

ABSTRACT

Title of dissertation: SPATIAL MODELING USING
TRIANGULAR, TETRAHEDRAL, AND
PENTATOPIC DECOMPOSITIONS

Michael Thomas Lee, Doctor of Philosophy,
2006

Dissertation directed by: Professor Hanan Samet
Department of Computer Science

Techniques are described for facilitating operations for spatial modeling using triangular, tetrahedral, and pentatopic decompositions of the underlying domain. In the case of terrain data, techniques are presented for navigating between adjacent triangles of a hierarchical triangle mesh where the triangles are obtained by a recursive quadtree-like subdivision of the underlying space into four equilateral triangles. We describe a labeling technique for 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. This results in algorithms that have a worst-case constant time complexity, as they make use of a fixed number of bit manipulation operations.

In the case of volumetric data, we consider a multi-resolution representation based on a decomposition of a field domain into nested tetrahedral cells generated by recursive tetrahedron bisection, that we call a *Hierarchy of Tetrahedra (HT)*. We describe our implementation of an HT, and discuss how to extract conforming

meshes from an HT so as to avoid discontinuities in the approximation of the associated scalar field. This is accomplished by using worst-case constant time neighbor finding algorithms. We also present experimental results in connection with a set of basic queries for performing analysis of volume data sets at different levels of detail.

In the case of four-dimensional data which can include time as the fourth dimension, we present a multi-resolution representation of a four-dimensional scalar field based on a recursive decomposition of a hypercubic domain into a hierarchy of nested four-dimensional simplexes, that we call a *Hierarchy of Pentatopes (HP)*. This structure allows us to generate conforming meshes that avoid discontinuities in the corresponding approximation of the associated scalar field. Neighbor finding is an important part of this process and using our structure, it is possible to find neighbors in worst-case constant time by using bit manipulation operations, thereby avoiding traversing the hierarchy.

SPATIAL MODELING USING TRIANGULAR, TETRAHEDRAL,
AND PENTATOPIC DECOMPOSITIONS

by

Michael Thomas Lee

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:

Professor Hanan Samet, Chair/Advisor

Professor Larry Davis

Professor Leila De Floriani

Professor Shunlin Liang

Professor Amitabh Varshney

© Copyright by
Michael Thomas Lee
2006

TABLE OF CONTENTS

List of Figures	iv
1 Introduction	1
1.1 Motivation	1
1.2 Related Work	8
1.3 Outline of Thesis	13
2 Two-Dimensional Triangle Quadtrees	14
2.1 Tree Node Labeling	14
2.2 Neighbor Finding	17
2.2.1 Step One : Locating the Nearest Common Ancestor	19
2.2.2 Step Two : Updating the Path to Contain the Neighbor	21
2.2.3 Step Three : Updating the Rest of the Path to the Neighbor	23
2.2.4 Putting it all Together to Find a Neighbor	25
2.3 Extensions to the Entire Sphere	26
2.4 Constant-Time Neighbor Finding Algorithm	32
2.4.1 Square Quadtrees	33
2.4.2 Rightward Transitions	37
2.4.3 Leftward Transitions	45
2.4.4 Vertical Transitions	50
2.4.5 Transitions Across Different Faces of the Icosahedron	53
2.5 Neighbor Finding Using Octahedra and Tetrahedra	58
2.5.1 Octahedron	58
2.5.2 Tetrahedron	61
2.6 Finding Neighbors of Greater or Equal Size	64
2.7 Comparison with Method based on Icosahedron	65
3 Three-Dimensional Hierarchies of Tetrahedra	84
3.1 A Hierarchy of Tetrahedra	84
3.1.1 Tetrahedral Decomposition	84
3.1.2 Labeling Tetrahedra in an HT	85
3.1.3 Encoding an HT	87
3.2 Neighbor Finding	88
3.2.1 Locating the Nearest Common Ancestor	89
3.2.2 Updating the Location Code	91
3.2.3 Extensions to the Entire Cube	92
3.3 Constant-Time Neighbor Finding Algorithm	93
3.3.1 Neighbor Type 1	95
3.3.2 Neighbor Type 2	95
3.3.3 Neighbor Type 3	98
3.3.4 Neighbor Type 4	102
3.3.5 Updating the Neighbor Mask	103
3.3.6 Transitions Across the Six Top Level Tetrahedra	104

3.4	Edge Neighbors	104
3.5	Extracting a Conforming Tetrahedral Mesh	105
3.5.1	Axis-aligned Clusters	107
3.5.2	Plane-aligned Clusters	107
3.5.3	Non-aligned Clusters	109
3.6	Algorithms for Selective Refinement	110
3.6.1	A Depth First Approach	110
3.6.2	A Priority Based Approach	113
3.6.3	An Incremental Approach	115
3.7	Experimental Results	118
4	Four-Dimensional Hierarchies of Pentatopes	124
4.1	Pentatopic Decomposition	124
4.2	Labeling Pentatopes in an HP	127
4.3	Neighbor Finding	130
4.3.1	Locating the Nearest Common Ancestor	131
4.3.2	Updating the Location Code	132
4.3.3	Extensions to the Entire Hypercube	133
4.4	Constant-Time Neighbor Finding Algorithm	133
4.4.1	Neighbor Type 1	136
4.4.2	Neighbor Type 2	136
4.4.3	Neighbor Types 3, 4, and 5	138
4.4.4	Updating the Neighbor Mask	143
4.4.5	Transitions Across the 24 Top Level Pentatopes	145
4.5	Clusters of Pentatopes in an HP	146
4.6	A Depth First Algorithm for Selective Refinement	147
4.7	Experimental Results	149
5	Conclusions	157
	Bibliography	161

LIST OF FIGURES

1	Two possible orientations for a triangle: (a) tip-up, and (b) tip-down.	15
2	Labeling of a tree which is three levels deep.	17
3	STOPTAB(Neighbor_Direction,Child_Type) relation indicating when to cease the search for the nearest common ancestor in step 1 of the neighbor finding algorithm.	21
4	NEXTTAB(Neighbor_Direction,Child_Type) indicating the child type of the neighboring child of the nearest common ancestor.	22
5	Example showing the top-level triangle faces of an icosahedron corresponding to the surface of the Earth.	27
6	Execution trace of procedure EXT_STEP_ONE for the left neighbor of 000010010001010001.	28
7	NEXTTOP(Neighbor_Direction,Child_Type) indicating neighbors for the triangles corresponding to the faces of the icosahedron.	29
8	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.	30
9	Examples of rightward transitions that generate a carry (denoted by a rightward pointing arrow) as the neighboring triangles are not siblings.	38
10	The 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'.	40
11	Example showing the steps in the generation of idmask 00ID11 for some input values.	41
12	The effect of procedure CONSTANT_RIGHT on different bit pattern pair values depending on whether or not there is an incoming carry from the right.	45
13	Examples showing how to find neighbors of equal size: (a) right neighbor of 00011100, (b) left neighbor of 01110001, (c) vertical neighbor of 10100111.	46
14	Examples of leftward transitions that generate a borrow (denoted by a leftward pointing arrow) as the neighboring triangles are not siblings.	47

15	Example showing the top-level triangle faces of an octahedron.	59
16	NEXTOCT(Neighbor_Direction,Child_Type) indicating neighbors of the triangles corresponding to the faces of the octahedron.	60
17	Example showing the top-level triangle faces of a tetrahedron.	62
18	NEXTTET(Neighbor_Direction,Child_Type) indicating neighbors for the triangles corresponding to the faces of the tetrahedron.	62
19	Example showing the triangle adjacencies of the tetrahedron.	63
20	Fekete's labeling scheme.	66
21	Tree using Fekete's scheme.	67
22	(a) Open direction A; (b) open direction B; (c) open direction C; (d) all three directions are open.	70
23	List of rules used in Fekete's algorithm.	72
24	Rules used to keep track of global status.	74
25	Layout for V=3, L=2, and R=1.	74
26	Steps taken while finding neighbors of 123 using Fekete's method. . .	77
27	Steps taken while finding neighbors of 143 using Fekete's method. . .	79
28	Example of one substitution table to correlate between the different orientations used to label two adjacent base triangles of the icosahedron.	83
29	Subdivision of the initial cubic domain into six tetrahedra.	85
30	Labeling of a 1/2 pyramid.	86
31	Labeling of a 1/4 pyramid.	87
32	Labeling of a 1/8 pyramid.	87
33	Nearest common ancestor of 210011.	90
34	Neighbor type 3 of 210011.	91
35	Neighbor type 4 of 210011.	94
36	Neighbor type 2 of 1010101011010.	97

37	Neighbor type 3 of 1010101011010.	101
38	Table indicating how to proceed at each level when searching for the neighboring tetrahedron.	103
39	Neighbor sequences for all edges of the tetrahedra.	105
40	Axis-aligned cluster.	107
41	Steps required to find an axis-aligned cluster.	108
42	Plane-aligned cluster.	108
43	Steps required to find a plane-aligned cluster.	109
44	Non-aligned cluster.	109
45	Steps required to find a non-aligned cluster.	110
46	The second column shows the number of tetrahedron splits per second, the third column shows the number of cluster computations per second.	115
47	Uniform LOD extraction (a): error threshold equal to 5.0% of the field range of the whole domain. The isosurface for a field value equal to 100.0 is shown. Variable LOD extraction based on a field value (b): error threshold equal to 0.1% of the field range enforced near isosurface of value 1.27 (blue). The isosurfaces for field values equal to 1.27 and 1.45 are shown.	120
48	Number of tetrahedra in the meshes at <i>uniform</i> LOD and percentage with respect to the number of tetrahedra in the mesh at full resolution extracted from the HT representation of the Plasma and Buckyball data sets, respectively.	121
49	Number of tetrahedra in the meshes at <i>variable</i> LOD based on a region of interest and percentage with respect to the number of tetrahedra in the mesh at full resolution extracted from the HT representation of the Buckyball data set. The error within the region is specified in the left column.	121
50	Number of tetrahedra in the meshes at <i>variable</i> LOD and percentage with respect to the number of tetrahedra in the mesh at full resolution extracted from the HT representation of the Plasma data set. The error within the proximity of the isosurface is specified in the left column.	122

51	Example of (a) an <i>h</i> -pentatope, (b) a <i>c</i> -pentatope, (c) an <i>s</i> -pentatope and (d) an <i>e</i> -pentatope. The figures show the unfolding in 3D space of each pentatope by representing its five tetrahedral faces.	126
52	Table with splitting rules. The second column denotes the shape of the pentatope σ which is split, the third column (G) indicates whether the parent of σ is child 0 or child 1 of the grandparent of σ , the fourth column (P) indicates whether σ is child 0 or child 1 of its parent, the fifth column shows the split edge of σ , the sixth column shows the pentatopes resulting from the split, and their vertices.	129
53	The table indicates how to proceed at each level when searching for the neighboring pentatope.	134
54	Example of neighbor type 2.	138
55	Example of neighbor type 3.	141
56	Swaps performed as a result of various bit changes.	144
57	Uniform LOD extraction from the Buckyball data set: error threshold equal to 1% of the field range. The isosurface for timestep 4 and field value 100 is shown.	151
58	Variable LOD based on a region of space in the Buckyball data set: error threshold equal to 1% of the field range within the selected area, and arbitrarily large elsewhere. The isosurface for timestep 4 and field value 100 is shown.	152
59	Variable LOD based on field value in the Buckyball data set: error threshold equal to 1% of the field range on the tetrahedra intersected by the isosurface with field value 100, and an error threshold arbitrarily large elsewhere. The isosurface for timestep 4 and field value 100 is shown. The number of pentatopes in the resulting 4D mesh is 70% of the number of pentatopes in the mesh obtained in the uniform LOD extraction (see Figure 57).	153
60	Uniform LOD extraction from the Ritchmyer data set: error threshold equal to 1% of the field range. The isosurface for timestep 4 and field value 100 is shown.	154
61	Variable LOD based on a region of space in the Ritchmyer data set: error threshold equal to 1% of the field range within the selected area, and arbitrarily large elsewhere. The isosurface for timestep 4 and field value 100 is shown.	155

62	Variable LOD based on field value in the Ritchmyer data set: error threshold equal to 1% of the field range on the tetrahedra intersected by the isosurface with field value 100, and an error threshold arbitrarily large elsewhere. The isosurface for timestep 4 and field value 100 is shown. The number of pentatopes in the resulting 4D mesh is 92% of the number of pentatopes in the mesh obtained in the uniform LOD extraction (see Figure 60).	155
63	Number of pentatopes in the meshes along with the percentage with respect to the number of pentatopes at full resolution, and the number of tetrahedra in the time slice with value 4 along with the percentage with respect to the number of tetrahedra at full resolution.	156
64	Number of pentatopes in the meshes along with the percentage with respect to the number of pentatopes at full resolution, and the number of tetrahedra in the time slice with value 4 along with the percentage with respect to the number of tetrahedra at full resolution.	156

Chapter 1

Introduction

1.1 Motivation

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). 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.

One interesting case is when a 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. This partitioning strategy is similar to methods based on the region quadtree [37, 43] (see also [67, 68]), where triangular regions are used instead of rectangular regions. We term the result a *triangle quadtree*.

Such meshes find uses in many applications such as finite element analysis (e.g., [3, 4, 10, 14, 20, 21, 35, 42, 61, 69, 78]), or for defining multiresolution representations of surfaces like subdivision surfaces [40]. 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., [3]). 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.

