In *Proc. 8th Intl. Symp. on Spatial Data Handling*, Vancouver, Canada, July 1998, pp. 22–33

Traversing the Triangle Elements of an Icosahedral Spherical Representation in Constant-Time*

Michael Lee and Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
magus@umiacs.umd.edu
hjs@cs.umd.edu
Tel: (301) 405-1755
Fax: (301) 314-9115

Abstract

Techniques are presented for moving between adjacent triangles of equal size in a hierarchical representation for spherical data that is projected onto the faces of an icosahedron. The faces of the icosahedron are represented by a triangular quadtree. The operations are analogous to those used for a quadtree representation of data on the two-dimensional plane where the underlying space is tessellated into squares. A new technique is presented for labeling the triangular faces as well as the smaller triangles within each of the triangular faces of the icosahedron. The labeling enables the implementation of the quadtrees corresponding to the individual triangle faces of the icosahedron as linear quadtrees (i.e., pointer-less quadtrees). Outlines of algorithms are given for traversing adjacent triangles of equal size in constant time. The labeling and algorithms can also be used with minor modification (and no change from a computational complexity standpoint) with a hierarchical representation for spherical data that is projected onto the faces of an octahedron.

Keywords: Spherical representations, neighbor finding, data structures, algorithms, quadtrees

1 Introduction

The representation of spatial data is an important issue in geographic information systems (GIS). In this paper we are interested in the efficient representation of spherical data such as the surface of the Earth. In many applications it is desired to make use of a recursive decomposition of the underlying space such as a quadtree. Projecting the sphere onto the plane poses problems in that units of equal area in the projection do not necessarily correspond to units of equal area on the sphere. This has led

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 Number DAAH04-93-G-0106.

to approximations of the sphere by Platonic solids where the surface is projected onto the faces of an inscribed regular polyhedron. The faces are decomposed using conventional techniques such as the region quadtree (e.g., [11, 12]). There is one quadtree for each face where the sphere is represented as a collection of n quadtrees where n is the number of faces in the inscribed polyhedron. The most commonly used polyhedra are the octahedron ($n = 8$) and the icosahedron ($n = 20$) whose faces are equilateral triangles. We prefer the icosahedron [4] as it provides the best approximation of the sphere although the octahedron [1, 6] is also used since it can be aligned so that the poles are at opposite vertices of the octahedron and the prime meridian and the Equator intersect at another vertex. For example, Figure 1 shows the top level triangular faces of an icosahedron corresponding to the surface of the Earth where the continents are highlighted.

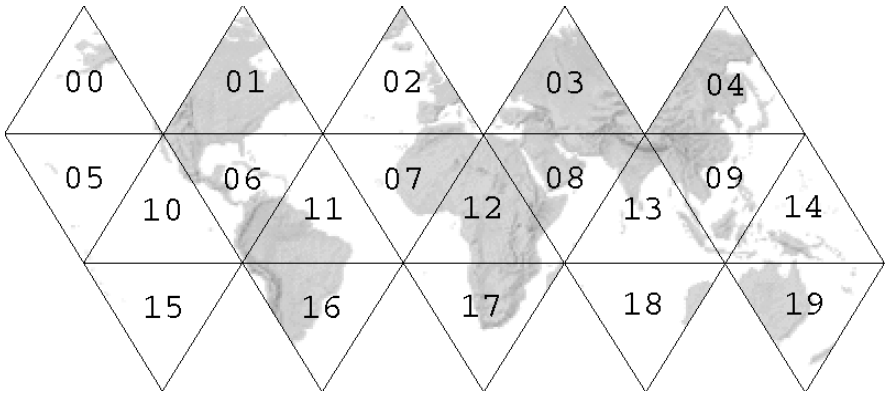

Figure 1: The top level triangular faces of an icosahedron corresponding to the surface of the Earth where the continents are highlighted.

In this paper we show how to move between adjacent triangular elements of an icosahedron approximation of the sphere by adapting traditional two-dimensional neighbor finding techniques [10, 11] for square quadtrees. We focus on linear quadtrees [5] which represent a quadtree as a collection of numbers corresponding to its leaf nodes thereby dispensing with the need for pointers. 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 of 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*.

