposition of a cube into six initial congruent tetrahedra. A new technique is presented for labeling the triangular faces. The labeling enables the implementation of a binary-like decomposition of each tetrahedron which is represented using a pointerless representation. Outlines of algorithms are given for traversing adjacent triangular faces of equal size in constant time.

1. Introduction

Several applications including scientific visualization, medical imaging, and finite element analysis, deal with increasingly large sets of three dimensional data describing scalar fields. A volume data set consists of a set of points in the three-dimensional space with a value for some function. Such data is often modeled by a mesh consisting of tetrahedral volume elements. The mesh can be regular or unstructured depending on the distribution of the points in the data set.

In order to handle volume data sets of large size and to accelerate rendering, multiresolution models have been proposed which allow approximating the mesh connecting the original data points with a collection of simplified meshes at different levels of detail [2, 3, 14, 22]. The resolution (i.e., the density of the cells) of an approximating mesh may vary in different parts of the field domain (e.g., inside a box, or along a cutting plane) or in the proximity of interesting field values.

In particular, when the data is regularly-distributed (i.e., from a grid), hierarchical tetrahedral meshes generated by recursive bisection are used [8, 9, 14, 20, 22]. We use the term *hierarchical regular tetrahedral mesh* to describe such meshes. Such meshes have been introduced for domain decomposition in finite element analysis [10, 13]. The basic element of decomposition is a tetrahedron which is bisected along its longest edge to generate the next level in the hierarchy. The hierarchy results from the application of a regular decomposition process to a cube initially split into six tetrahedra. This process is continued while obtaining smaller and smaller tetrahedra each of which is of one of three basic shapes so that all of the tetrahedra are elements of three similar types.

The techniques presented in this paper expand upon this basic decomposition strategy by ordering the tetrahedra is such a way that it becomes possible to find not just the children and the parent of a given tetrahedron, but also neighboring tetrahedra using simple arithmetic and bitwise operations. This allows us to move between adjacent tetrahedra (and any corresponding data) in constant time. The result is that we have more flexibility in extracting selectively refined meshes from the hierarchical structure and moving along adjacent tetrahedra, as in algorithms for isosurface extraction or in direct volume rendering techniques. Our work is related to our previous work on traversing triangle elements of a triangular mesh where each triangle is recursively decomposed into four equilateral triangles [11] where we also obtained constant time neighbor finding algorithms.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we present our method of labeling the tetrahedral elements within a mesh, which is new. Section 4 shows how to find a neighbor of equal size within one of the tetrahedra of the cube in time

* The support of the National Science Foundation under Grants EIA-99-00268 and IRI-97-12715 is gratefully acknowledged.

† This work was supported in part by Columbia Union College, Takoma Park, Maryland.

‡ The work was performed in part while Leila De Floriani was visiting the Center for Automation Research and the Institute for Advanced Computer Studies at the University of Maryland.

0-7695-0853-7/01 $10.00 © 2001 IEEE

proportional to the maximum level of resolution while Section 5 describes how to find the neighbors in constant time. Concluding remarks are drawn in Section 6.

2. Related work

Finding neighboring elements in a recursive decomposition of space is a subject of a considerable amount of research in recent years. The earliest results dealt with algorithms to find adjacent blocks (i.e., neighbors) of greater than or equal size in region quadtrees [16] and region octrees [17]. In this case, the decomposition was into congruent hyper-rectangle blocks. These algorithms made use of a pointer structure of the tree and thus had execution time proportional to the resolution of the underlying image. Subsequently, methods for representing such structures that did not require pointers were developed. In this case, each block in the decomposition is represented by a unique pair of numbers known as its *location code* where the first number indicates the depth of the tree at which i is found and the second number indicates the path from the root of the tree to i (e.g., [7]). The path consists of the concatenation of the two-bit (three-bit for three dimensions) numbers corresponding to the child types of each node that is traversed on the path from the root of the tree to i. We refer to the path as the *path array component of the location code*. Pointer-less representations enable finding neighbors of equal size in constant time as they consist of bit manipulations such as arithmetic and logical operations. These results were extended by Lee and Samet [11] to also apply to triangular blocks that are made up of four equilateral triangles.

Related work has also been done by Evans et al. [5] who use a hierarchy of right triangles based on a regular decompostion to decompose the domain of a two-dimensional scalar fields. Hierarchies of right triangles have been extensively used for multiresolution terrain modeling [4, 12, 15]. Coordinates are not explicitly stored since they can be calculated from the label (or location code) of the triangle. Since a pointer-based binary tree structure would be inefficient in its use of space, Evans et al. use an array where the label of a node determines the node location in the array. Algorithms for finding neighbors in time proportional to the length of the location code (i.e., proportional to the depth of the triangle) as well as in constant time by using a relatively small number of arithmetic and bitwise logical operations. Our work is related to that of Evans et al. in that we start with a unit cube (just as they do with a unit square).

Zhou et al. [22] present a multiresolution tetrahedral framework to manage regular volume data, based on a hierarchical regular tetrahedral mesh represented as a binary tree, stored in an array. Using this strategy, the addresses of children and parents are easy to obtain with simple calculations which are the same for any binary tree. A similar data structure is used by Gerstner and Rumpf [8] for extracting isosurfaces at different levels of detail.