Triangular meshes 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., [15]). Traditional ways of representing such data invariably resort to projections onto the plane (e.g., [74]) using one of many possible projections (e.g., [71]). Clearly, there is no perfect projection. Such applications 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., [17, 18, 23, 24, 30, 56]), 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 data.

The quadtree representation of the mesh corresponding to the surface 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 which do not use pointers. The most common of these representations is known as the *linear quadtree* [27] where the quadtree is represented as a collection of numbers corresponding to its leaf nodes. In particular, each face of the inscribed polyhedron is represented by a separate quadtree where 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*.

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 [70]). This enables the navigation to be performed very efficiently, as it just requires a few bit manipulation operations which can be implemented in hardware using just a few machine language instructions. 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 can be used in a ray tracer where a surface is represented by a quadtree triangle mesh instead of a quadtree square mesh [25, 29, 41, 66, 73]. 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” [20] by applying a “remeshing” process that yields a triangular hierarchy where groups of four triangles are aggregated into larger triangles (but see [14] 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., [20]) can be traversed efficiently using our techniques. In other applications (e.g., [3]), 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., [42, 78]) to overcome the fact that the mesh elements are square (e.g., [10], 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.

Hierarchical data representations play an important role when working with three-dimensional data. We consider the problem of modeling volume data sets, i.e., sets of points spanning a domain in the three-dimensional space, and having one or more scalar field values associated with each data point. Hierarchical modeling of volume data is useful in several applications, including scientific visualization, medical imaging, simulation, and finite element analysis. A volume data set can

be modeled by decomposing its domain using a tetrahedral mesh with vertices at the data points. When the data points are given at the vertices of a regular cubic grid, the resulting decomposition is a mesh generated by a recursive decomposition of tetrahedra based on the vertices of the regular grid.

Hierarchical models for volume data are an instance of *multi-resolution models*, also called *Level-Of-Detail (LOD) models*, which have been widely used for describing surfaces and two-dimensional height fields (see [13] for a survey). A *virtually continuous* set of simplified meshes at different LODs can be generated from a multi-resolution model. The resolution (i.e., the density of the cells) of an approximating mesh may vary in different parts of the field domain, or in the proximity of interesting field values. It has been shown [12] that queries on a LOD model are instances of *selective refinement*, which is the process of extracting meshes at a variable resolution from a multi-resolution model.

In our work, we consider hierarchical meshes generated by recursive bisection of a tetrahedron along its longest edge, and we use the term *Hierarchies of Tetrahedra (HTs)* to describe them. Such meshes have been used for multi-resolution modeling of regularly-spaced volume data sets because of their ability to generate highly adaptive domain decompositions.

When performing selective refinement on hierarchical models, a major issue is the topological consistency of the extracted mesh, which must be ensured to avoid discontinuities in the approximation of the scalar field. Consistency must be maintained while applying local refinement or coarsening. Thus extracting a consistent mesh involves detecting those tetrahedra which form clusters that must be split, or

merged, simultaneously. Detecting these clusters requires neighbor finding [65, 67].

A four-dimensional representation addresses the problem of modeling time-varying volumetric data sets, i.e., sets of points in the three-dimensional Euclidean space describing a scalar field at different instances of time. Time-varying scalar fields arise in engineering, biomedical and other scientific applications, which produce very large data sets by numerical simulations or acquisition. The growth in the capability for computing and storing large data sets has resulted in incredible quantities of large simulation data. The huge size of available data sets poses interesting challenges for inspecting, analyzing and visualizing such data, that naturally leads to the investigation of hierarchical methods to control and adjust the level of detail of a given data set. The purpose of such models is to support selective refinement (i.e., extraction of adaptively refined representations) as well as progressive transmission efficiently, thus reducing space requirements and enhancing computational performance.

Time-varying volumetric data sets are sets of points in the three-dimensional Euclidean space describing a scalar field (e.g., pressure, temperature, strength of an electric, or a magnetic field) at different instances of time. Many visualization tools treat them as collections of 3D scalar fields. This does not take into account the fact that oblique cross sections can be very relevant features, or that smooth animation at interactive rates are often needed. These operations are not supported when a time-varying volume data set is modeled as a collection of representations of 3D scalar fields, each corresponding to a different time slice. Thus, time-varying data sets are often viewed as four-dimensional scalar fields by considering time as

the fourth dimension [39, 76]. These 4D scalar fields are analyzed by extracting isosurfaces, consisting of tetrahedral cells, which are visualized by cutting them with different planes or through direct volume rendering techniques. The domain of a 4D scalar field can be modeled either as a hypercubic grid, or as a simplicial mesh with vertices at the data points, obtained by triangulating the hypercube. Isosurface extraction algorithms, however, are much simpler on simplicial meshes.

Multi-resolution representations are a very effective way for handling large data sets describing time-varying scalar fields because of their ability of focusing attention on a region of interest and reducing the size of the representation. This will allow not only visualization and inspection of large-size time-varying data sets in real-time, but it will also effectively support analysis and visualization of salient features of scientific data sets. As mentioned above, an effective way of dealing with time-varying data sets is to model them as 4D scalar fields.

In this work, we consider a recursive decomposition of a hypercube into a hierarchy of nested 4-dimensional simplexes, that we call *pentatopes*. We call the resulting hierarchy a *Hierarchy of Pentatopes (HP)*. A hierarchy of pentatopes can be used as the domain decomposition for a four-dimensional scalar field. A major issue with any multi-resolution model based on a nested mesh is the topological consistency of the adaptive domain decomposition defined by the mesh, since inconsistencies may produce discontinuities in the corresponding approximation of the scalar field, and thus in the isosurfaces. Consistency must be maintained by splitting all pentatopes which share an edge at the same time. This is achieved through an efficient technique based on computing face-neighbors of a pentatope. We pro-

pose a neighbor finding algorithm which makes use of a pointer-less representation of a nested simplicial mesh. In such a representation, pentatopes are implicitly described as strings of bits, called *location codes*, corresponding to the path from the root of the hierarchy representing the nested simplicial mesh. The algorithm performs bitwise manipulation of the location code of the pentatopes to find neighbors in worst-case constant time.

Therefore, regardless of whether we are interested in the two-dimensional, three-dimensional, or four-dimensional case, efficient neighbor finding is an important operation if we want to analyze and display triangular, tetrahedral, or pentatopic meshes. In this thesis, we present algorithms for neighbor finding and show how to obtain a worst-case constant time implementation.

1.2 Related Work

A Quaternary Triangular Mesh (QTM) is a region quadtree composed of triangles. A particular QTM (based on an octahedron) is described by Dutton [18]. The scheme proposed by Dutton uses a regular subdivision of each triangle region into four subregions. Goodchild and Yang [30] simplify Dutton's cell labeling approach by using a different numbering of the triangles in order to obtain an addressing system that provides easy transformation to and from latitude and longitude. They also use an octahedron to model the sphere because the vertices can be aligned with the poles and equator. Fekete [23] uses an icosahedron to model the sphere since it gives a better initial approximation. Our methods differ from these methods

(e.g., [23, 30, 56]), which have a worst-case execution time proportional to the maximum level of decomposition. We achieve constant-time execution 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 [65, 67] for quadtree square meshes (which work for both pointer-based and linear quadtrees) can be adapted to deal with quadtree triangle meshes. This results in worst-case constant time algorithms for finding neighbors of equal size.

Hierarchical triangle meshes based on recursive triangle bisection have been extensively used for view-dependent terrain rendering (see, for instance, [16, 22, 48, 49, 57]). Recently, Lindstrom and Pascucci [49] have designed and implemented a framework for performing out-of-core view-dependent rendering of large terrain surfaces based on hierarchical meshes.

Evans et al. [22] use a hierarchy of right triangles to decompose a two-dimensional surface which is given as an array of elevation values. Coordinates are not explicitly stored since they can be calculated from the label (or location code) of the triangle. This is only possible because a regular decomposition rule is used. Children are formed by bisecting the parent triangle. A single bit of 0 or 1 indicates which child was chosen at each level in the decomposition. Since a pointer-based binary tree structure would be inefficient in its use of space, they use an array where the label of a node determines the node's location in the array. To ensure that nodes of different depths have different location codes, they prepend a 1 to the node label. In general, there is a one-to-one mapping between sequential integers and location

codes with depth information. Evans et al. choose to work with a continuous sequence of integer values instead of using depth explicitly. They find neighbors using recurrence relations and recursive algorithms which run in time proportional to the length of the location code (i.e., proportional to the depth of the triangle), and also give faster neighbor calculation code which uses a relatively small number of arithmetic and bitwise logical operations to find the location codes of neighbors in constant time. They also store three extra bits with each triangle to indicate the size of the neighboring triangles in each of the three possible directions.

Hierarchical tetrahedral meshes have been studied in finite element analysis and in computer graphics for describing three-dimensional scalar fields when the field values are given at the vertices of a regular square grid in 3D space. Examples are tetrahedral meshes generated by the so-called *red/green tetrahedron refinement* technique (see, for instance, [33]), or hierarchical meshes formed by tetrahedral and octahedral elements [32].

A common way of generating hierarchical meshes consists of recursively bisecting tetrahedra on their longest edge (see, for instance, [31, 44, 55, 60, 79]). Such meshes have been introduced for domain decomposition in finite element analysis [34, 50, 62], and they have been applied in scientific visualization, for instance, to generate progressive volume models of ultrasound data [64], or for space/time-efficient progressive encoding of isosurfaces at a variable resolution [60]. A generalization and analysis of such meshes in arbitrary dimensions is presented in [58] in connection with adaptive mesh generation. Zhou et al. [79] proposed a representation for a hierarchical tetrahedral mesh as a full binary forest, stored as an array.

A similar data structure has been used by Gerstner and Rumpf [28] for extracting isosurfaces at different levels of detail. An indexing scheme for out-of-core encoding and traversal has been proposed in [59].

An important issue when using hierarchical tetrahedral meshes is that if the domain is adaptively refined, the field associated with the extracted mesh (and, thus, the resulting isosurfaces) may present discontinuities in areas of transition. One way of ensuring continuity is through error saturation [28, 79], thus implicitly forcing all parents to be split before their descendants (see also [49] for an effective saturation technique for terrain data). In our approach, the continuity of the field is ensured by efficiently extracting meshes without cracks through a neighbor finding technique. Hebert [34] computes parents, children, and neighbors in a hierarchical tetrahedral mesh in a symbolic way, but finding neighbors still takes time proportional to the depth in the hierarchy.

In [31] an algorithm for interactively extracting and rendering isosurfaces of large volume data sets is presented, which extends the ROAMing algorithm introduced in [16]. The authors use a refinement scheme based on tetrahedron bisection, and propose a data structure for representing such meshes that directly encodes clusters of tetrahedra which must be split together to ensure consistency.

The problem of modeling and encoding time-varying scalar fields has been recently considered by some authors [5, 38, 39, 63, 76]. In [38], a loss-less single resolution compression technique is proposed for encoding very large and regularly-sampled 4D data. Atalay and Mount [2] extend the techniques of Hebert [34] for use in data sets with temporal components and other higher dimensional meshes, adding

both a pointerless representation and improved neighbor finding algorithms which work in arbitrary dimensions. In [39], the problem of tracking and visualizing local features from a time-varying volumetric data set is considered, based on extracting time-varying isosurfaces and interval volumes using isosurfaces in higher dimensions.

Algorithms for isosurface extraction from 4D scalar fields have also been developed. Extensions of the marching cube algorithm to 4D have been proposed [5, 63], which differ in the number of cases counted for the 4-cube, that is, 272 [63], and 222 [5]. In order to locate cells which are actually intersected by an isosurface, spatial and temporal coherence can be used (see, for instance, [72, 77]). Also, in [6], an algorithm for constructing the isosurfaces in any dimensions from a set of scalar values given at the vertices of a regular grid of hypercubes is proposed. Weigle and Banks [75] have designed a recursive algorithm for isosurface extraction from four-dimensional simplicial complexes, counting 5 possible different cases for a 4-simplex. The algorithm has been applied in [76] for visualizing unsteady 3D scalar fields.

In [5, 6], applications of four-dimensional scalar fields to extracting time-varying isosurfaces, interval volumes in 3D space, and morphing of isosurfaces are discussed. An interval volume is the set of points in a scalar field enclosed between two isosurfaces defined by two different isovalues [53]. Algorithms for computing an interval volume between two isosurfaces of a 3D scalar field have been presented in [26, 39, 51, 53]. In [7], interval volumes are used to segment a volume data set, and several new techniques for directly rendering the 3D field based on interval volumes are presented.

1.3 Outline of Thesis

In Chapter 2, we describe our two-dimensional structure and present algorithms for navigating within the structure in constant-time. Several top-level polyhedra are discussed, giving multiple options when modeling spherical data.

In Chapter 3, we describe our three-dimensional structure and present algorithms for navigating within the structure in constant-time. This chapter includes techniques for computing clusters along with three algorithms for performing selective refinement on an HT. Experimental results on the HT are also given.

In Chapter 4, we describe our four-dimensional structure and present algorithms for navigating within the structure in constant-time. This includes a discussion on clusters of pentatopes along with a depth-first algorithm for selective refinement on an HP. Experimental results on the HP are also given.

The conclusions can be found in Chapter 5.

Chapter 2

Two-Dimensional Triangle Quadtrees

2.1 Tree Node Labeling

We initially consider a general navigation problem where the underlying surface is a sphere represented by a collection of triangle meshes. In particular, we assume the the sphere is approximated by an icosahedron whose faces are each represented by a triangle mesh.