One of the attractions of the linear quadtree when the faces correspond to squares (e.g., for the cube) is the ability to make use of binary arithmetic to navigate between any pair of adjacent nodes (i.e., corresponding to squares) of equal size in time independent of the depth of the quadtree at which the nodes are found [13]). In this paper we show how to adapt the linear quadtree to triangles and in particular to the icosahedron so such navigation can be performed. The adjacent triangles are not restricted to lie on the same face of the icosahedron. Our solution is in contrast to existing methods [3] which take time proportional to the maximum level of resolution and require the use of a number of tables in order to deal with transitions between triangles in different faces of the icosahedron. It is difficult to adapt these methods to use binary arithmetic as the transitions and node labels depend on orientation. Note that our approach is not restricted to the icosahedron and could also be used with an octahedron. In this case, again, our methods differ from existing methods (e.g., [6, 9]) which also take time proportional to the maximum level of resolution.

The rest of this paper is organized as follows. In Section 2 we present our method of labeling the triangle elements of the faces of the icosahedron, which is new. Section 3 shows how to find a neighbor of equal size within one of the triangles of the icosahedron in time proportional to the maximum level of resolution while Section 4 extends the method to the entire sphere (i.e., the 20 triangle quadtrees). Section 5 describes how to find the neighbors in constant time. Concluding remarks are drawn in Section 6.

2 Tree Node Labeling

Each of the icosahedron's 20 triangular faces is decomposed recursively into four equilateral triangles. The result is a triangle quadtree. These triangles always have one of two orientations: tip-up (Figure 2a) and tip-down (Figure 2b). *Tip-up* means that the corresponding triangle *points* upward (really toward the north pole). *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 short of some transformation such as rotation), we subdivide the two triangle types differently. 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 the triangles, we use the terms *vertical*, *left*, and *right* to refer to neighboring triangles of equal size along the horizontal, left angular, and right angular edges, respectively.

Figure 2: Possible triangle orientations: (a) tip-up, and (b) tip-down.

There are several advantages to using this coding scheme. If we use the top-most or bottom-most point to locate a triangle (since we only need one vertex, the orientation, and the size to determine the other two vertices), then 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, children are always one half of the size (one quarter of the area) of their 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 which is encoded using this node labeling method.

Our triangle labeling method is similar to that used for the octahedron [1, 2, 6]. However, the difference is that the neighbor finding methods that are used there take time proportional to the maximum level of resolution although they are based on the same principle as our methods that have the same execution time complexity. In contrast, our triangle labeling method is very different from that used by Fekete [3] for the icosahedron and likewise for the neighbor finding methods. In particular, Fekete's labeling method is based on a 'floating' scheme where the labels associated with each triangle are based not on its global orientation but, instead, are based on that of its parent. This enables the path components of all location codes that correspond to the neighbors of a particular triangle to differ by one directional code (at different depths of the hierarchy) at the expense of