Hebert [10] introduces the idea of using symbolic algorithms to find parents, children, and neighbors in a nested tetrahedral mesh. Operations are done within symbolic tetrahedral codes which contain a path to the lattice origin of the tetrahedron and a triple (permutation number, rotation number, and descendent number) identifying the tetrahedron relative to the lattice origin. These lattice origins effectively indicate which cubes (or sub-cubes) contain a given tetrahedron. In particular, the center of each cube is used to represent the cube and acts as the reference point for locating the tetrahedra. Using this technique, three of the four tetrahedra sharing a face will share the same lattice origin and will require only a table lookup to get the symbolic code of the appropriate neighbor. For the fourth tetrahedron along the remaining face, the path to the lattice origin must be updated to get the complete symbolic code of the neighbor.

3. Labeling the tetrahedral decomposition

In this section, we review the recursive tetrahedral decomposition rule [22] that forms the basis of our regular tetrahedral data structure, and introduce rules for labeling the children of a tetrahedron and the vertices of these children. We consider a subdivision of a cube which contains six tetrahedra, all adjacent along the main diagonal of the cube (see Figure 1a). We are using a linear representation for the hierarchical tetrahedral meshes instead of a pointer based tree structure. Thus our encoding will be a sequence of locational codes, one for each tetrahedron. As these location codes determine a tetrahedron, the terms location code and tetrahedron code will be used interchangeably.

For simplicity, we try to order the vertices of a tetrahedron such that its longest edge is v_3v_4. This avoids calculating edge lengths and makes the process of finding the midpoint v_m easier. Since the longest edge in the cube is the diagonal, we label the diagonal v_4v_3, where v_4 is the vertex of the cube at the origin of the coordinate system. The remaining vertices must all be either v_1 or v_2, and no tetrahedron can contain more than one vertex of type v_1 and one vertex of type v_2. Thus we use the label v_1 for the three vertices which are closest to vertex v_4 and label v_2 for the three vertices which are closest to vertex v_3. If we consider the main diagonal as a directed edge from vertex v_4 to vertex v_3, then we can label the tetrahedra themselves (effectively the children of the cube) in counter-clockwise order so that the first tetrahedron is labeled 0 and the last is labeled 5 (see Figure 1a).

Each tetrahedron t (shown in Figure 1a) is recursively subdivided into two tetrahedra by bisecting the longest edge (v_3v_4 in Figure 2a). As we will see below, the shape of each

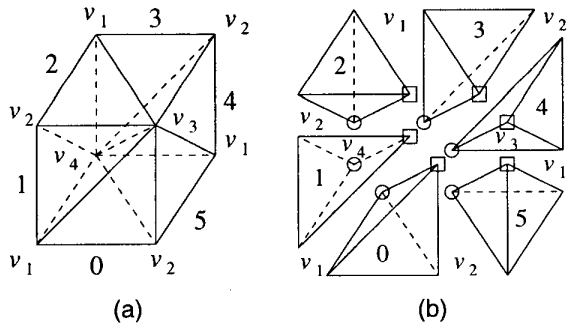

Figure 1. Labeling of cube.

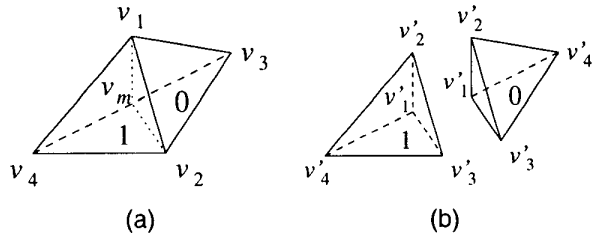

Figure 3. (a) Labeling of 1/2 pyramid, and (b) the resulting pair of 1/4 pyramids.

of the six initial tetrahedra is analogous to the result of splitting a pyramid three times in succession thereby creating what we term a *1/8 pyramid.*

Splitting the 1/8 pyramid along the longest edge results in two tetrahedra which which have shape identical to that obtained by splitting a pyramid with a square base in half along the diagonal of its base. Therefore, we call the resulting shape a *1/2 pyramid* (see Figure 2b). If $t = [v_1, v_2, v_3, v_4]$, then the two resulting 1/2 pyramids are $t_0 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_4]$ and $t_1 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_2, v_1, v_3]$, where v_m is the midpoint of edge v_3v_4.

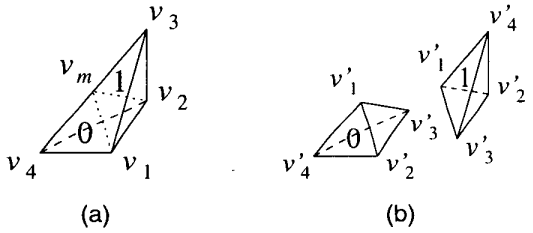

Figure 2. (a) Labeling of 1/8 pyramid, and (b) the resulting pair of 1/2 pyramids.