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., [78]). 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 correspond-

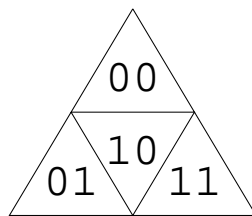
ing to the depth and path array, respectively. As these location codes determine a triangle rather than just a point, the terms *location code* and *triangle code* are used interchangeably. We also often use the term *code* to refer to the path array. Moreover, since we decompose each triangle into four 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 1a):

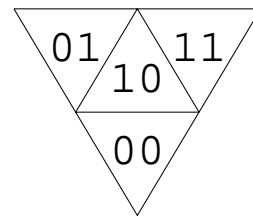
Top triangle:	00
Bottomleft triangle:	01
Center triangle:	10
Bottomright triangle:	11

Tip-down triangles use the following bit patterns for children (see Figure 1b):

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



(a)



(b)

Figure 1: Two possible orientations for a triangle: (a) tip-up, and (b) tip-down.

Our node labeling scheme is almost the same as that proposed by Goodchild and Yang [30]. 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 triangle. As we will see in Section 2.4, 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., [18, 23]) which are based on a “floating” labeling scheme (see [47] 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 2 for an example of a tree which is encoded using this node labeling method. This is in contrast with other methods (e.g., [18, 23, 56]) 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).

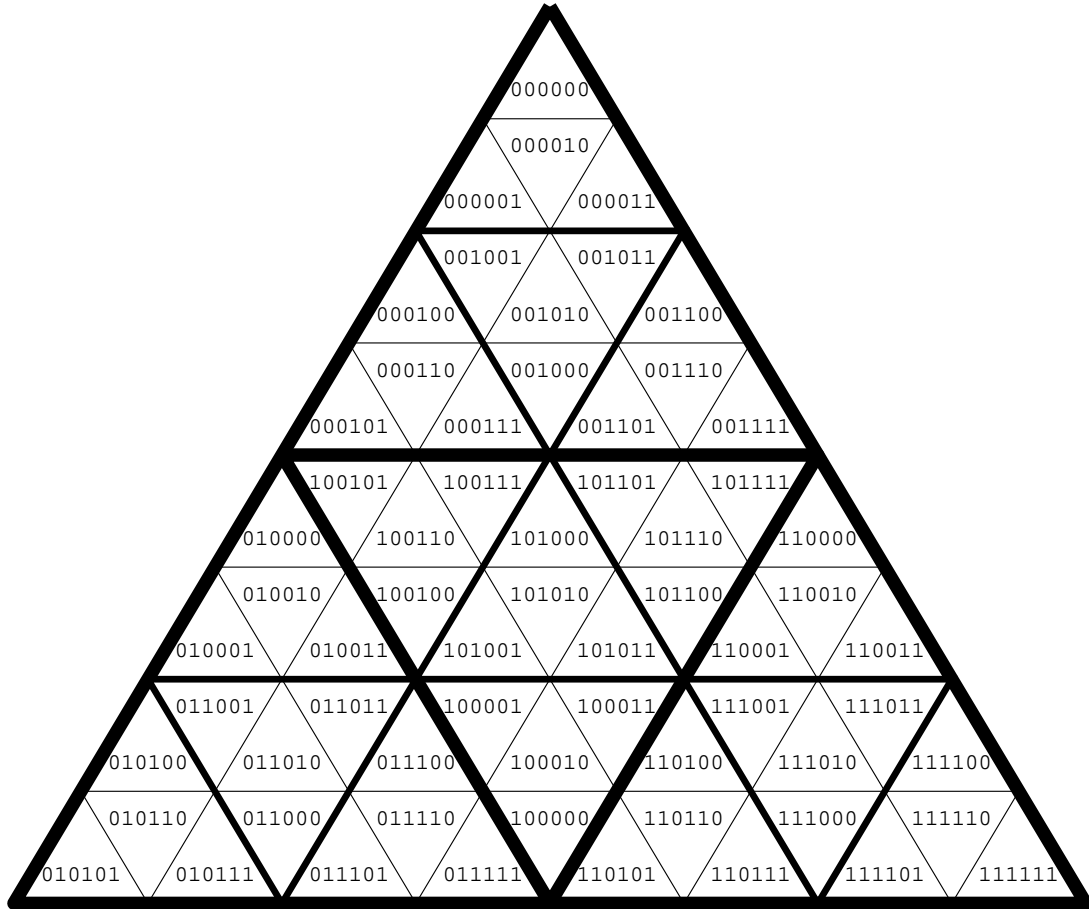


Figure 2: Labeling of a tree which is three levels deep.

2.2 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 [30] which is based on the approach of Samet [65, 67]. We present it here as it is the basis of our extension to the entire sphere in Section 2.3 as well as our constant-time algorithm in Section 2.4 (see also [46]). 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 which 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.

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 . Section 2.2.1 discusses step one of our algorithm. This involves locating the nearest common ancestor. In this step we just decide where in the path array corresponding to the location code (i.e., at what depth) to start step two. Section 2.2.2 describes step two of our algorithm. This involves changing the bit pattern of the location code to ensure that the child identified in the code for the nearest common ancestor (determined in step one) will contain the neighbor that we want. Section 2.2.3 explains step three of our algorithm. This involves updating the rest of the path to the neighbor

in the location code (from the depth in step two down to the input node's depth in the tree). Section 2.2.4 describes how to combine steps one, two and three into one routine which completes the whole task of neighbor finding. Observe that in an actual implementation the steps would be combined into one procedure. We have presented the process using this approach because it will be useful when we describe the extension of the algorithms to deal with transitions between different faces of the icosahedron in Section 2.3.

2.2.1 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 [65, 67] 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 (its parent contains a node that is adjacent to and to the right of p). If we look at Figure 1a, then we see that this is true for children 01 and 10. Also, in Figure 1b, 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, then 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 which does not equal 01 or 10, so we continue upward. EXCODE[3]=10 which means that we stop here. This sets us up for step two, described in Section 2.2.2. 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 Figure 3 (it is similar to the stop component in the conversion table used in [30]). 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.

```
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. */  
begin  
  value path_array CODE;  
  value integer NEIGHBOR_DIR;  
  reference integer CHILD_TYPE;
```

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

Figure 3: STOPTAB(Neighbor_Direction,Child_Type) relation indicating when to cease the search for the nearest common ancestor in step 1 of the neighbor finding algorithm.

```

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

```

2.2.2 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. 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 , 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 were 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 2.2.1. 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 1a and 1b 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 Figure 4 (it is similar to the new address component in the conversion table used in [30]). 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.

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

Figure 4: NEXTTAB(Neighbor_Direction,Child_Type) indicating the child type of the neighboring child of the nearest common ancestor.

```

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
which 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 Figure 4;  
    CODE[DEPTH] ← NEXTTAB[NEIGHBOR_DIR][CHILD_TYPE];  
end;
```

2.2.3 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 2.1) 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 1a and 1b are vertical reflections of each other).

For example, let's consider the location code 010010110000. This is EXCODE from Section 2.2.1. 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.

```
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 Figure 4;
  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 ;

```

2.2.4 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 2.3. Thus, the following routine is sufficient for finding any neighbor of equal size within one triangle quadtree.

```

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.

2.3 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 5). 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

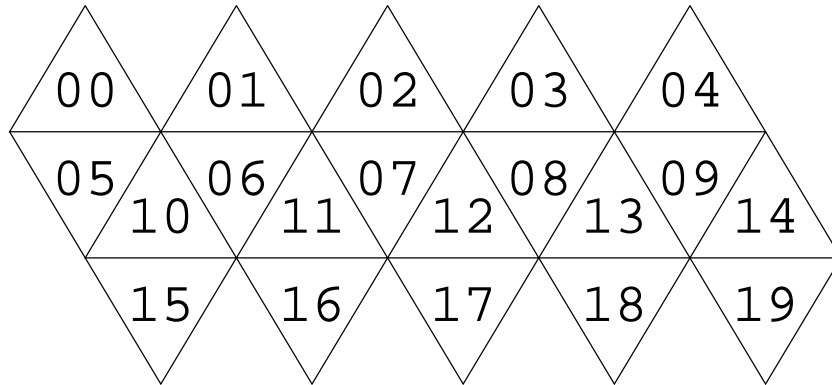


Figure 5: Example showing the top-level triangle faces of an icosahedron corresponding to the surface of the Earth.

code because we are at `CODE[0]`), we stop looking for the nearest common ancestor. Obviously, the entire sphere contains every 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`. Also, note that since Figure 5 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`. Figure 6 traces the execution of procedure `EXT_STEP_ONE` for this neighbor. Notice that in this case, the nearest common ancestor is the entire sphere.

```
procedure EXT_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. */
begin
  value path_array CODE;
```

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

Figure 6: Execution trace of procedure EXT_STEP_ONE for the left neighbor of 000010010001010001.

```

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

```

Step two is similar to the one described in Section 2.2.2 and is encoded by procedure EXT_STEP_TWO. The only modification from procedure STEP_TWO is the use of a different relation NEXTTOP (Figure 7) to indicate how to update CODE[0]. It summarizes the actions for all possible neighbors from Figure 5 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 Figure 7 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.

```

procedure EXT_STEP_TWO(CODE,NEIGHBOR_DIR,CHILD_TYPE,DEPTH);
/* Obtain the child type of the neighboring child of the nearest common ancestor

```

Child Type	Neighbor Direction		
	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

Figure 7: NEXTTOP(Neighbor_Direction,Child_Type) indicating neighbors for the triangles corresponding to the faces of the icosahedron.

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. */

```

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 Figure 4;
  preload integer array NEXTTOP[0:2][0:19] with Figure 7;
  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 Figure 8 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 Figure 4 in Section 2.2.3. 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 2.2.3 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 Figure 4 since no special treatment is required for the vertical case.

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

Figure 8: 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.

As an example, let’s continue to consider the location code with path array 000010010001010001 which was previously labeled as EXCODE2. The 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 will update in step three. Once again, we want to find the left neighbor. Using Figure 8, EXCODE2[1] (01) be-

comes 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.

```

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 Figure 4;
  preload integer array REFLTOP[0:2][0:3] with Figure 8;
  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;
    end;
  end;

```

The procedure for finding the neighbor which combines the three steps (i.e., FIND_NEIGHBOR given in Section 2.2.4) 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.

```
procedure EXT_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);  
  EXT_STEP_ONE(CODE(P),NEIGHBOR_DIR,CHILD_TYPE,DEPTH);  
  EXT_STEP_TWO(CODE(P),NEIGHBOR_DIR,CHILD_TYPE,DEPTH);  
  EXT_STEP_THREE(CODE(P),NEIGHBOR_DIR,CHILD_TYPE,LEV(P),DEPTH);  
end;
```

2.4 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 2.2 and 2.3 by an arithmetic operation that takes constant time instead of time proportional to 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 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 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 [70] for square quadtrees implemented us-

ing 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 2.5.

2.4.1 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-left-most 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* [52, 67]. 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 [70] 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 1s, performing the addition, and then restoring the y bits to their original values. This technique is shown in the procedure `SCHRACK_RIGHT` given below.

```

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←AND(CODE(P),ODDBITMASK);
  /* Load the y bit positions with 1s */
  CODE(P)←OR(CODE(P),ODDBITMASK);
  /* Add one (move right) */
  CODE(P)←CODE(P)+1;
  /* Clear the y bit positions */
  CODE(P)←AND(CODE(P),EVENBITMASK);
  /* Restore the original y bits */
  CODE(P)←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 `SHRACK_LEFT`, `SHRACK_UP`, and `SHRACK_DOWN`, not given here (see [47]), use a similar technique to `SHRACK_RIGHT` to calculate the left, up, and down neighbors of equal size. In particular, `SHRACK_LEFT` differs from `SHRACK_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 `SHRACK_DOWN` and `SHRACK_UP`, and procedures `SHRACK_RIGHT` and `SHRACK_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 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.

2.4.2 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 9). 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 00s with 11s 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 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 00s with 11, once the addition has taken place, any 00s that became 00 (i.e., were affected by the addition) must be set to their proper value which is 01, while all 00s that remained 11 (i.e., were unaffected by the addition) must be reset to their original value which is 00.

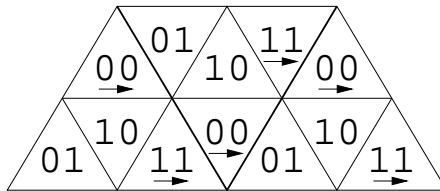


Figure 9: Examples of rightward transitions that generate a carry (denoted by a rightward pointing arrow) as the neighboring triangles are not siblings.

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:

1. to identify the bit positions which 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 *idmask*s 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 *idmask*s are formed by invoking a procedure `MAKE_IDMASK(INPUT, AB, XY)` which 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_ABIDXY¹.

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, Figure 10 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, ?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

¹Procedure 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.

leftward and vertical transitions, as well as transitions between neighboring triangles that are in different base triangles of the icosahedron.

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

Figure 10: The 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'.

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 Figure 11 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 Figure 11), 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.

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

Figure 11: Example showing the steps in the generation of idmask 00ID11 for some input values.

```

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 9). 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 00s in the path array of the input and to replace them with 11s 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. 11s not affected by the addition (which were originally 00s) must be changed back to 00, and
2. 11s which were affected by the addition and thus became 00s (again, only the ones which were originally 00s) 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 which 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 which were generated by an addition or a propagation of a carry (they are set to 01).