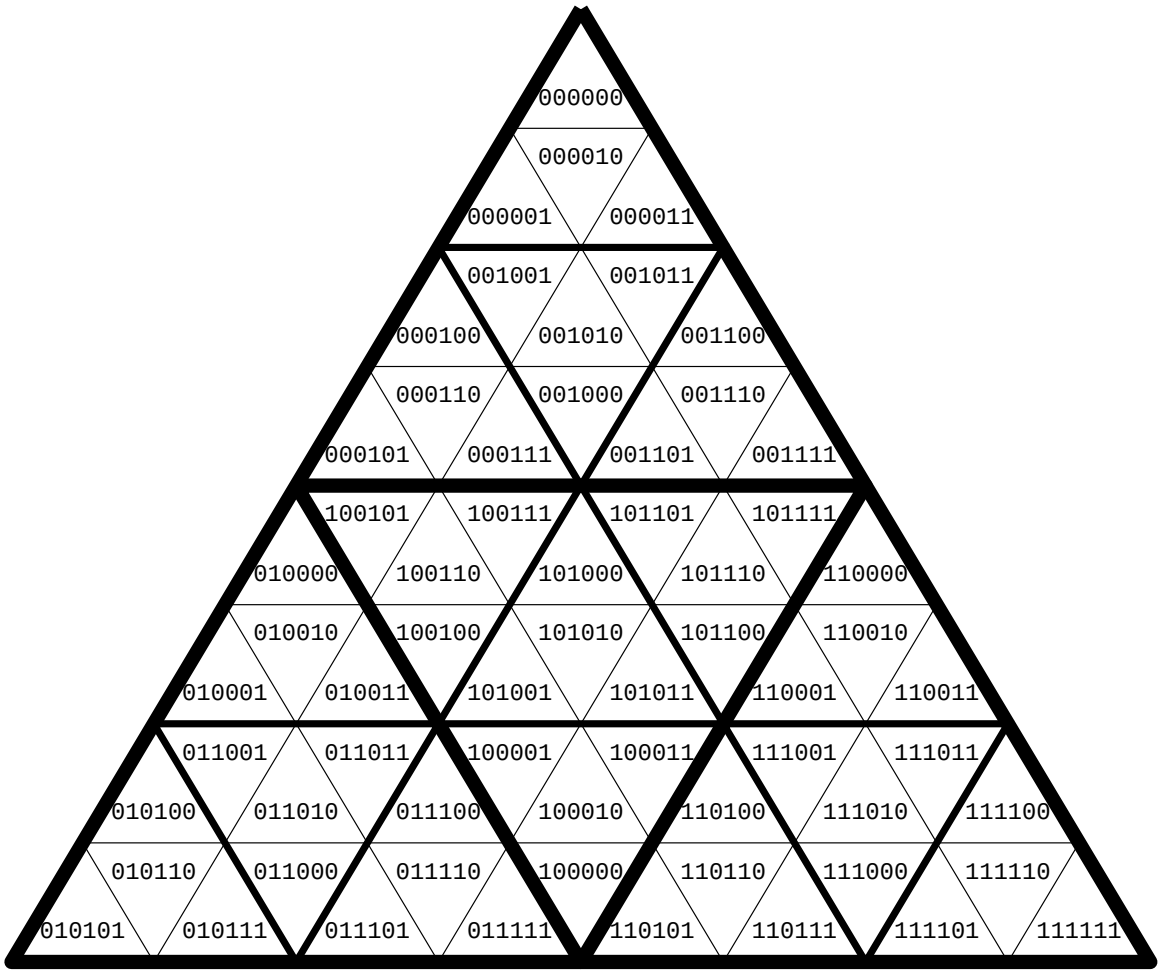

Figure 3: Example labeling of a tree which is three levels deep.

added complexity. In contrast, our labeling method is considerably simpler and can be enhanced easily to yield a neighbor finding method that locates equal-sized neighbors in constant time.

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 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. Our algorithm is decomposed into three steps to make it easier to understand. The first step finds an ancestor of p which also contains the desired neighbor q of p. This node is called the *nearest common ancestor* of q and p. The technique used for finding the nearest common ancestor is effectively the same as that found in most standard quadtree implementations [10, 11] 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* which must contain the right neighbor of *p*. We stop when we encounter a node that has a right sibling (or the parent contains a node that is adjacent and to the right of *p*). If we look at Figure 2a, then we see that this is true for children `01` and `10`. Also, in Figure 2b, 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. A similar analysis is used to determine the nearest common ancestor when finding the left or vertical neighbor of a node.

Step two identifies and sets the position in the path array to the child type of the nearest common ancestor (found in step one). This step is easy. Let's 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*, then it is easy to determine what child contains *q*. We move left. If child `10` contains *p*, then child `01` must contain the neighbor node *q*. If we were looking for a right neighbor, then we move right. The same procedure also holds for vertical neighbors.

The final step 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 the 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. Of course, the layout of the children that we have chosen (see Section 2) keeps things simple. Keeping in mind that a tip-up triangle is always adjacent to a tip-down triangle (and vice versa), reflection for the triangles has 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 2a and 2b are vertical reflections of each other).