All possible orientations (including rotation and reflection) of the 1/2 pyramid generated by the subdivision of the 1/8 pyramid can be summarized in one basic configuration which is shown in Figure 3a. Splitting the 1/2 pyramid along the longest edge results in two tetrahedra which have shape identical to that obtained by splitting a 1/2 pyramid into two halves. Therefore, we call the resulting shape a *1/4 pyramid* (see Figure 3b). If $t = [v_1, v_2, v_3, v_4]$, then the two resulting 1/4 pyramids are $t_0 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_3]$ and $t_1 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_4]$, where v_m is the midpoint of edge v_3v_4.

All possible orientations (including rotation and reflection) of the 1/4 pyramid generated by the subdivision of the 1/2 pyramid can be summarized in one basic configuration which is shown in Figure 4a. Splitting the 1/4 pyramid along the longest edge results in two tetrahedra which have shape identical to that obtained by splitting a 1/4 pyramid into two halves, and the resulting shape is a *1/8 pyramid* (see Figure 4b).If $t = [v_1, v_2, v_3, v_4]$ and the parent was child 0, then the two resulting 1/8 pyramids are $t_0 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_3]$ and $t_1 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_4]$, where v_m is the midpoint of edge v_3v_4. On the other hand, if $t = [v_1, v_2, v_3, v_4]$ and the parent was child 1, then we swap the labels of the children so that $t_0 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_4]$ and $t_1 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_3]$.

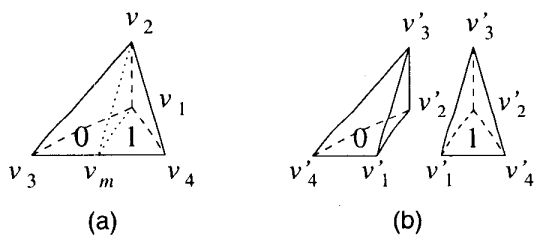

Figure 4. (a) Labeling of 1/4 pyramid, and (b) the resulting pair of 1/8 pyramids.

Notice that after the above three decomposition steps we ended up with the same shape with which we started the decomposition process. This is why used the term 1/8 pyramid to describe the tetrahedra that resulted from the initial decomposition of the unit cube. In fact, if we continue to subdivide, we will always continue to generate the same three shapes. In particular, we will always cycle through the 1/2, 1/4, and 1/8 pyramids. Note that using our labeling scheme for vertices and children, it is quite simple to traverse the tree using only local computations to determine where we are in space. The only computation is finding the midpoint v_m of the longest edge. All other vertices can be obtained directly from the vertices of the parent tetrahedron.

4. Neighbor finding

In this section we describe how to find an equal-sized face neighbor of a tetrahedron. The algorithm uses an approach analogous to that defined in [18]. We will not make use of the actual coordinate values of the tetrahedron corresponding to a given location code. Instead, only the location code itself will be processed. Elements of the path array will be referenced using array notation.

We identify four neighbor directions. They are based on the four faces of an arbitrary tetrahedron denoted by $t = [v_1, v_2, v_3, v_4]$. A neighbor of type 1 is the tetrahedron which shares face $v_1v_2v_3$. A neighbor of type 2 is the tetrahedron which shares face $v_1v_2v_4$. A neighbor of type 3 is the tetrahedron which shares face $v_1v_3v_4$. A neighbor of type 4 is the tetrahedron which shares face $v_2v_3v_4$. It should be clear that repeated application of a given neighbor type will continuously switch between the two neighbors which share the specified face. The neighbor finding process consists of two steps. The first identifies the nearest common ancestor of the tetrahedron t and its neighbor t' of type i. The second updates the location code for the neighbor using the information obtained while finding the nearest common ancestor.

4.1. Locating the nearest common ancestor

For a given neighbor direction i (which determines the face we must cross to get the neighbor), we simply scan the location code from right to left until the neighbor direction forces us to cross face $v_1v_2v_3$ of the ancestor. This works because face $v_1v_2v_3$ is shared by siblings and the parent of these sibling ancestors is also the nearest common ancestor of the input tetrahedron t and its neighbor t'.

As an example, consider the location code `210011`. Since the depth is five, this location code refers to a 1/4 pyramid. If we want to find a neighbor type 3 (face $v_1v_3v_4$), then we must first find the nearest common ancestor using the right to left scan which was just described. As our neighbor direction forces us to cross face $v_1v_3v_4$, we must look at the parent (`21001`). Keeping the same neighbor direction means that we must now cross face $v_2v_3v_4$ of the parent. Again, we must look at the next ancestor (`2100`). Keeping the same neighbor direction means that we must now cross face $v_1v_2v_3$ of the ancestor. Crossing face $v_1v_2v_3$ is our stopping condition, so we stop at `2100`. Officially, the nearest common ancestor (`210`) is one level up (see Figure 5), but we need to know which child contained our input tetrahedron in order to get the appropriate sibling for the neighbor.