The first task is performed by taking the XOR of the idmask with the result of the addition, thereby creating a bit pattern which 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 11s at positions in the path array of the input which originally contained 00 (which is desired) and resetting to 11 all 00s at positions in the path array of the input which originally contained 00.

Once the first task has been completed, perform the second task. In particular, all 11s which were affected by the addition and thus became 00s (again, only the ones which were originally 00s) must be changed to 01 (because 00 plus one is 01). This is achieved by constructing a mask which has a 11 at every pair of positions in the path array of the input which 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 which were part of the original 00 pair with a 01. Taking the AND of the resulting mask with t yields the desired result.

```
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);
/* Fixup 00s that got hit with a carry */
CODE(P)←AND(CODE(P),OR(COMPLEMENT(00ID11),EVENBITMASK));
end;

```

Figure 12 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 Figure 12 (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, 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 Figure 12 (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).

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

Figure 12: The effect of procedure `CONSTANT_RIGHT` on different bit pattern pair values depending on whether or not there is an incoming carry from the right.

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 13a.

2.4.3 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

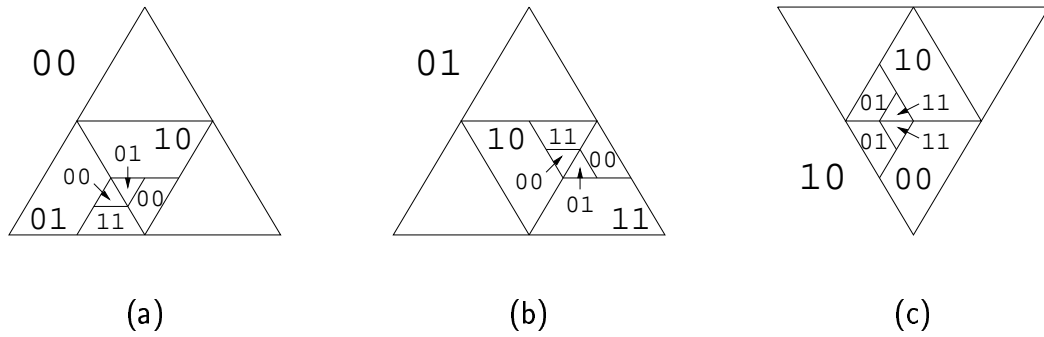


Figure 13: Examples showing how to find neighbors of equal size: (a) right neighbor of 00011100, (b) left neighbor of 01110001, (c) vertical neighbor of 10100111.

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 siblings².

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 14). 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 from a 01 child

²We could also implement the subtraction by the addition of -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.

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 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.

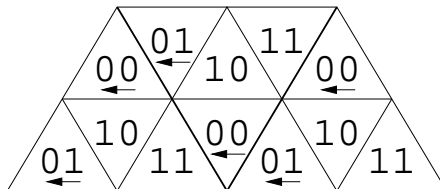


Figure 14: Examples of leftward transitions that generate a borrow (denoted by a leftward pointing arrow) as the neighboring triangles are not siblings.

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 easily done 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.

```

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 which 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 which were generated by a subtraction or a propagation of a borrow (they are set to 00).

The first task is performed by taking the XOR of the idmask with the result of the subtraction thereby creating a bit pattern which 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 which originally contained 01 (which is desired) and resetting to 10 all 11s at positions in the path array of the input which originally contained 01.

Once the first task has been completed, perform the second task. In particular, all 00s which were affected by the subtraction and thus became 11s (again, only the ones which were originally 01s) must be changed to 00 (because 01 minus one is 00). This is achieved by constructing a mask which has a 11 at every pair of positions in the path array of the input which 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.

```

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);
/* Fixup 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 `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 13b.

2.4.4 Vertical Transitions

In this Section, we consider a transition from a triangle to its vertical neighbor. This is a very simple transition as 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 1a and 1b that with exception of the path array component corresponding to the containing sibling triangles, 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 2.2.

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., right-most 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 is the complement of *i*. Idmask ?0ID10 is formed by a call to MAKE_IDMASK_?0ID10 given below.

```

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.

`CONSTANT_VERTICAL` first computes `idmask ?OID10` by invoking procedure `MAKE_IDMASK_?OID10`. Next, it creates a new mask m from `?OID10` which is zero at all bit positions with the exception of the rightmost 10. This is achieved by taking the `COMPLEMENT` of `?OID10`. The result is a mask n which contains 11 in all bit-pair positions to the right of the rightmost 10 of `?OID10` 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 `?OID10`. This works because all items to the left of the rightmost 10 in p are the complements of the corresponding items in `?OID10` 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 Figure 4 in Section 2.2.2.

```
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 ?OID10,MASK;
  ?OID10←MAKE_IDMASK(P,?0,10);
  MASK←COMPLEMENT(?OID10);
  /* Use carry to find what to update */
  MASK←MASK+1;
  /* Clear out everything but carry */
  MASK←AND(MASK,?OID10);
  /* Update the path array */
```

```

    CODE(P) ← XOR(CODE(P), MASK);
end;

```

As an example of the action of procedure `CONSTANT_VERTICAL`, 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. `?OID10` is 10100000 since both `VCODE[1]` and `VCODE[2]` have value `?0`. The operation `COMPLEMENT(?OID10)` yields 01011111 which is stored in variable `MASK`. Adding one to `MASK` yields 01100000. Applying `AND(MASK, ?OID10)` changes `MASK` to 00100000. The final `XOR` of `MASK` with `VCODE` changes `VCODE` to 10000111. This example is illustrated in Figure 13c.

2.4.5 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 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 Figure 7 which was described in Section 2.3.

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 5 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 2.3, we solved this problem by making use of Figure 8. 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 what was done for the 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 which 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).

```

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.

Observe that procedure CONSTANT_REFLECTION works for both left and right neighbors. 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.

```

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 Figure 7. 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 was 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 [47]), 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 Figure 7 if overflow occurs.

```
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 Figure 7;
  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;
  end;
```

```
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 Figure 7;
  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]
  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;

```

2.5 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 2.5.1 describes the modifications to the algorithms for the icosahedron needed for the octahedron while Section 2.5.2 deals with the tetrahedron.

2.5.1 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 15). Our numbering scheme has the property that triangles 0 to 3 are tip-up, and 4 to 7 are tip-down.

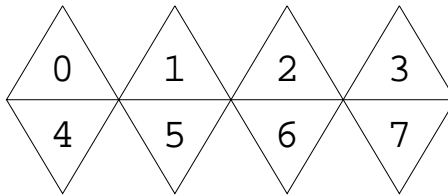


Figure 15: Example showing the top-level triangle faces of an octahedron.

The only modification with respect to step two of the algorithm in Section 2.2 is the use of a different relation `NEXTOCT` (Figure 16) to indicate how to update `CODE[0]`. It summarizes the actions for all possible neighbors from Figure 15 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 which is analogous to those given in Section 2.4.5 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

Child Type	Neighbor Direction		
	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

Figure 16: NEXTOCT(Neighbor_Direction,Child_Type) indicating neighbors of the triangles corresponding to the faces of the octahedron.

are done. We update the root value using Figure 16. Also, we throw away the result of `CONSTANT_RIGHT` (only the overflow condition was 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 [47]), 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 [47]), just needs to call procedure `CONSTANT_VERTICAL`, and then update the root value using Figure 16 if overflow occurs. It is identical to procedure `EXT_CONSTANT_VERTICAL` once we replace table `NEXTTOP` (Figure 7) by `NEXTOCT` (Figure 16).

```

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 NEXTTOCT[0:2][0:7] with Figure 16;
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
    /* 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]←NEXTTOCT[‘RIGHT’][CODE(P)[0]];
  end;
  /* Set CODE(P) to the new path array value */
  CODE(P)←CODE(NEWP);
end;

```

2.5.2 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 1b (see Figure 17).

The only modification with respect to step two of the algorithm in Section 2.2 is the use of a different relation NEXTTET (Figure 18) to indicate how to update

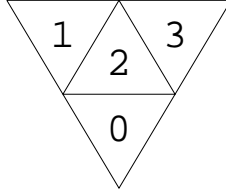


Figure 17: Example showing the top-level triangle faces of a tetrahedron.

CODE[0]. It summarizes the actions for all possible neighbors from Figure 17 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.

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

Figure 18: NEXTTET(Neighbor_Direction,Child_Type) indicating neighbors for the triangles corresponding to the faces of the tetrahedron.

If we examine the triangle adjacencies for the tetrahedron (see Figure 19), then we notice that some of the transitions result in a “flipped” result. Compensating for this “flipped” result isn’t a significant problem since 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 which is analogous to those given in Section 2.4.5 for the icosahedron in the sense that they work regardless of whether the neighbors are on the

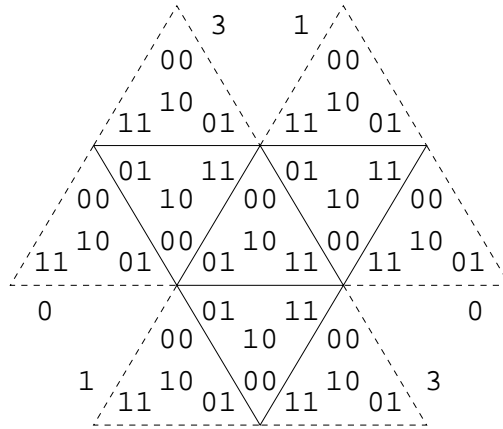


Figure 19: Example showing the triangle adjacencies of the tetrahedron.

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 Figure 18. If the current child of the root corresponds to the faces of the tetrahedron labeled 0 or 3, then 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 [47]), 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 [47]), 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.

```

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 Figure 18;
  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;
end;

```

2.6 Finding Neighbors of Greater or Equal Size

The algorithms in Sections 2.2–2.5 assumed 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 1.1, 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 which 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 what 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 [9]. In fact, this is how the list is usually implemented (e.g., [1]). In this case, the search takes time logarithmic in the size of L , which is the total number of nodes in the triangle hierarchy.

2.7 Comparison with Method based on Icosahedron

Our coding scheme is different from that proposed by Fekete [23]. In particular, our scheme was chosen to be simpler and more regular (and also stable under

resolution doubling), thereby making it easier to traverse the tree, locate children, locate siblings, etc. In order to see this, we now explain Fekete's method.

Fekete [23] proposed a location code whose path array elements are labeled according to the following labeling convention, as illustrated in Figure 20. Label the vertices of the parent A, B, and C. Label the midpoints as A' (between B and C), B' (between A and C), and C' (between A and B). The children can be determined by the vertices and midpoints of the parent.

Children are labeled as follows:

Triangle	A B'C'	1
Triangle	A'B C'	2
Triangle	A'B'C	3
Triangle	A'B'C'	4

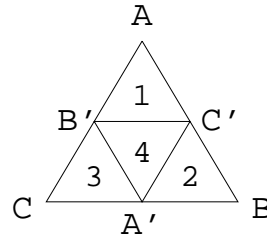


Figure 20: Fekete's labeling scheme.

See Figure 21 for an example of a tree encoded using this scheme. Notice how the positions of the labels change at each level in the tree. The labeling scheme was designed so that the label sequences for adjacent triangles will differ in exactly one symbol. Using the terminology of Section 2.2.1, this symbol is in the position of the child of the nearest common ancestor of the two adjacent triangles. This makes the determination of triangle adjacency somewhat simpler, but doesn't guarantee that only adjacent triangles will differ in exactly one symbol. If direction is important, this labeling scheme must keep up with the current orientation since the relative

orientations of the triangles change at each level of subdivision. This is particularly important if we want to know which neighbor we are finding. Also, orientation for this labeling scheme is not as simple as tip-up or tip-down, since two different ‘tip-up’ triangles may have different orientations. A triangle’s first vertex (labeled A) may be its top-most point, while another triangle’s first vertex may be its bottom-left-most point.

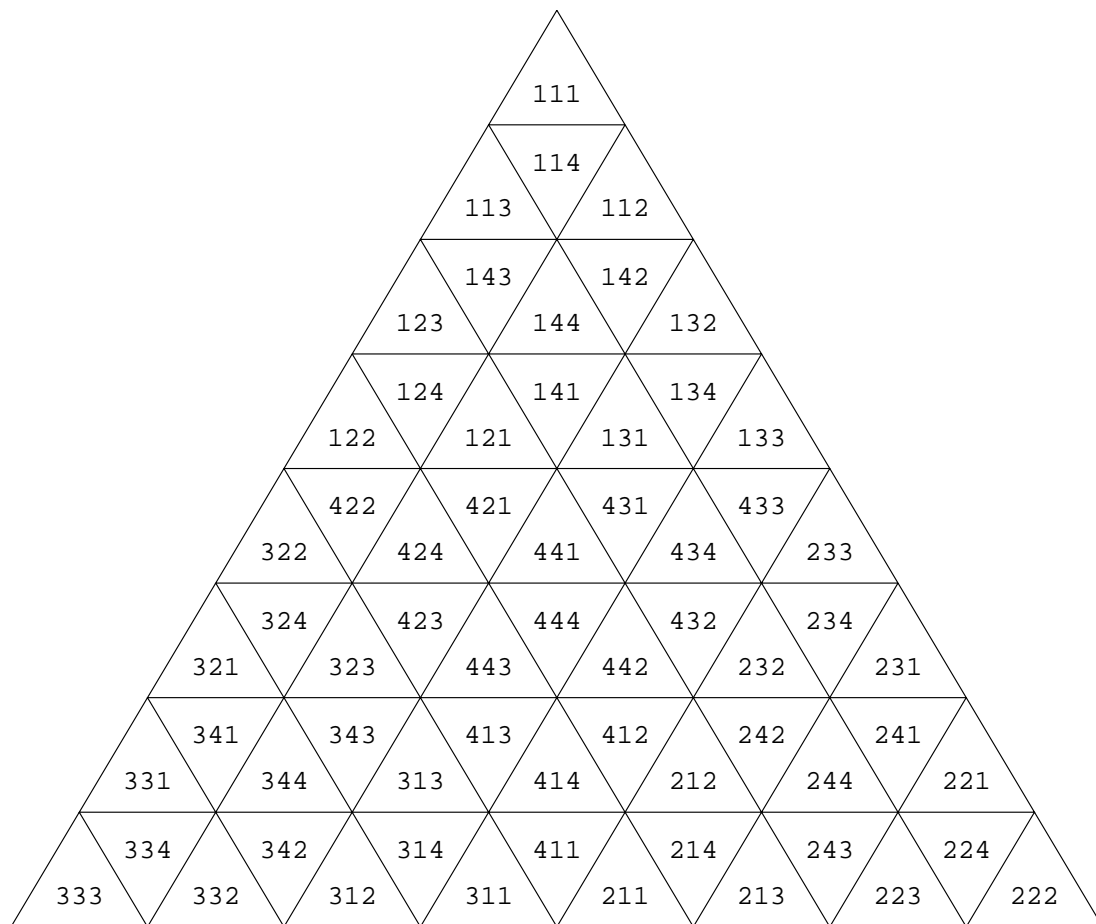


Figure 21: Tree using Fekete's scheme.

Using this labeling scheme, where triangles are determined by their three vertices and children are determined relative to their parents, we can calculate the locations of a child's vertices by computing the midpoints of specific line segments

(between the parent's vertices. The midpoint between two vertices `pt1` and `pt2` is denoted by `MIDPOINT(pt1,pt2)`).