Since step one (finding the nearest common ancestor) involves examining each two-bit pair in the path array of the location code, the computational complexity 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 the computational complexity is on the order of the length of the code. Overall, neighbor finding requires time proportional to the length of the location code.

4 Extensions to the Entire Sphere

Indexing the entire icosahedron (rather than just one of its faces) requires 20 of the previously described triangular quadtrees. This means that whenever we reach the top level (or root) of one of these trees, special work is required although it doesn't really create any substantial difficulties.

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 in 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 triangular faces of the icosahedron are numbered isn't important since tables will be used most of the time. Thus we 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 all tip-up, 5 to 9 are all tip-down, 10 to 14 are all tip-up, and 15 to 19 are all tip-down.

Neighbor finding involves several modifications to our previous algorithm, but, as we show, the 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, then we stop looking for the nearest common ancestor. Obviously, the entire sphere contains every possible location and is therefore an ancestor to every node. We always stop at the top level. Also, note that since Figure 1 is really a sphere, every triangle has a neighbor in every direction (triangles on the ends wrap around), so we are well prepared for step two.

Step two is basically unchanged. The only modification is the use of a relation to indicate how to update the path array component corresponding to the child of the root. It summarizes the actions for all possible neighbors from Figure 1. This relation is used only when the nearest common ancestor from step one is the entire sphere.

Step three requires one more relation 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 same technique described for neighbors in the same face in Section 3. The rationale for this additional relation is as follows. If we consider the left neighbor case and use a standard "mirror reflection", then 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, then 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.

5 Constant Time Neighbor Finding Algorithm

We now describe how to find neighbors in constant time. Only the ideas behind the algorithms are given (see [7] for pseudo-code). The algorithms make use of the carry (borrow) property of addition (subtraction) to 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 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. 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 [13] 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.

Neighbor finding in square quadtrees is achieved in constant time by making use of the equivalence between the path array 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-left-most corner), chosen in a consistent manner, of the blocks corresponding to the leaf nodes. The result of the bit interleaving is also known as the *Morton code* [8, 11]. For example, if we want the Morton code based on coordinates x and y, then the code 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, then we start this process by incrementing x_0 by one. If there is a carry, then we add one to x_1. If there is another carry, then 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 [13] 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.

Using standard Morton codes for square quadtrees, we find a neighbor by addition by skipping 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. Nevertheless, the values of the path array of the triangle quadtree can be manipulated in an analogous manner to that of the values of the path array of the square quadtree. In the rest of this section we show how this is done.

We first consider a transition from one triangle to its right neighbor. This requires that 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 brothers. 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 4a). This is the case when the neighboring triangles are not brothers. Making a transition from a `11` child to a `00` child is handled by use of addition. Basically, we add one to the bit string represented by the path array 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 properly handled:

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

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

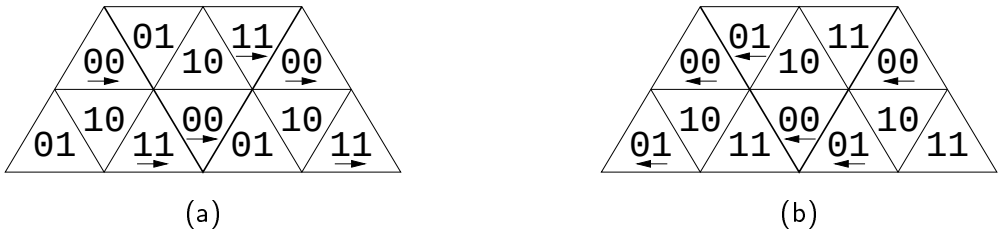

Figure 4: (a) Examples of rightward transitions that generate a carry (denoted by a rightward pointing arrow). (b) Examples of leftward transitions that generate a borrow (denoted by a leftward pointing arrow).

The 00 case is handled by using the concept of an *idmask* which we name 00ID11. It is formed by invoking a procedure MAKE_IDMASK(INPUT,V,MP) which sets all pairs of bits in *idmask* for which the corresponding bit pairs in INPUT have value V to MP, while the bits corresponding to the other bit pairs are set to 00. For right neighbors, the idmask 00ID11 is formed by a call to MAKE_IDMASK(INPUT,00,11) and contains 11 in the bit positions of the path array that have the value 00 and 00 in the bit positions that have other values. We use the marking pattern 11 because taking its exclusive or with any input sequence will ensure that all pairs of bits with value 00 will be changed to 11 and all bit positions with other bit pattern value pairs will be left alone since the result of applying 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.