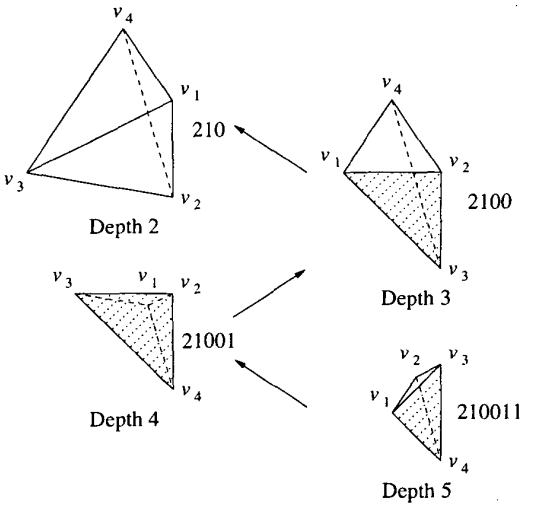

Figure 5. Finding the nearest common ancestor of 210011.

4.2. Updating the location code

In this step we just invert the one bit corresponding to the child of the nearest common ancestor. This process works regardless of the original neighbor type which we were trying to find. No further work is necessary, as all neighbors location codes differ just by this one bit. This process is relatively similar to the location codes used by Fekete [6] for two-dimensional triangular meshes except that our tetrahedron codes are three-dimensional.

If we continue with our example from Section 4.1, then we know that the nearest common ancestor is `210`. Since `2100` has a sibling in the desired neighbor direction, we just invert the last bit to point to the new sibling. In this example, the sibling of `2100` is `2101`, so the neighbor of `210011` which shares face $v_1v_3v_4$ is `210111` (see Figure 6).

4.3. Extensions to the entire cube

Since we actually have six tetrahedra at the first decomposition level of the cube, we need to make sure that our transitions work between these six top level tetrahedra. The order of the vertices for these six tetrahedra have been originally selected so that they imitate the ordering or layout of tetrahedra at lower levels of the decomposition. Note that, the labeling of these top six tetrahedra themselves is not critical since we can always use table lookup to find the top-level neighbors.

In terms of neighbor finding, the first change is that we must stop whenever we encounter the top of our location code. If the neighbor direction forces us to exit the cube,

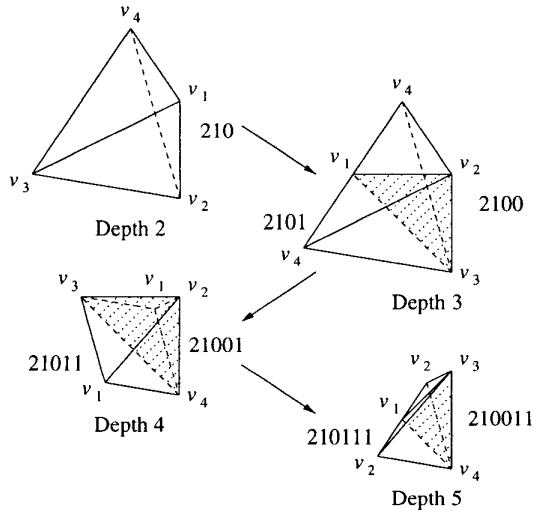

Figure 6. Updating the location code when finding the neighbor of type 3 of 210011.

then we need to return an error. Otherwise, we know that a neighbor must exist, so we consider the entire cube as the nearest common ancestor for the two neighbors.

When we encounter the top level, finding the neighbor is no longer simply a matter of inverting one bit. However, the process is still quite simple. We only need to pick a new top level tetrahedron, since the rest of the path will be identical for both neighbors. This property is similar to the fact that two neighbors within one top level tetrahedron differ by only one bit. Therefore, we simply select the new top level bits based on a table lookup.

As an example, we again consider the tetrahedron of location code `210011`, and we try to find its neighbor of type 4 (sharing face $v_2v_3v_4$). Since our neighbor direction forces us to cross face $v_2v_3v_4$, we must look at the parent (`21001`). Keeping the same neighbor direction means that we must now cross face $v_1v_2v_4$ of the parent. Again, we must look at the next ancestor (`2100`). Keeping the same neighbor direction means that we must now cross face $v_2v_3v_4$ of the ancestor. Again, we must look at the next ancestor (`210`). Keeping the same neighbor direction means that we must now cross face $v_1v_2v_4$ of this ancestor, so we must look at the next ancestor. Keeping the same neighbor direction for the next ancestor (`21`) means that we must now cross face $v_1v_3v_4$ to find the neighbor. Again, we must look at the next ancestor (`2`). Keeping the same neighbor direction means that we must now cross face $v_1v_3v_4$ of this ancestor. We cannot find the parent of this tetrahedron because we are at the top level, so we use a table lookup to determine that the appropriate neighbor is `3`. Therefore, the neighbor of `210011` which shares face $v_2v_3v_4$ is `310011` (see Figure 7).