```
procedure FEKETE_GET_CHILD_1 (PA,PB,PC,CA,CB,CC);
/* Return in CA, CB, and CC the locations of the vertices of the 1 child of the parent
   having vertices at locations PA, PB, and PC. */
begin
  value point PA,PB,PC;      /* Parent vertices */
  reference point CA,CB,CC;  /* Child vertices */
  CA←PA;
  CB←MIDPOINT(PA,PC);
  CC←MIDPOINT(PA,PB);
end;
```

```
procedure FEKETE_GET_CHILD_2 (PA,PB,PC,CA,CB,CC);
/* Return in CA, CB, and CC the locations of the vertices of the 2 child of the parent
   having vertices at locations PA, PB, and PC. */
begin
  value point PA,PB,PC;      /* Parent vertices */
  reference point CA,CB,CC;  /* Child vertices */
  CA←MIDPOINT(PB,PC);
  CB←PB;
  CC←MIDPOINT(PA,PB);
end;
```

```
procedure FEKETE_GET_CHILD_3 (PA,PB,PC,CA,CB,CC);
/* Return in CA, CB, and CC the locations of the vertices of the 3 child of the parent
   having vertices at locations PA, PB, and PC. */
begin
  value point PA,PB,PC;      /* Parent vertices */
  reference point CA,CB,CC;  /* Child vertices */
  CA←MIDPOINT(PB,PC);
  CB←MIDPOINT(PA,PC);
  CC←PC;
end;
```

```
procedure FEKETE_GET_CHILD_4 (PA,PB,PC,CA,CB,CC);
/* Return in CA, CB, and CC the locations of the vertices of the 4 child of the parent
   having vertices at locations PA, PB, and PC. */
begin
  value point PA,PB,PC;      /* Parent vertices */
  reference point CA,CB,CC;  /* Child vertices */
  CA←MIDPOINT(PB,PC);
```