Therefore, finding a right neighbor of equal size proceeds as follows. Replace all occurrences of 00s in the input with 11s by taking the exclusive or of the input with the idmask 00ID11. Next, add 1. After the addition, apply the following two steps:

1. Change all 11s not affected by the addition (which were originally 00s) back to 00 by taking the exclusive or of idmask 00ID11 with the result of the addition thereby creating a bit pattern *t*. This leaves all pairs of bits that were not originally 00 alone since the result of applying exclusive or of any bit value *i* with 0 is *i*. It also resets to 00 all 11s at positions in the original path array which originally contained 00 (which is desired) and resets to 11 all 00s at positions in the original path array which originally contained 00.

2. Change all 11s which were affected by the addition and thus became 00 (again, only the ones which were originally 00s) to 01 (as 00 plus one is 01) by constructing a mask which has a 11 at every pair of positions in the original path array which did not contain 00 (obtained by complementing idmask 00ID11). Next, or this mask with EVENBITMASK (an alternating bit pattern starting with 0 at its left end — that is, 010101...) which marks the even positions in the original path array which were part of the 00 pair with a 01. Taking the and of the resulting mask with *t* yields the desired result.

As an example, 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 both RCODE[1] and RCODE[4] have value 00. Taking the exclusive or of RCODE with 00ID11 changes RCODE to 11011111. Adding one to RCODE changes it to 11100000. Taking an exclusive or of RCODE with 00ID11 changes RCODE to 00100011. Taking the or of EVENBITMASK with the complement of 00ID11 yields 01111101. The final and of the latter with RCODE changes RCODE to 00100001. This example is illustrated in Figure 5aa.

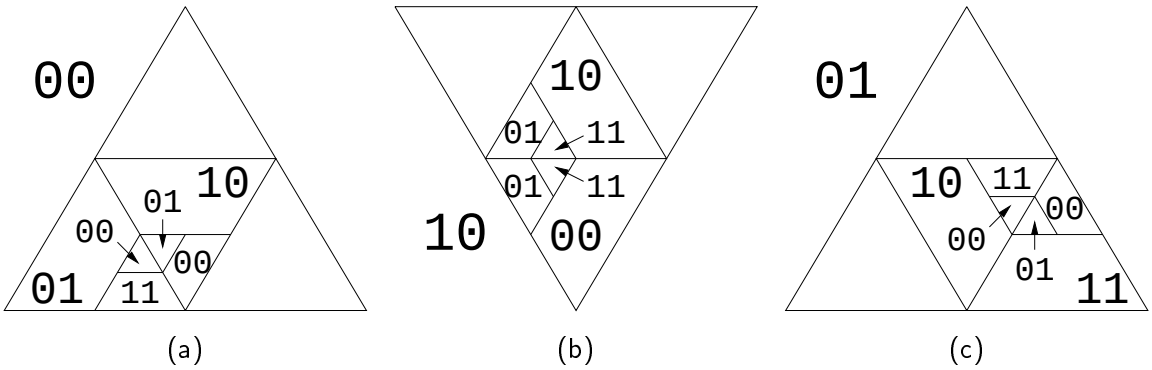

Figure 5: Examples showing how to find neighbors of equal size: (a) right neighbor of 00011100, (b) vertical neighbor of 10100111, (c) left neighbor of 01110001.

Next, we consider a leftward transition. 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 brothers, we subtract 1 from the path array value of the location code and propagate a borrow when moving between triangles that are not brothers.

Below, we look at leftward transitions from the different children. Transitions from a 10 child to a 01 child or from a 11 to a 10 child are achieved by subtracting one when the neighboring triangles are brothers. 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 4b). This is the case when the neighboring triangles are not brothers. Making a transition from a 00 child to a 11 child is handled easily by 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 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 properly handled:

1. A borrow will be generated if necessary (i.e., the 01 is at the extreme right of the path array)
2. A borrow will be properly propagated (i.e., the 01 is the recipient of a borrow).