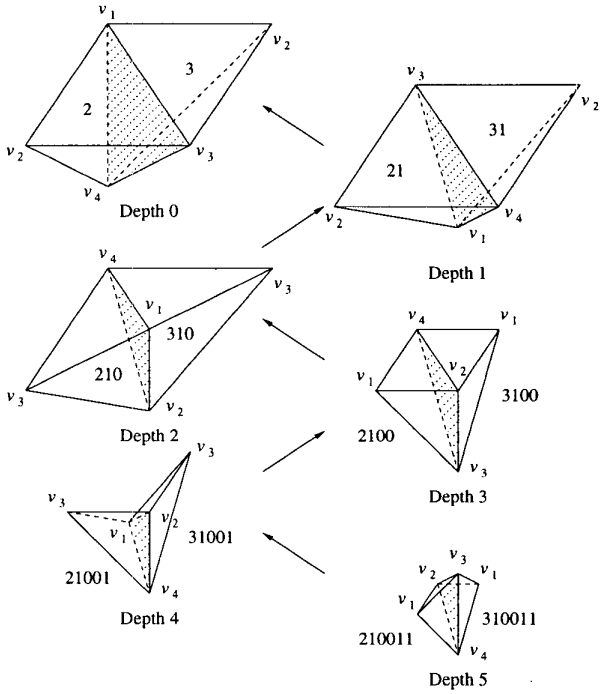

Figure 7. Example illustrating finding neighbor type 4 of 210011.

5. Constant-time neighbor finding algorithm

In this section we describe how to perform neighbor finding in worst-case constant time. The algorithms presented here make use of the carry property of addition to quickly find a neighbor without specifically searching for a nearest common ancestor. We replace the iteration which was part of the right to left scan in the previous neighbor finding algorithm by an arithmetic operation that takes constant time instead of as much as the depth of the tree. The algorithms make use of just a few bit manipulation operations which 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 is assumed to fit in one computer word. If more than one word is needed, then the algorithms are a bit slower but still take constant time.

We use an identification technique similar to the one used in navigating between 2D triangle meshes (see [11]). In the 2D case, we used bit masks to identify certain bit patterns within the location codes. We called them idmasks. These bit masks help us to identify which nodes contain (or fail to

contain) a sibling in the appropriate neighbor direction, and therefore which positions in the location code should propagate the carry. Assuming that we generate our bit masks correctly, finding the location of the nearest common ancestor is as simple as a single addition. We use the highest carry position after the addition to determine which bit gets inverted in order to find the neighbor.

Determining which positions should propagate the carry is not always an easy task. The first thing we need to consider is what bit patterns indicate a carry based on the neighbor direction which we are given. To simplify our tables and algorithms, we only consider the recurrence relations for a 1/8 pyramid. It requires a maximum of two steps (or changes in level) in order to ensure that we are working within a location code of a 1/8 pyramid.

5.1. Neighbor type 1

Neighbor type 1 always goes straight to the sibling (the nearest common ancestor is the parent), so not much work is required. In fact, finding the sibling is simply a matter of inverting the last bit (based on the level or depth in the hierarchy) in the location code.

5.2. Neighbor type 2

Face $v_1v_2v_4$, corresponding to neighbor type 2, is always contained by either face $v_1v_2v_3$ or face $v_1v_2v_4$ of the third ancestor. This is a direct result of the splitting rules given in Section 3. If face $v_1v_2v_4$ in the child is contained in face $v_1v_2v_3$ of the third ancestor, then we know the neighbor, because face $v_1v_2v_3$ in the third ancestor is shared by the ancestor and its sibling. However, if face $v_1v_2v_4$ in the child is contained in face $v_1v_2v_4$ of the third ancestor, then the identity of the neighbor isn't immediately obvious. We must continue searching for the neighbor through face $v_1v_2v_4$, corresponding to neighbor type 2, for the third ancestor. This process continues until we can determine the sibling of an ancestor and we know that we can find the sibling when we are on face $v_1v_2v_3$ on the ancestor.

Since our goal is to find the appropriate neighbor in constant time, we want to use simple addition and take advantage of any carries. In particular, we want a carry to occur whenever we need to continue searching (looking at the ancestor) in the hierarchy. Finding neighbor type 2 requires finding either neighbor type 1 or 2 of the third ancestor (i.e., having the same shape, double the edge size, and eight times the volume) depending on which child of the third ancestor was needed to reach the input location code. This means that we need a carry if the child of the third ancestor was a child of type 0, and no carry if it was a child of type 1.

The algorithm locates the positions within the location code where the sibling can be determined because the relevant face is face $v_1v_2v_3$. Let a denote these positions. Since we want carries where the sibling cannot be determined, we need to complement a and then perform the addition. To isolate the one bit that needs to be inverted in the location code, we perform a logical AND on the result of the addition with the value originally in a (i.e., the original uncomplemented idmask) which denotes the positions where we can determine the sibling. Let b denote the result. The bits are offset by one position at this point, so we shift b to the left by one bit position. Finally, we simply invert the appropriate bit in the location code, by using a logical XOR operation between the location code and the current bit mask b (which contains only one bit marking the position where we found the sibling).