```

    CB←MIDPOINT(PA,PC);
    CC←MIDPOINT(PA,PB);
end;

```

Fekete [23] prefers to perform neighbor finding by walking through the path array of the location code from top to bottom (i.e., from left to right, or equivalently from large node to small node), building a potential path array (termed a *path* from now on) to the neighbor. This path consists of directions which are expressed as the labels of the children. Upon reaching the end of the path array of the input location code, the potential path is actually the complete path to the neighbor. In order to determine the direction to the neighbor (e.g., to make sure that we get the left neighbor), care must be taken to keep track of the global directions. This amounts to six different orientations on account of the six possible placements of the labels 1, 2, and 3 within each triangle. A state table is needed to keep track of the changes at each level.

Fekete makes use of the concept of an “open direction”, where direction d is said to be open if both the d -neighbor s of triangle t and t itself are in the same base triangle of the icosahedron. In particular, Fekete looks for “open directions” as each element of the path is examined. An open direction d with respect to triangle t is said to be “new” if the d -neighbor s of t is a brother of t (i.e., s and t have the same parent). The four possible configurations leading to the discovery of a new open direction are shown in Figure 22. Old open directions are directions in which neighbors were already discovered while processing previous elements of the path (corresponding to larger nodes or triangles). Notice that all directions are always

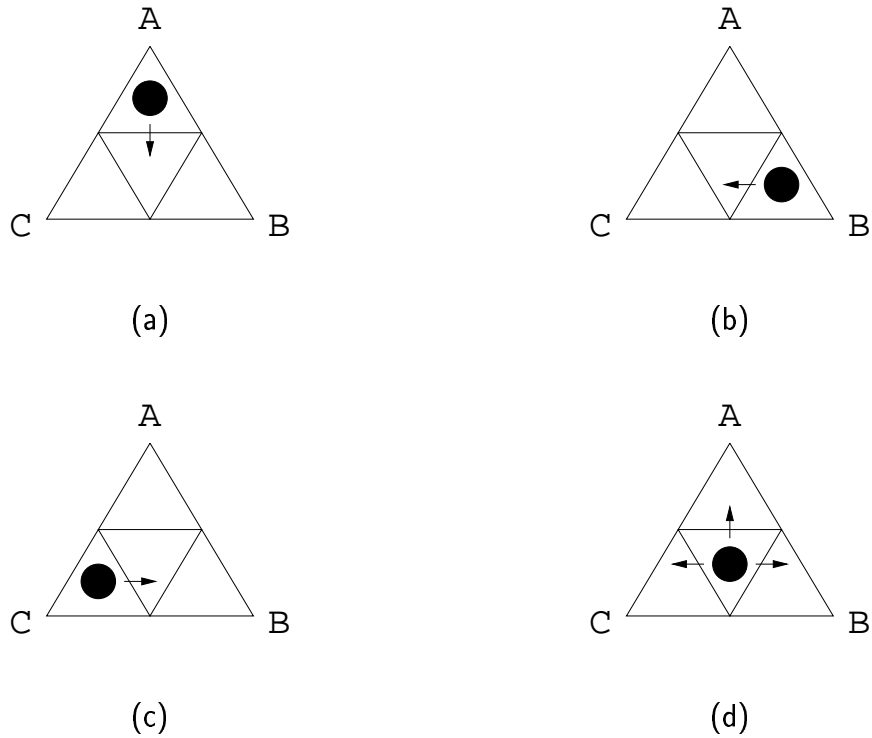


Figure 22: (a) Open direction A; (b) open direction B; (c) open direction C; (d) all three directions are open.

specified relative to the entire region represented by the tree corresponding to one of the 20 base triangles of the icosahedron, and not relative to the individual triangles at each level in the tree since these directions are constantly changing.

Fekete's algorithm treats the path array as a string of symbols, where the left-most symbol corresponds to the root of a base triangle of the icosahedron containing the triangle whose neighbor is being sought. This string is processed from left to right so that the right-most symbol corresponds to the subdivision leading to the smallest possible triangle. At any given level, there is a relationship between the parent node and the child node corresponding to the current symbol in the path string. If the current symbol is a 1, 2, or 3, this relationship can be seen in Figure 22a–c, based

on where the child (marked by the large dot) is located relative to the parent. In these cases, exactly one new open direction (marked by the arrow in the Figure) is discovered at this level of the algorithm. Whenever this occurs, the neighbor in the new open direction can be determined by copying the substring of the original input path string up to the current symbol (locating the parent) and appending a 4 because the brother is 4 in all three cases. If an open direction is found to be “new” at this level, it is considered a new open direction even if it was encountered earlier in the algorithm (which would technically also make it an old open direction). We call this *Rule 1*.

Any open directions from previous levels of the algorithm become old open directions. An equal-size neighbor will always exist in these directions. None of the situations shown in Figure 22 apply in this case because the node and its corresponding neighbors are not brothers. Whenever this occurs, the neighbors in old open directions can be determined by appending the current symbol to the potential neighbor string from the previous level. We call this *Rule 2*.

If the current symbol is a 4, then the relationship between the parent node and the child node can be seen in Figure 22d. In this case, all three directions are considered new open directions (marked by the arrows in the Figure) because all three were just discovered at this level of the algorithm. We call this *Rule 3*.

These rules are essential to Fekete’s algorithm. They are used extensively in the examples which follow. A rules summary can be found in Figure 23. All open directions encountered are kept on a list so that border cases can be identified. Any missing directions reveal which borders are adjacent to the triangle. Interestingly,

Fekete’s algorithm can be summarized as processing the path array of the input looking for the rightmost (and hence final) position where Rule 1 (i.e., the rightmost open direction) or Rule 3 can be applied. Using the terminology of Section 2.2.1, this is the position of the child of the nearest common ancestor of the two adjacent triangles. It is the only one which is changed.

Rule	Condition	Action
1	The current symbol is not 4 and the direction being considered matches the <i>new</i> open direction.	Copy the substring of the <i>original</i> path string up to the current symbol and append a 4.
2	The current symbol is not 4 and the direction being considered matches an <i>old</i> open direction.	Take the <i>current</i> path string of the potential neighbor and append the current symbol.
3	The current symbol is 4 (thus there are three new open directions).	Copy the substring of the <i>original</i> path string up to the current symbol and append a 1, 2, or 3 as appropriate for the direction being considered.

Figure 23: List of rules used in Fekete’s algorithm.

Earlier we mentioned that Fekete needs to keep track of global directions since the individual triangles at each level can have many orientations. This affects the determination of open directions as well as the appended symbols from Rule 3. If we look again at Figure 20 and the labeling of children, the following observations can be made. Our initial global directions are A, corresponding to vertical of center; B, corresponding to right of center; and C, corresponding to left of center. We’ll abbreviate this as A=V, B=R, and C=L. Notice that for parent triangle ABC, child 1 is AC’B’ (in clockwise order). Since B and C switch positions, the directions associated with them should be switched so that now A=V, B=L, and C=R. Thus the children of

child 1 are oriented as follows. Child 14 is in the center because this child is always in the center. Child 11 is the vertical brother of 14 because the vertical orientation of 1 is the same as that of its parent. Child 12 is the left brother of 14, because the left to right directions of 1 are the opposite of those of the parent and 2 was on the right in the parent. Child 13 is the right brother of 14 again because the left to right directions are reversed and 3 was on the left in the parent. The other children have similar analyses. Notice that with respect to parent triangle ABC , child 2 is $C'BA'$. Since A and C switch positions, the directions associated with them should also be switched so that $A=V$, $B=R$, and $C=L$ would become $A=L$, $B=R$, and $C=V$. Child 3 is $B'A'C$, so that $A=V$, $B=R$, and $C=L$ would become $A=R$, $B=V$, and $C=L$. For child 4 every vertex changes. Of course, vertical stays vertical (whether top or bottom), so only the two marked L and R change. Thus $A=V$, $B=R$, and $C=L$ would become $A=V$, $B=L$, and $C=R$.

The global positions of the children are most useful in these algorithms. Child 4 is always in the center, so our three global directions are relative to child 4. In effect, we need to invert or reverse our previous associations. Also, since vertex A corresponds to child 1, B to 2, and C to 3, we use the appropriate numbers instead of the letters. We abbreviate vertical as V , left as L , and right as R . Let's suppose that our base triangle is oriented as in Figure 20 so that $V=1$, $L=3$, and $R=2$. Notice that we can also get this by reversing the equalities in $A=V$, $B=R$, and $C=L$ to get $V=A$, $L=C$, and $R=B$; then, substituting numbers for letters, we get $V=1$, $L=3$, and $R=2$. From the previous paragraph we know that child 1 switches B and C . Since B corresponds to 2 and C to 3, we switch 2 and 3 so triangle 1 has global status $V=1$, $L=2$, and $R=3$. If

we next look at child 2 of 1, then we know that we have to switch 1 and 3 for child 2, so that triangle 12 has global status $V=3$, $L=2$, and $R=1$. If we next look at child 3 of 12, we know that we have to switch 1 and 2 for child 3, so that triangle 123 has global status $V=3$, $L=1$, and $R=2$. Finally, if we look at child 4 of 123, we know from the previous paragraph that we need to switch L and R. Therefore, triangle 1234 has global status $V=3$, $L=2$, and $R=1$. The rules used for each of the four children are summarized in Figure 24.

Child Label	Action	Example
1	Swap 2 and 3	$V=3$ $L=2$ $R=1$ becomes $V=2$ $L=3$ $R=1$
2	Swap 1 and 3	$V=1$ $L=2$ $R=3$ becomes $V=3$ $L=2$ $R=1$
3	Swap 1 and 2	$V=3$ $L=2$ $R=1$ becomes $V=3$ $L=1$ $R=2$
4	Swap L and R	$V=1$ $L=2$ $R=3$ becomes $V=1$ $L=3$ $R=2$

Figure 24: Rules used to keep track of global status.

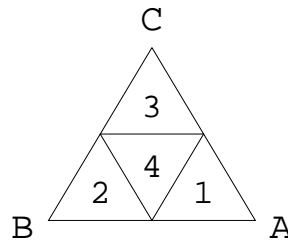


Figure 25: Layout for $V=3$, $L=2$, and $R=1$.

New open directions are easy to determine from the global status. Say, for example, that $V=3$, $L=2$, and $R=1$, as shown in Figure 25. If we are considering child 1, then 1 is to the right of 4 because $R=1$. Therefore, 4 is the only brother of 1 and 4 is to the left of 1, so left is the only new open direction for child 1. If we are considering child 2, then 2 is to the left of 4 because $L=2$. Therefore, 4 is the only brother of 2 and 4 is to the right of 2, so right is the only new open direction for child

2. Likewise, with child 3, 3 is vertical of 4 because $V=3$. Therefore, 4 is the only brother of 3, so vertical is the only new open direction for child 3. Remembering that child 4 always invokes Rule 3 means that when $V=3$, $L=2$, and $R=1$, the paths to the three brothers terminate with 3 for vertical, 2 for left, and 1 for right.

To gain a better understanding of the rules in Figure 23, consider the following example. Suppose we have a current path string of 1 and we are considering the next symbol which is 2 (i.e., triangle 12). Our current position can be found in Figure 21. If we look for an open direction with respect to 12, we see that Figure 22c applies because triangle 12 holds the ‘large dot’ position in this figure relative to triangle 1. If we are looking for a right neighbor, Rule 1 should be used because the direction being considered (right) matches the new open direction (the arrow in Figure 22c points to the right). Rule 1 indicates that the same-size neighbor is 14. This can be verified using Figure 21.

On the other hand, suppose we are looking for a vertical neighbor of 12 and we are currently looking at the symbol 2. We know from the previous level that vertical was an open direction and that the potential vertical neighbor was 4. This was the case because we were previously looking at the path string 1, in which case the vertical direction was open. At this point (i.e., when looking at the symbol 2), Rule 2 should be used because the direction being considered (vertical) doesn’t match the new open direction (right), but it does match an old open direction. Rule 2 indicates that we can get the same-size neighbor by appending the current symbol of 2 to the previous potential pathname string of 4. This gives a result of 42 as the vertical neighbor. This can be verified using Figure 21.

It is important to note that we cannot find a left neighbor for 12 because left is neither a new nor an old open direction. Notice that triangle 12 doesn't have a left neighbor within Figure 21. This is a border case which has to be dealt with differently.

Now, let us modify our example so that our current path string is 1 while the next symbol is 4 (i.e., triangle 14). Our current position can be found in Figure 21. If we look for an open direction which corresponds to this position, we see that Figure 22d applies because triangle 14 contains the 'large dot' position with respect to triangle 1. At this point, Rule 3 should be used regardless of which neighbor we are trying to find. Rule 3 indicates that the same size neighbors all start with 1. In fact, these neighbors are 11 for vertical, 12 for left, and 13 for right. This can be verified using Figure 21.

At this point, let us consider two complete examples. Let FKCODE1 be 123 and let FKCODE2 be 143. Fekete labels the global directions based on the orientation of the root triangle. For these examples, global direction A (i.e. the direction toward the edge across from vertex A) corresponds to the vertical neighbor, global direction B (i.e., the direction toward the edge across from vertex B) corresponds to the left neighbor, and global direction C (i.e., the direction toward the edge across from vertex C) corresponds to the right neighbor.

The results for our first example are found in Figure 26. Let us first examine the vertical neighbor case. Our initial global status is noted in the Figure with $V=1$, $L=3$, and $R=2$ (recall Figure 20). Fekete's algorithm starts by examining the first symbol. The first symbol in FKCODE1 is 1. This is not a 4 so we look for a new

Partial Path String	Potential Neighbor			Open Directions		Global		
	Vert	Left	Right	New	Old	V	L	R
1	4	-	-	Vert	-	1	3	2
12	42	-	14	Right	Vert	1	2	3
123	124	-	143	Vert	Vert+Right	3	2	1

Figure 26: Steps taken while finding neighbors of 123 using Fekete's method.

open direction. Since $V=1$, our only open global direction is vertical (A). A single new open direction is discovered and it is vertical (A), so we copy the substring of the original path string up to the current symbol (which is empty) and append a 4 (Rule 1). Substring (empty) append 4 yields 4 (see Figure 26).

The algorithm continues by examining the next symbol. We first change the global status to $V=1$, $L=2$, and $R=3$ because we have just finished processing child 1. The next symbol in FKCODE1 is 2. This is not a 4, so we look for a new open direction. Since $L=2$, our only open global direction is to the right (C as 4 is to the right of 2). As the new open direction is to the right (C) and not vertical (A), we just append the symbol 2 to the current string (Rule 2). String 4 append 2 yields 42 (see Figure 26).

Next, the algorithm examines the last symbol. We first change the global status to $V=3$, $L=2$, and $R=1$ because we have just finished processing child 2. The last symbol in FKCODE1 is 3. This is not a 4 so we look for a new open direction. Since $V=3$, our only open global direction is vertical (A as 4 is vertical of 3). A single new open direction is discovered and it is vertical (A), so we copy the substring of the original path string up to the current symbol (which is 12) and append a 4 (Rule 1). Substring 12 append 4 yields 124 (see Figure 26). Therefore, the vertical

neighbor of FKCODE1 is 124. This can be seen in Figure 21.

Now let us examine the left neighbor case. This is quite simple in that there is never a left neighbor. This can be easily seen by examining the result of finding the vertical neighbor, where we recall that we never encountered the left direction (B). In particular, the last entry in the 'Old Open Directions' column of Figure 26 doesn't include direction left (B), so this is actually a border case. Therefore, FKCODE1 doesn't have a left neighbor inside the triangle.

We conclude by examining the right neighbor case. From our analysis of the vertical neighbor case we saw that this case does not arise until after Fekete's algorithm processes the second symbol. At this point our global status is $V=1$, $L=2$, and $R=3$, corresponding to child 1. The second symbol in FKCODE1 is 2. This is not a 4 so we look for a new open direction. Since $L=2$, our only open global direction is to the right (C as 4 is to the right of 2). A single new open direction is discovered and it is to the right (C), so we copy the substring of the original path string up to the current symbol (which is 1) and append a 4 (Rule 1). Substring 1 append 4 yields 14 (see Figure 26).

The algorithm continues by examining the last symbol. We first change the global status to $V=3$, $L=2$, and $R=1$ because we have just finished processing child 2. The last symbol in FKCODE1 is 3. This is not a 4 so we look for a new open direction. Since $V=3$, our only open global direction is vertical (A as 4 is vertical of 3). As the new open direction is vertical (A) and not to the right (C), we just append the symbol 3 to the current string (Rule 2). String 14 append 3 yields 143 (see Figure 26). Therefore, the right neighbor of FKCODE1 is 143. This can be seen

in Figure 21.

Partial Path String	Potential Neighbor			Open Directions		Global		
	Vert	Left	Right	New	Old	V	L	R
1	4	-	-	Vert	-	1	3	2
14	11	12	13	All	Vert	1	2	3