As in the case of the rightward movement, the 01 case is handled by using the concept of an *idmask*. The idmask identifies the bit positions where we need to modify the path array value before and after performing the subtraction. However, unlike the rightward movement, we must identify the bit positions in the path array that have the value 01 and change them to 00 prior to the subtraction while leaving all other bit pattern pairs alone. This is not easily done if we were to use the marking pattern of 11, as we did in the case of a rightward movement, since our goal is to change a bit pattern pair whose two values are not the same. This task is more easily accomplished by observing that the result of taking the exclusive or of bit pattern pair 01 with the bit pattern pair 01 is 00, while the result of taking the exclusive or of all other bit pattern pairs with the 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(INPUT,01,01)`.

Therefore, finding a left neighbor of equal size proceeds as follows. Replace all occurrences of 01s in the input with 00s by taking the exclusive or of the input with the idmask 01ID01. Next, subtract 1. After the subtraction, apply the following two steps:

1. Change all 00s not affected by the subtraction (which were originally 01s) back to 01 by taking the exclusive or of idmask 01ID01 with the result of the subtraction thereby creating a bit pattern *t*. This leaves all pairs of bits that were not originally 01 alone since the result of applying exclusive or of any bit value *i* with 0 is *i*. It also resets to 01 all 00s at positions in the original path array which originally contained 01 (which is desired) and resets to 10 all 11s at positions in the original path array which originally contained 01.
2. Change all 00s which were affected by the subtraction and thus became 11 (again, only the ones which were originally 01s) to 00 (because 01 minus one is 00) by constructing a mask which has a 11 at every pair of positions in the original path array which did not contain 01, and a 01 in the positions that did contain 01 (obtained by taking the complement of the result of shifting idmask 01ID01 to the left by one bit position). Taking the and of the resulting mask with *t* yields the desired result.

As an example, 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 LCODE[1] and RCODE[4] have value 01. Taking the exclusive or of LCODE with 01ID01 changes LCODE to 00110000. Subtracting one from LCODE changes it to 00101111. Taking the exclusive or of LCODE with 01ID01 changes LCODE to 01101110. Complementing the result of shifting 01ID01 by one bit to the left yields 01111101. The final and of the latter with the value of LCODE changes LCODE to 01101100. This example is illustrated in Figure 5ac.

We now examine a vertical transition. It differs from rightward and leftward transitions in that the path array values do not change except for the transition between brother triangles. In particular, we need to make one, and only one, transition from the least significant 00 child (i.e., right-most in the path array) to the least significant 10 child or vice versa (i.e., from the least significant 10 child to the least significant 00 child). This is done by identifying the rightmost ?0 child and complementing the left bit of its bit pattern pair value. All remaining bit pattern pairs are left alone. Once again, we 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. It is formed by a call to MAKE_IDMASK(INPUT,?0,10).

Finding a vertical neighbor proceeds as follows: Create a new mask *m* from ?0ID10 which 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* which 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* yielding *p* means that all 11s to the right of the rightmost 01 have become 00 while the rightmost 01 has become a 10. All other bit pair positions in *n* are unchanged. We can now obtain our desired mask *m* by taking the and of *p* and ?0ID10. The reason is that all items to the left of the rightmost 10 in *p* are the complement of the corresponding items in ?0ID10 while all items to the right of the rightmost 10 in *p* are 0. Our final step is an exclusive or 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 result of applying exclusive or of any bit value *i* with 1 is the complement of *i*.

As an example, let us 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. Taking the complement of ?0ID10 yields 01011111 which is stored in variable MASK. Adding one to MASK yields 01100000. Taking the and of MASK with ?0ID10 changes MASK to 00100000. The final exclusive or of MASK with VCODE changes VCODE to 10000111. This example is illustrated in Figure 5ab.

We now show how to make transitions across different faces of the icosahedron. They arise if the addition steps in the rightward and vertical transitions generated a carry past the left-most end of the the path array or if the subtraction step in the leftward transition generated a borrow past the left-most end of the the path array. In this case, a carry (borrow) or overflow indicator is set. This flag is tested by a one cycle machine instruction on most computer architectures.