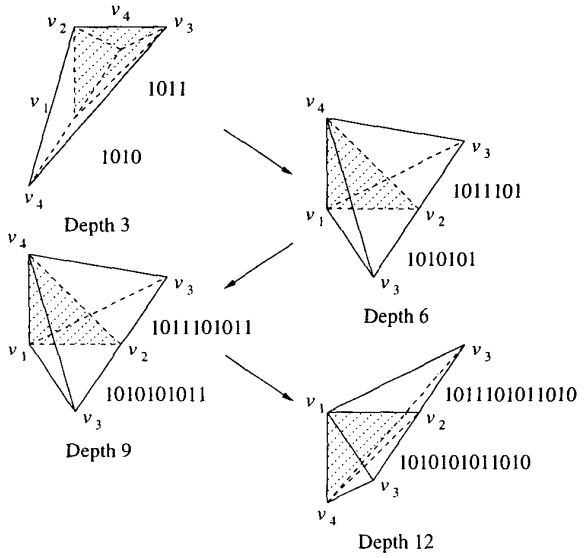

Figure 8. Example illustrating finding neighbor type 2 of 1010101011010.

As an example, let us consider location code `1010101011010` (see Figure 8). We can determine the sibling at any position where the first bit (out of 3) is `1`. The mask marking these positions is `0000100000000`, so this is stored in a. The complement of bit pattern a is `1111011111111`. Adding one to this value gives us `1111100000000`. We isolate the one bit that needs to be inverted using the logical AND. This gives us `0000100000000`. We need to shift left, so we get `0001000000000`. Finally, we use the logical XOR to invert the appropriate bit. The input location code `1010101011010` XOR `0001000000000` gives us our final answer of `1011101011010`.

5.3. Neighbor type 3

Face $v_1v_3v_4$, corresponding to neighbor type 3, is generally contained by either face $v_1v_3v_4$ or face $v_2v_3v_4$ of the third ancestor. If it is contained by any other face, then the neighbor can be determined without examining the third ancestor. This is a direct result of the splitting rules given in Section 3. Whenever the neighbor cannot be determined because the nearest common ancestor is beyond the third ancestor (this will occur if face $v_1v_3v_4$ in the child is contained in face $v_1v_3v_4$ or face $v_2v_3v_4$ of the third ancestor), then we must continue searching for the neighbor through the appropriate face of the third ancestor. Notice that if this is face $v_1v_3v_4$ of the third ancestor, then we continue to search for the same neighbor type (i.e., 3). Otherwise, if it is face $v_2v_3v_4$ of the third ancestor, then we must change our strategy a bit, effectively finding neighbor type 4 of the third ancestor.

Since our goal is to find the appropriate neighbor in constant time, we want to use simple addition and take advantage of any carries. We want a carry to occur whenever we need to continue searching higher in the hierarchy. Finding neighbor type 3 either terminates at the second ancestor or requires finding neighbor type 3 or 4 of the third ancestor depending on the bits corresponding to the first and second ancestors of the node. Basically, there should be no carry if the first ancestor was child 0 of the second ancestor. Otherwise, we want a carry and the neighbor type will depend on the child type of the second ancestor.

Since a carry occurs whenever we search higher than the third ancestor, and we might need to find either neighbor type 3 or 4 of the third ancestor, we need an indicator to keep track of which neighbor type we want to find at each level (or at least at every third level). We use a "neighbor mask" to store this information.

The first step in finding neighbor type 3, is identifying the positions within the location code where the sibling can be determined because the relevant face is face $v_1v_2v_3$. These positions will be recorded in a mask a whose construction is described below. Since neighbors are determined before we reach the third ancestor (otherwise, we continue upwards in the hierarchy), we will examine the bits in sets of three, where the leftmost (or most significant) bit is called bit 1, the next (or middle) bit is called bit 2, and the rightmost (or least significant) bit is called bit 3.

If bit 2 is 0 and we are looking for neighbor type 3 at this level (this is determined by examining the neighbor mask), then we can identify the neighbor. If bits 1 and 3 are the same and we are looking for neighbor type 4 at this level, then we can identify the neighbor. The mask a is constructed based on these patterns. Next, we complement a prior to the addition because we want a carry to occur whenever we cannot identify the neighbor at a given level. The carry is propagated until we reach the bit corresponding to the level at which this neighbor can be identified. To isolate the one bit that needs to be inverted in the location code, we perform a logical AND on the result of the addition with the value in a (i.e., the original uncomplemented idmask) which denotes the positions where we can determine the sibling. Let b denote the result. Depending on which neighbor type we are finding at this point (again, determined by examining the neighbor mask), the bits might be offset by one position. If so, we shift b to the right by one bit position. Finally, we simply invert the appropriate bit in the location code, by using a logical XOR operation between the location code and the current bit mask b (which contains only one bit marking the position where we found the sibling).

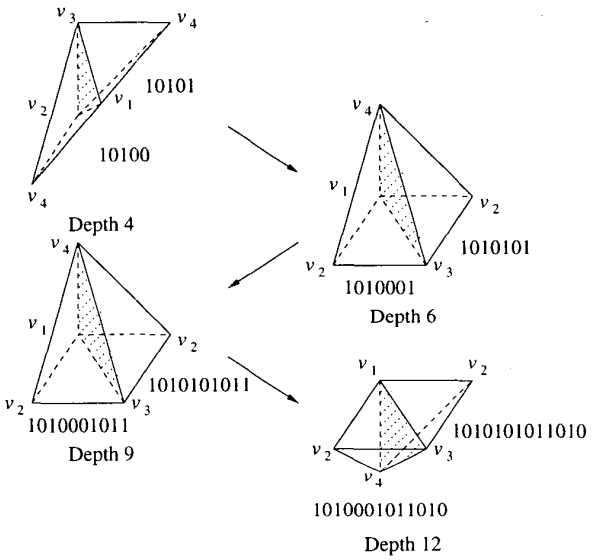

Figure 9. Example illustrating finding neighbor type 3 of 1010101011010.