143	113	123	144	Right	All	1	3	2

Figure 27: Steps taken while finding neighbors of 143 using Fekete's method.

The results for our second example are found in Figure 27. Let us first examine the vertical neighbor case. Our initial global status is noted in the figure as $V=1$, $L=3$, and $R=2$ (recall Figure 20). Fekete's algorithm starts by looking at the first symbol. The first symbol in `FKCODE2` is 1. This is not a 4 so we look for a new open direction. Since $V=1$, our only open global direction is vertical (**A**). A single new open direction is discovered and it is vertical (**A**), so we copy the substring of the original path string up to the current symbol (which is `empty`) and append a 4 (Rule 1). Substring (`empty`) append 4 yields 4 (see Figure 27).

The algorithm continues by examining the next symbol. We first change the global status to $V=1$, $L=2$, and $R=3$ because we have just finished processing child 1. The next symbol in `FKCODE2` is 4 which means that there are three new open directions. As all three directions are now open, we copy the substring of the original path string up to the current symbol (which is 1) and append a 1 for the vertical case (Rule 3) because $V=1$. Substring 1 append 1 yields 11 (see Figure 27).

Next, the algorithm examines the last symbol. We first change the global status to $V=1$, $L=3$, and $R=2$ because we have just finished processing child 4. The last symbol in `FKCODE2` is 3. This is not a 4 so we look for a new open direction.

Since $L=3$, our only open global direction is to the right (C as 4 is to the right of 3). As the new open direction is to the right (C) and not vertical (A), we just append the symbol 3 to the current string (Rule 2). String 11 append 3 yields 113 (see Figure 27). Therefore, the vertical neighbor of FKCODE2 is 113. This can be seen in Figure 21.

Now let us examine the left neighbor case. In our analysis of the vertical neighbor case, we saw that this situation does not arise until after Fekete's algorithm processes the second symbol. At this point our global status is $V=1$, $L=2$, and $R=3$, corresponding to child 1. The second symbol in FKCODE2 is 4, which means that there are three new open directions. As all three directions are now open, we copy the substring of the original path string up to the current symbol (which is 1) and append a 2 for the left neighbor case (Rule 3) because $L=2$. Substring 1 append 2 yields 12 (see Figure 27).

The algorithm continues by examining the last symbol. We first change the global status to $V=1$, $L=3$, and $R=2$ because we have just finished processing child 4. The last symbol in FKCODE2 is 3. This is not a 4 so we look for a new open direction. Since $L=3$, our only open global direction is to the right (C as 4 is to the right of 3). As the new open direction is to the right (C) and not to the left (B), we just append the symbol 3 to the current string (Rule 2). String 12 append 3 yields 123 (see Figure 27). Therefore, the left neighbor of FKCODE2 is 123. This can be seen in Figure 21.

We conclude by examining the right neighbor case. In our analysis of the vertical neighbor case, we saw that this situation does not arise until after Fekete's

algorithm processes the second symbol. At this point, our global status is $V=1$, $L=2$, and $R=3$, corresponding to child 1. The second symbol in `FKCODE2` is 4, which means that there are three new open directions. As all three directions are now open, we copy the substring of the original path string up to the current symbol (which is 1) and append a 3 for the right neighbor case (Rule 3) because $R=3$. Substring 1 append 3 yields 13 (see Figure 27).

Next, the algorithm examines the last symbol. We first change the global status to $V=1$, $L=3$, and $R=2$ because we have just finished processing child 4. The last symbol in `FKCODE2` is 3. This is not a 4, so we look for a new open direction. Since $L=3$, our only open global direction is to the right (C as 4 is to the right of 3). A single new open direction is discovered and it is to the right (C), so we copy the substring of the original path string up to the current symbol (which is 14) and append a 4 (Rule 1). Substring 14 append 4 yields 144 (see Figure 27). Therefore, the right neighbor of `FKCODE2` is 144. This can be seen in Figure 21.

If Fekete [23] had used a more traditional quadtree-like approach to neighbor finding (e.g., [67]), his technique would have been along the following lines. To find the nearest common ancestor (step one), he would somehow have to track all the changes in orientation so he could determine when he finds the proper ancestor. Next (step two), he would have to find the child of the nearest common ancestor which contained the neighbor (again making use of the orientation). Finally, when step three is executed there is no more work to do, as the label sequences for neighbors differ by exactly one symbol and this symbol was updated in step 2. This is a direct result of the fact that there is a change in orientation at each level of the triangle

quadtrees. As we shall now see, this advantage doesn't extend to the domain of the entire sphere.

Finding neighbors in different base triangles of the polyhedron (e.g., in an icosahedron) is cumbersome using Fekete's method. The problem is that there is no relationship between the ways in which the elements of the adjacent base triangles are labeled. In other words, they are labeled independently of each other, although each base triangle is labeled in a manner consistent with what we have described in Figure 20. Thus we need to use some relationship between the orientations used for the adjacent base triangles of the icosahedron. Fekete [23] suggests that this can be achieved by using substitution tables (see Figure 28). These substitutions are something like Figure 8. Fekete points out that there are six unique relative orientations between two adjacent triangles, as there are three possible ways of choosing the position of child 1 and two ways of choosing the position of child 2, leaving just one choice for the position of child 3, as the position of child 4 is always the same. Thus, six substitution tables are required. One of these six tables is used when finding a neighbor along each of the 60 possible cases of triangle adjacencies (three for each of the 20 base triangles). Note that the 60 different adjacencies do not all make use of the same table, although several may use the same table. The table used depends on the labeling scheme and on the initial orientation that was adopted for each of the 20 base triangles.

At this point, Fekete [23] no longer has the advantage of requiring no work in step three of the algorithm. In fact, our approach has the advantage that nodes corresponding to the faces of the icosahedron labeled 5 to 14 aren't treated any

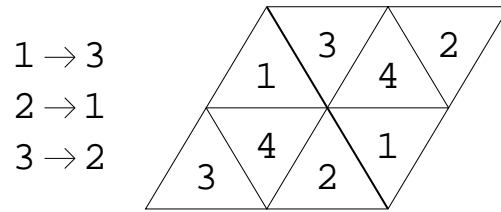


Figure 28: Example of one substitution table to correlate between the different orientations used to label two adjacent base triangles of the icosahedron.

differently than their children. Only nodes corresponding to the faces of the icosahedron labeled 0 to 4 and 15 to 19 require special cases for left and right neighbors in our approach, and this only requires the introduction of one additional relation.

Chapter 3

Three-Dimensional Hierarchies of Tetrahedra

3.1 A Hierarchy of Tetrahedra

The techniques up to this point are only applicable to the surface of three-dimensional data. If we want to represent the complete three-dimensional space or object, then we need a three-dimensional decomposition strategy along with appropriate supporting algorithms. We consider the problem of modeling volume data sets, i.e., sets of points spanning a domain in the three-dimensional space, including scalar field values for each data point. Such data sets are often modeled using tetrahedral meshes. If the the data points come from a regular grid, then it is possible to build a regular tetrahedral mesh from the data. We consider a regular decomposition strategy.

3.1.1 Tetrahedral Decomposition

The bisection rule for tetrahedra consists of replacing a tetrahedron t with the two tetrahedra obtained by splitting t at the middle point of its longest edge and by the plane passing through such point and the opposite edge in t . This rule is applied recursively to an initial decomposition of the cubic domain into six tetrahedra (see Figure 29a). Splitting a tetrahedron in the initial cube subdivision results in two tetrahedra with a shape identical to that obtained by splitting a pyramid with a

square base in half along the diagonal of its base. We call such shape a $1/2$ pyramid (see Figure 30a). Splitting a $1/2$ pyramid along its longest edge results in two tetrahedra whose shape we call a $1/4$ pyramid (see Figure 31a). Finally, splitting a $1/4$ pyramid along its longest edge results in two tetrahedra whose shape we call a $1/8$ pyramid (see Figure 32a). Each of the six initial tetrahedra also have the shape of a $1/8$ pyramid. These shapes are cyclic in the sense that every three levels of decomposition result in a congruent shape.

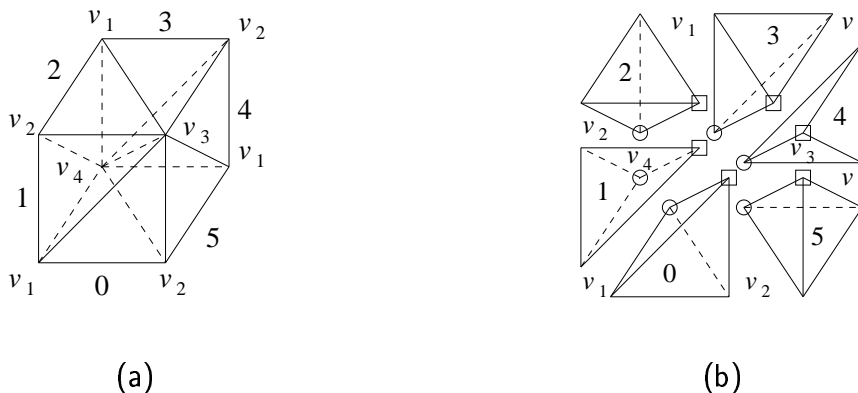


Figure 29: Subdivision of the initial cubic domain into six tetrahedra.

3.1.2 Labeling Tetrahedra in an HT

A *location code* for a tetrahedron t in an HT is a pair of numbers, in which the first denotes the level of t in the tree, and the second indicates the path from the root of the tree to t . The location code for a tetrahedron is defined on the basis of a labeling scheme for its children and for the vertices of these children in the hierarchy, as explained below.

For simplicity of computation, we order the vertices of a tetrahedron in such a way that its longest edge is v_3v_4 . Since the longest edge in the initial 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.

Let $t = [v_1, v_2, v_3, v_4]$ be a tetrahedron and v_m is the midpoint of edge v_3v_4 in t .

The following labeling conventions are assumed for the tetrahedra in the hierarchy:

- If t is a $1/2$ pyramid, 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 $t_1 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_4]$ (see Figure 30a).

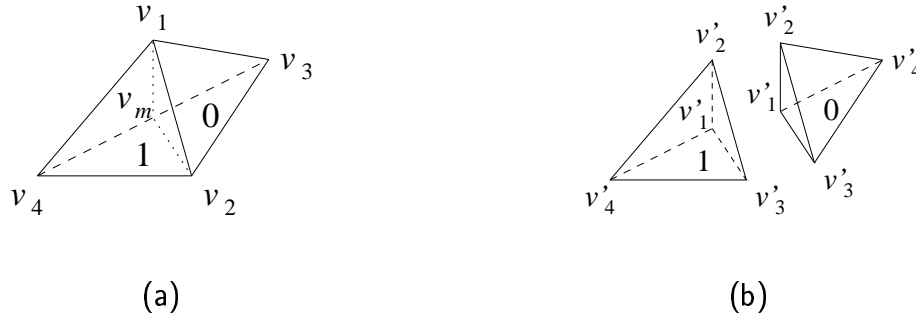


Figure 30: Labeling of a $1/2$ pyramid.

- If t is a $1/4$ pyramid 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 $t_1 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_4]$ (see Figure 31a).
- If t is a $1/4$ pyramid 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 $t_1 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_1, v_2, v_3]$ (see Figure 31a).
- If t is a $1/8$ pyramid, 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 $t_1 = [v'_1, v'_2, v'_3, v'_4] = [v_m, v_2, v_1, v_3]$ (see Figure 32a).

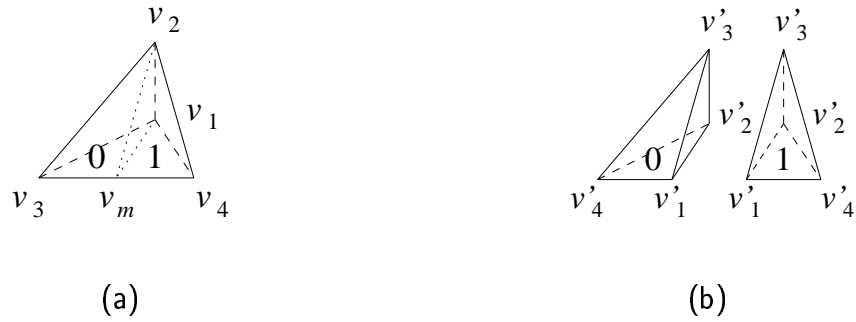


Figure 31: Labeling of a 1/4 pyramid.

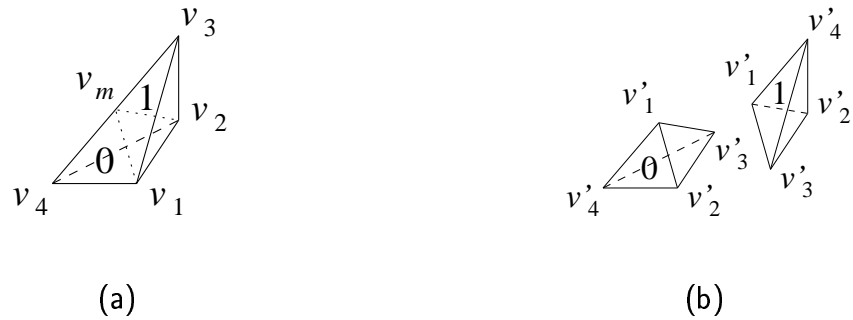


Figure 32: Labeling of a 1/8 pyramid.

3.1.3 Encoding an HT

Our encoding of a hierarchy of tetrahedra makes use of a linear representation instead of a pointer-based one. The data structure consists of:

- a field table containing the field values at the n data points;
- a forest of six almost full binary trees, each of which describes the nested subdivision of one tetrahedron in the initial cube and is encoded as an array containing the errors associated with the tetrahedra corresponding to the internal nodes of the tree.

Note that leaf nodes correspond to the tetrahedra in the mesh at full resolution, and thus, they have a null error. Also, we do not store location codes, but they are

computed on-the-fly and used for indexing the field table and for neighbor finding. To handle large-size data sets, we just store the above data structure on secondary storage, and we perform random accesses to the field and error values.

In our current implementation, we encode with each tetrahedron t the field error associated with t , computed as the absolute value of the maximum of the differences between the actual field value and the interpolated one at each point which falls inside t . An alternative, which is useful in rendering applications, is to store the isosurface error (see [8, 31]), which can also be computed from the field error and the gradient.

If n is the number of points in the data set, $12n$ bytes are required for the error values plus $2n$ bytes for the field table, assuming two bytes per error and field value, leading to a total cost of $14n$ bytes. Note that there are $6n$ tetrahedra in the mesh at full resolution, since each cube of the input grid is subdivided into six tetrahedra, and thus, $6n$ internal tetrahedra. If both the error and the field values are quantized as in [31] to one byte, the storage cost of the data structure reduces to $7n$ bytes.

3.2 Neighbor Finding

In this Section, we describe how to find an equal-sized face neighbor of a tetrahedron. This is used during mesh generation to extract a conforming mesh from the HT so as to avoid any discontinuities in the approximation of the scalar field. The problem is to find the neighbors of a given tetrahedron along an edge, which reduces

to the subproblem of finding the tetrahedron adjacent to a given tetrahedron along a specified face. The algorithm uses the approach defined in [67, 68]. 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 based on the four faces of an arbitrary tetrahedron $t = [v_1, v_2, v_3, v_4]$. Neighbor of type 1 is the tetrahedron which shares face $v_1v_2v_3$ with t . Neighbor of type 2 is the tetrahedron which shares face $v_1v_2v_4$ with t . Neighbor of type 3 is the tetrahedron which shares face $v_1v_3v_4$ with t . Neighbor of type 4 is the tetrahedron which shares face $v_2v_3v_4$ with t . It should be clear that repeated application of a given neighbor type will continuously switch between the two neighbors which share the listed face.

3.2.1 Locating the Nearest Common Ancestor

The first step is to find the nearest common ancestor of tetrahedron t and its neighbor t' of type i . Since we are working with location codes, we want to find the location code of the nearest common ancestor within the given location code.

Basically, given a neighbor direction i (which determines the face we must cross to get the neighbor), we just scan in 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 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. Since 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 33), but we need to know which child contained our input tetrahedron in order to get the appropriate sibling for the neighbor.

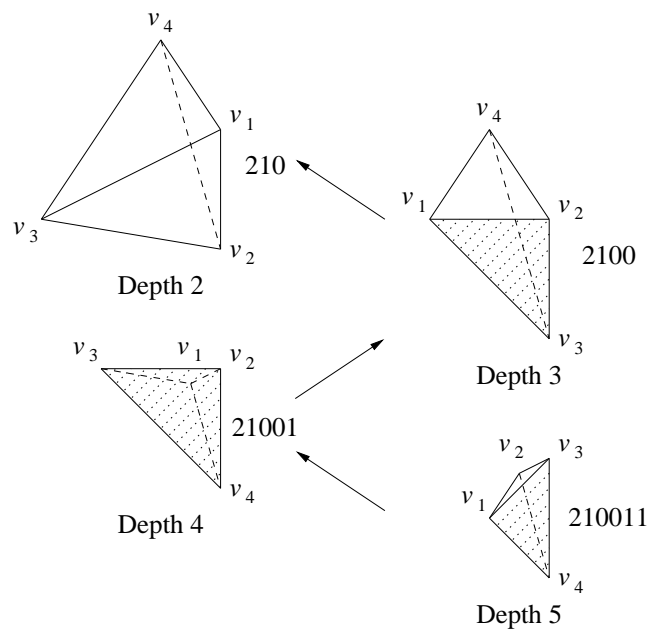


Figure 33: Nearest common ancestor of 210011.

3.2.2 Updating the Location Code

In this step we simply invert the one bit corresponding to the child of the nearest common ancestor. This works regardless of the original neighbor type which we were trying to find. No further work is necessary, since all neighbors' location codes differ by just this one bit.

If we continue with our example from the previous section, 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 34).

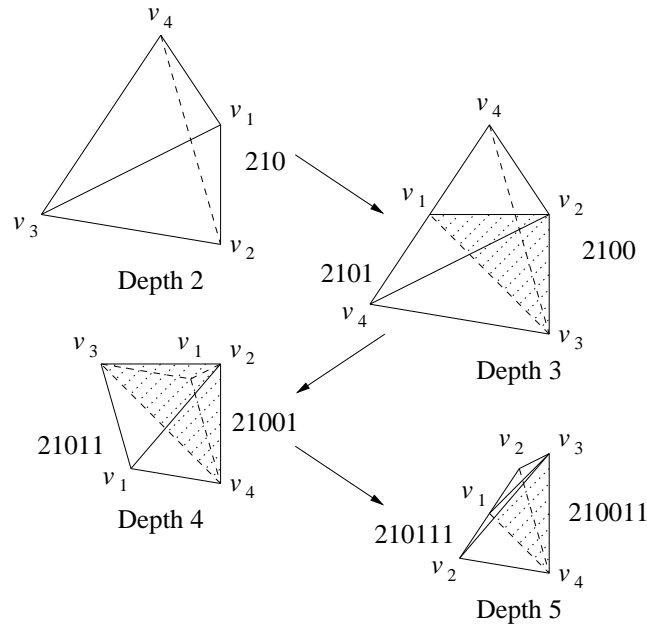


Figure 34: Neighbor type 3 of 210011.

3.2.3 Extensions to the Entire Cube

Since we actually have six tetrahedra as our first decomposition level of the cube, we need to make sure that our transitions work between these six top level tetrahedra. The vertices for these six tetrahedra have been initially labeled so that they imitate the labels of vertices at lower levels in the decomposition. Note that the labeling of these top six tetrahedra themselves is not critical since we can simply use table lookup for 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 we are leaving the cube at this point, then we need to return an error. Otherwise, we know that a neighbor must exist, so we consider the entire cube 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.

Let us consider again the tetrahedron of location code 210011, and let us 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 35).

3.3 Constant-Time Neighbor Finding Algorithm

Here 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 time proportional to 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 can fit in one computer word. If more than one word is needed, then the algorithms are a bit slower but still take constant

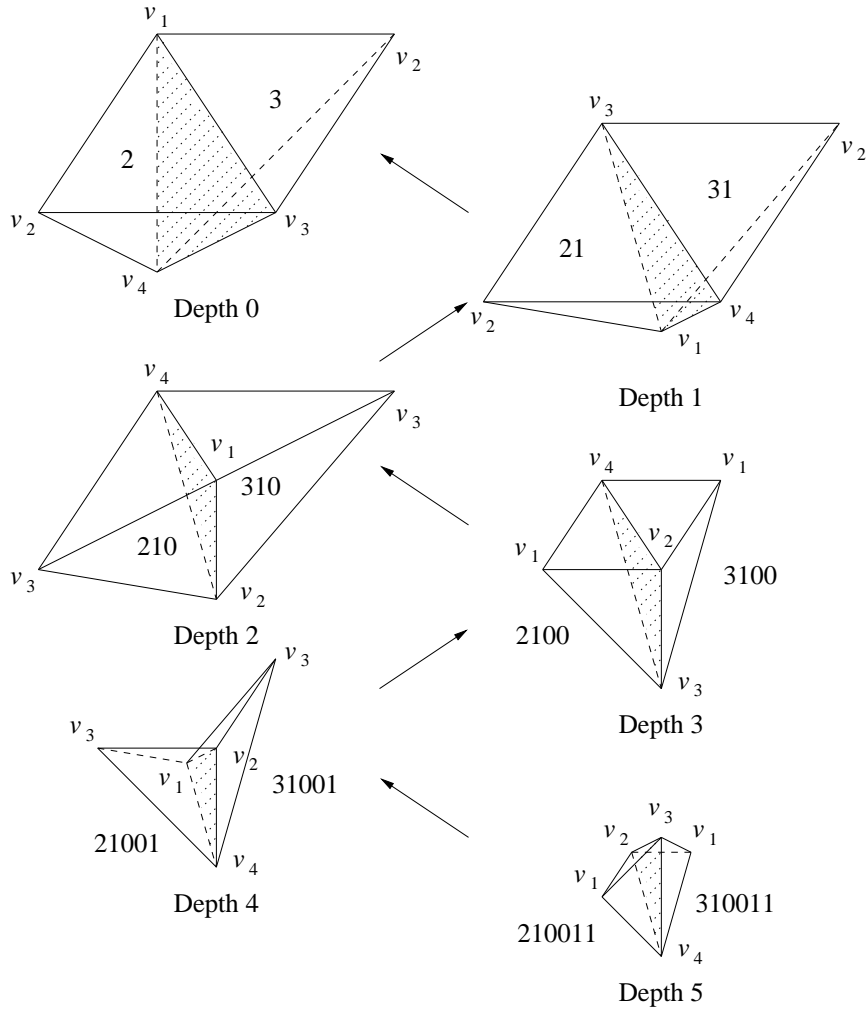


Figure 35: Neighbor type 4 of 210011.

time.

We will use an identification technique similar to the one used in navigating between 2D triangle meshes (see Section 2.4). In the 2D case, we used bit masks to identify certain bit patterns within the location codes. We called them idmasks. These idmasks will 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 we generate our idmasks 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.

3.3.1 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.