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. Left and right transitions between nodes corresponding to the faces of the icosahedron labeled 00 to 04 and 15 to 19 are handled in the same way as in Section 4 except that we now want to perform them in 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 just like vertical transitions in that we use reflection. The difference is that we perform reflection for all occurrences of 11 and 01 for right and left neighbors, respectively. These situations are identified by complementing the left bit of the bit pattern value of each ?1 child. This is done by using the idmask ?1ID10 which identifies the bit positions in the path array of the input with value ?1 and marks them with 10. It is formed by a call to MAKE_IDMASK(INPUT,?1,10) All remaining bit pattern pairs are left alone.

We use a marking pattern of 10 as it changes child 01 to 11 (for left neighbor reflection) and child 11 to 01 (for right neighbor reflection) using the exclusive or operation. In fact, the desired neighbor is obtained by taking the exclusive or of the original input value with idmask ?1ID10. The same technique works for both left and right neighbors. In particular, when it is invoked 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 can only contain the bit pattern pairs with values 00 and 01. Thus all 01s are 'marked' by ?1ID10 (with the pattern 10). Therefore, one application of exclusive or to the input with ?1ID10 changes all 01s to 11s as desired. Similarly, when it is invoked 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 can only contain the bit pattern pairs with values 00 and 11. Thus all 11s are 'marked' by ?1ID10 (with the pattern 10). Therefore, one application of exclusive or to the input with idmask ?1ID10 changes all 11s to 01s as desired.

6 Conclusions and Future Work

The triangle coding scheme described in this paper provides a new way to handle spherical data using a standard quadtree-like approach. Our algorithms assumed an icosahedron but they also work for the tetrahedron and the octahedron. The only modification that is needed is to include a mechanism to handle the case that the neighboring triangles are in different base triangles of the solid (i.e., tetrahedron and octahedron). Our coding scheme is particularly useful for traversing the

triangular elements. We did not address other operations such as determining whether two triangle elements 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 triangle elements that connect any two points of the sphere [6].

References

[1] G. Dutton. Geodesic modeling of planetary relief. *Cartographica*, 21(2&3):188–207, Summer & Autumn 1984.

[2] G. Dutton. Locational properties of quaternary triangular meshes. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, vol. 2, pages 901–910, Zurich, Switzerland, July 1990.

[3] G. Fekete. Rendering and managing spherical data with sphere quadtrees. In *Proceedings IEEE Visualization'90*, A. Kaufman, ed., pages 176–186, San Francisco, October 1990.

[4] G. Fekete and L. S. Davis. Property spheres: a new representation for 3-d object recognition. In *Proceedings of the Workshop on Computer Vision: Representation and Control*, pages 192–201, Annapolis, MD, April 1984. Also University of Maryland Computer Science Technical Report TR–1355, December 1983.

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

[6] M. F. Goodchild and Y. Shiren. A hierarchical spatial data structure for global geographic information systems. *CVGIP: Graphical Models and Image Understanding*, 54(1):31–44, January 1992.

[7] M. Lee and H. Samet. Navigating throught triangle meshes implemented as linear quadtrees. Computer Science Technical Report TR–3900, University of Maryland, College Park, MD, April 1998.

[8] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.

[9] E. J. Otoo and H. Zhu. Indexing on spherical surfaces using semi-quadcodes. In *Advances in Spatial Databases—3rd International Symposium, SSD'93*, D. Abel and B. C. Ooi, eds., vol. 692 of Springer-Verlag Lecture Notes in Computer Science, pages 510–529, Singapore, June 1993.

[10] H. Samet. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, 18(1):37–57, January 1982. Also in *Digital Image Processing and Analysis: Vol. 2: Digital Image Analysis*, R. Chellappa and A. Sawchuck, eds., pages 399–419, IEEE Computer Society Press, Washington, DC, 1986; and University of Maryland Computer Science Technical Report TR–857, January 1980.

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

[12] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[13] G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Understanding*, 55(3):221–230, May 1992.