As an example, let us consider location code `1010101011010` (see Figure 9). We can determine the sibling if bit 2 is `0` (for neighbor type 3), or if bits 1 and 3 are the same (for neighbor type 4). This is not true for the rightmost triple (`010`, starting with neighbor type 3), not true for the adjacent triple (`011`, still using neighbor type 3), true for the next adjacent triple (`101`, still using neighbor type 3), and true for the leftmost triple (`010`, with neighbor type 4 since the previous triple caused the neighbor type to change). Therefore, the mask a which marks the true positions is `0100100000000`. The complement of a is `1011011111111`. Adding one to this value gives us `1011100000000`. We isolate the one significant bit using the logical AND. This gives us `0000100000000`. Finally, we use the logical XOR

to invert the appropriate bit. The input location code 1010101011010 XOR 0000100000000 gives us our final answer of 1010001011010.

5.4. Neighbor type 4

Finding a neighbor of type 4 is quite similar to the technique used for neighbor type 3. Again, we use simple addition and take advantage of the carries. Finding neighbor type 4 either terminates at the first ancestor or requires finding neighbor type 3 or 4 of the third ancestor depending on the bits corresponding to the current node and its second ancestor. Basically, there should be no carry if the second ancestor was child 0 of the third ancestor and the current node is child 0, or if the second ancestor was child 1 of the third ancestor and the current node is child 1. Otherwise, we want a carry and the neighbor type will depend on the child type of the second ancestor.

5.5. The neighbor mask

When seeking neighbor type 3 (4) and finding that the child of the third ancestor is child 1, we switch to the method of finding neighbor type 4 (3). This causes somewhat of a problem since we need to know the neighbor type for which we are searching along the entire location code simultaneously. If we drop back to an iterative approach, then we lose our constant time behavior. Thus, we introduce a neighbor mask which stores the state of our neighbor switching. This allows us to make neighbor type 3 and 4 transitions in constant time.

Of course, we need to be able to update or maintain our neighbor mask in constant time too. Note that the neighbor mask only changes when the bit corresponding to a 1/2 pyramid changes. When such a bit changes, we need to make sure that all bits in our neighbor mask which occur before the given bit are changed also. This is easily accomplished in constant time using just a few bit operations not given here.

5.6. Transitions across the six top level tetrahedra

Transitions between the six top level tetrahedra are relatively simple. This situation arises if the addition from our constant time algorithm generates a carry past the leftmost end of the input location code. This is done in the same way as described for the traditional neighbor finding method that does not run in constant time (see Section 4.3).

6. Concluding remarks

We have described a constant time technique for navigating between adjacent tetrahedra in a hierarchical regular tetrahedral mesh. Fast navigation is useful for many applications. One such application is in the generation of a selectively refined mesh from a multiresolution tetrahedral model of a scalar field. In particular, we want to extract a consistent (i.e., with no "cracks" which means that adjacent faces are equal) tetrahedral mesh for a specified error tolerance. While we have only described the technique used for equal-sized neighbors, it is easy to extend the technique in order to handle the variable resolution mesh (with many different tetrahedron sizes) which is generated during such error based extractions. In this case, our neighbor finding methods can be used to speed up the process of ensuring that all neighboring faces are of the same size especially when they do not result from a split of the same larger ancestor tetrahedron. Figure 10 is an example of such a mesh for a bucky-ball data set. Moreover we can obtain a description of the resulting mesh which also contains adjacencies among tetrahedra. Adjacencies are necessary for depth sorting and composition of tetrahedral cells involved in direct volume rendering techniques [19, 21]. Octree data structures and algorithms have many similar benefits, but do not grant the same level of flexibility as true tetrahedral decompositions which are necessary in order to generate consistent meshes. Both octrees and the techniques presented here assume a cube is used as the basis for further subdivision. However, unstruc-

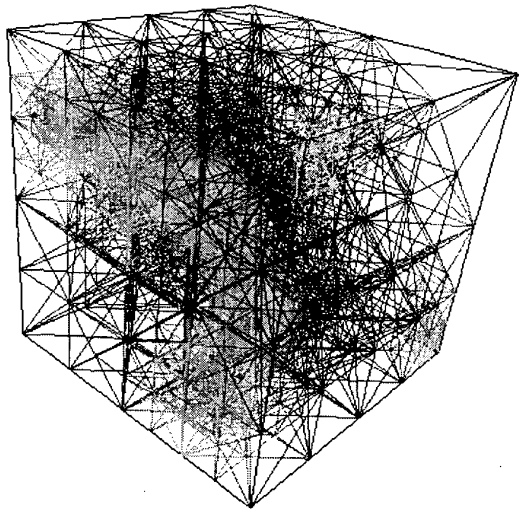

Figure 10. Example of a consistent mesh with 1365 vertices and 6556 tetrahedra generated using an error tolerance of 14% from a bucky-ball data set with 32K vertices and 192K tetrahedra.

tured meshes can be stored with minimal overhead using the same techniques, if the unstructured mesh is placed within the enclosing cube's boundaries.

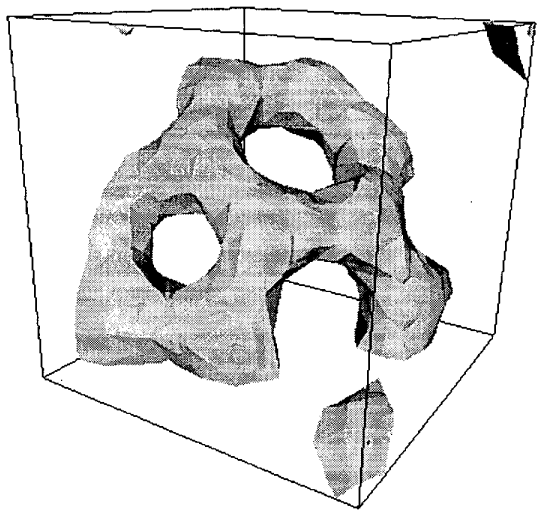

Figure 11. Sample isosurface using a field value in the 40% range for the bucky-ball data set (viewed from a different angle).

Once we have extracted a consistent mesh we often want to find isosurfaces within the extracted mesh. This is usually done by a two-step process which, in the absence of an index on the scalar field values (such as the pyramid [1]), must visit every tetrahedron in the extracted mesh to determine if it is crossed by the isosurface. On the other hand, our neighbor finding methods enable us to perform the mesh and isosurface extraction simultaneously by using the combined information (i.e., the scalar field value for the isosurface and the error threshold) when processing the complete multiresolution tetrahedral structure. Figure 11 is a sample isosurface for the bucky-ball data set in Figure 10 (viewed from a different angle).

References

[1] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 265–272, Nashville, TN, Apr. 1990. (Also *Proceedings of the Fifth Brazilian Symposium on Databases*, Rio de Janeiro, Brazil, April 1990, 15–26).

[2] P. Cignoni, L. DeFloriani, C. Montani, E. Puppo, and R. Scopigno. Multiresolution modeling and visualization of volume data based on simplicial complexes. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 19–26, Washington, DC, Oct. 1994.

[3] L. De Floriani, E. Puppo, and P. Magillo. A formal approach to multiresolution hypersurface modeling. In W. Strasser, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*, pages 302–323. Springer-Verlag, 1997.

[4] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization '97*, pages 81–88, Phoenix, AZ, Oct. 1997.

[5] W. Evans, D. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, to appear 2000. (Also University of Arizona Technical Report 97-09, May 1997).

[6] G. Fekete. Rendering and managing spherical data with sphere quadtrees. In *Proceedings of Visualization '90*, pages 176–186, San Francisco, CA, Oct. 1990.

[7] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, Dec. 1982.

[8] T. Gerstner and M. Rumpf. Multiresolutional parallel isosurface extraction based on tetrahedral bisection. In *Proceedings 1999 Symposium on Volume Visualization*, ACM Press, 1999.

[9] R. Gross, C. Luerig, and T. Ertl. The multilevel finite element method for adaptive mesh optimization and visualization of volume data. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization '97*, volume 24, pages 387–394, Phoenix, AZ, Oct. 1997.

[10] D. J. Hebert. Symbolic local refinement of tetrahedral grids. *Journal of Symbolic Computation*, 17:457–472, 1994.

[11] M. Lee and H. Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Transactions on Computer Graphics*, 19(2):79–121, Apr. 2000. (Also University of Maryland Computer Science TR–3900, April 1998).

[12] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time continuous level of detail rendering of height fields. In *Proceedings of the SIGGRAPH '96 Conference*, pages 109–118, New Orleans, Aug. 1996.

[13] J. M. Maubach. Local bisection refinement for n-simplicial grids generated by reflection. *SIAM Journal on Scientific Computing*, 16(1):210–227, Jan. 1995.

[14] M. Ohlberger and M. Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 56:365–385, 1997.

[15] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *Proceedings IEEE Visualization '98*, pages 19–26, Research Triangle Park, NC, Oct. 1998.

[16] H. Samet. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, 18(1):37–57, Jan. 1982.

[17] H. Samet. Neighbor finding in images represented by octrees. *Computer Vision, Graphics, and Image Processing*, 46(3):367–386, June 1989.

[18] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[19] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, Nov. 1990. (*Proceedings of San Diego Workshop on Volume Visualization*).

[20] J. Wilhelms and A. van Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 17–18, Washington, DC, Oct. 1994.

[21] P. L. Williams. Visibility ordering of meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, Apr. 1992.

[22] Y. Zhou, B. Chen, and A. Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization'97*, pages 135–142, Phoenix, AZ, Oct. 1997.