```
procedure NEIGHBOR_1(TETRA);
/* Determine the location code of the neighbor which shares face 123 of the tetra-
   hedron with location code TETRA. */
begin
   value pointer location_code TETRA;

   /* Flip the last bit */
   CODE(TETRA) ← XOR(CODE(TETRA), 1);
end;
```

3.3.2 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 3rd ancestor. This is a direct result of the splitting

rules given in Section 3.1.2. If face $v_1v_2v_4$ in the child is contained in face $v_1v_2v_3$ of the 3rd ancestor, then we know the neighbor, because face $v_1v_2v_3$ in the 3rd 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 3rd ancestor, then 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 3rd 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 3rd ancestor (same shape, double size, eight times the volume) depending on which child of the 3rd ancestor was needed to reach the input location code. This means that we need a carry if the child of the 3rd ancestor was child 0, and no carry if it was child 1.

```

procedure NEIGHBOR_2(TETRA);
/* Determine the location code of the neighbor which shares face 124 of the tetra-
   hedron with location code TETRA. */
begin
  value pointer location_code TETRA;
  path_array TEST,FLIP;

  /* Identify positions where sibling can be determined */
  TEST←AND(CODE(TETRA),POS1MASK);
  /* Use carry to find rightmost sibling position */
  FLIP←COMPLEMENT(TEST)+1;
  /* Clear out everything but the final carry */
  FLIP←AND(FLIP,TEST);

```

```

/* Make sure we adjust to the proper level */
FLIP ← SHIFT_LEFT (FLIP) ;
/* Flip the appropriate bit */
CODE (TETRA) ← XOR (CODE (TETRA) , FLIP) ;
end;

```

Notice that the first line of NEIGHBOR_2 locates the positions within the location code where the sibling can be determined because the relevant face is face $v_1v_2v_3$. This result is stored in TEST. Since we want carries where the sibling cannot be determined, we need to complement TEST and then do the addition. To isolate the one bit that needs to be flipped in the location code, we “and” the result of the addition with the value stored in TEST (the positions where we CAN determine the sibling). The bits are offset by one position at this point, so we shift the answer left by one position. Finally, we simply flip the appropriate bit in the location code, by using “xor” between the location code and the current bit mask (which contains only one bit marking the position where we found the sibling).

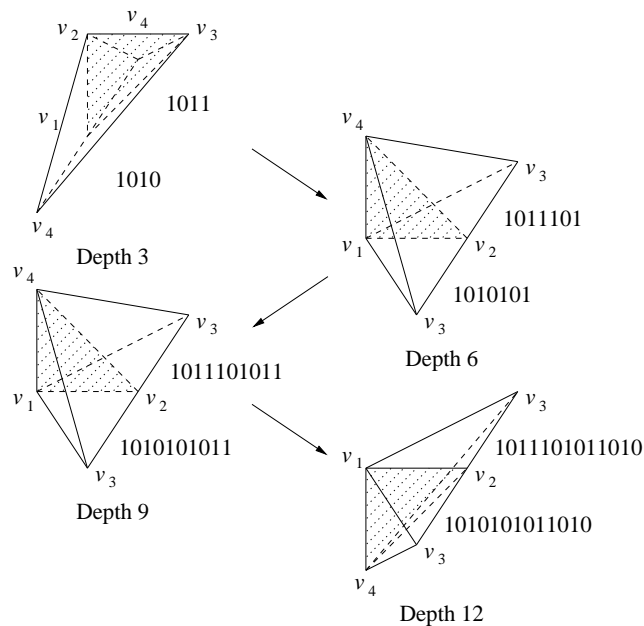


Figure 36: Neighbor type 2 of 1010101011010.

As an example, let us consider location code 1010101011010 (see Figure 36). 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.

3.3.3 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 3rd ancestor. If it is contained by any other face, then the neighbor can be determined without examining the 3rd ancestor. This is a direct result of the splitting rules given in Section 3.1.2. Whenever the neighbor cannot be determined because the nearest common ancestor is beyond the 3rd 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 3rd ancestor), we must continue searching for the neighbor through the appropriate face of the 3rd ancestor. Notice that if this is face $v_1v_3v_4$ of the 3rd ancestor, then we continue to search for the same neighbor type (3). Otherwise, if it is face $v_2v_3v_4$ of the 3rd ancestor, then we must change our strategy a bit, effectively finding neighbor type 4 of the 3rd 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 2nd ancestor or requires finding neighbor type 3 or 4 of the 3rd ancestor depending on the bits corresponding to the 1st and 2nd ancestors of the node. Basically, there should be no carry if the 1st ancestor was child 0 of the 2nd ancestor. Otherwise, we want a carry and the neighbor type will depend on the child type of the 2nd ancestor.

Since a carry occurs whenever we search higher than the 3rd ancestor, and we might need to find either neighbor type 3 or 4 of the 3rd 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 will use a “neighbor mask” to store this information.

```
procedure NEIGHBOR_3(TETRA);
/* Determine the location code of the neighbor which shares face 134 of the tetra-
   hedron with location code TETRA. */
begin
  value pointer location_code TETRA;
  path_array TEST,FLIP;

  /* Identify positions where sibling can be determined */
  TEST←CODE(TETRA);
  TEST←XOR(TEST,SHIFT_LEFT(SHIFT_LEFT(AND(TEST,POS3MASK))));
  TEST←COMPLEMENT(TEST);
  TEST←AND(TEST,XOR(MASK(TETRA),POS2MASK));
  TEST←OR(TEST,SHIFT_LEFT(TEST));
  TEST←AND(TEST,POS1MASK);
  /* Use carry to find rightmost sibling position */
  FLIP←COMPLEMENT(TEST)+1;
  /* Clear out everything but the final carry */
  FLIP←AND(FLIP,TEST);
  /* Make sure we adjust to the proper level */
  if AND(FLIP,MASK(TETRA)) then FLIP←SHIFT_RIGHT(FLIP);
  /* Flip the appropriate bit */
```

```
CODE(TETRA) ← XOR(CODE(TETRA), FLIP);  
end;
```

The first step in finding neighbor type 3, is identifying where (i.e., at what level in the location code) we can determine the neighbor. We want to construct the mask TEST so that it marks the locations where the neighbor can be determined. Since neighbors are determined before we reach the 3rd 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 bits 1 and 3 are the same, then we can identify the neighbor. If bit 2 is 0 and we are looking for neighbor type 3 at this level (determined by looking at the neighbor mask), then we can identify the neighbor. If bit 2 is 1 and we are looking for neighbor type 4 at this level, then we can identify the neighbor. Notice that the mask TEST is constructed based on these patterns. We complement TEST before the addition because we want a carry to occur whenever we cannot identify the neighbor at a given level. The carry continues 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 flipped in the location code, we “and” the result of the addition with the value stored in TEST (the positions where we CAN determine the sibling). Depending on which neighbor type we are finding at this point (again, determined by looking at the neighbor mask), the bits might be offset by one position. If so, we shift the answer right by one position. Finally, we simply flip the appropriate bit in the location code, by using “xor” between the location code and the current bit mask

(which contains only one bit marking the position where we found the sibling).

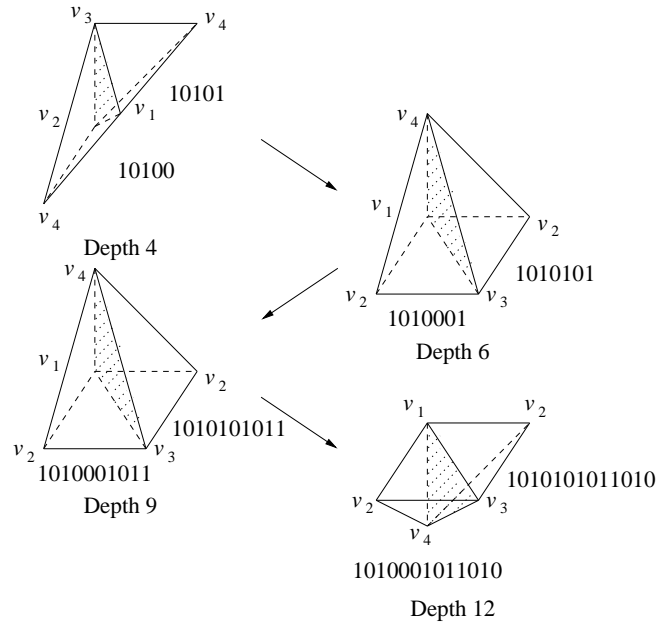


Figure 37: Neighbor type 3 of 1010101011010.

As an example, let us consider location code 1010101011010 (see Figure 37). We can determine the sibling if bits 1 and 3 are the same, or if bit 2 is 0 (since we are only required to find neighbor type 3 in this case). This is true for the first triple (010), true for the second triple (101), not true for the third triple (011), and not true for the fourth triple (010). Therefore, the mask a which marks the true positions is 010010000000. The complement of a is 101101111111. Adding one to this value gives us 101110000000. We isolate the one significant bit using the logical AND. This gives us 000010000000. Finally, we use the logical XOR to invert the appropriate bit. The input location code 1010101011010 XOR 000010000000 gives us our final answer of 1010001011010.

3.3.4 Neighbor Type 4

Finding neighbor type 4 either terminates at the 1st ancestor or requires finding neighbor type 3 or 4 of the 3rd ancestor depending on the bits corresponding to the current node and its 2nd ancestor. Basically, there should be no carry if the 2nd ancestor was child 0 of the 3rd ancestor and the current node is child 0, or if the 2nd ancestor was child 1 of the 3rd 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 2nd ancestor.

```
procedure NEIGHBOR_4(TETRA);
/* Determine the location code of the neighbor which shares face 234 of the tetra-
   hedron with location code TETRA. */
begin
  value pointer location_code TETRA;
  path_array TEST,FLIP;

  /* Identify positions where sibling can be determined */
  TEST←CODE(TETRA);
  TEST←XOR(TEST,SHIFT_LEFT(SHIFT_LEFT(AND(TEST,POS3MASK))));
  TEST←COMPLEMENT(TEST);
  TEST←AND(TEST,XOR(MASK(TETRA),POS1MASK));
  TEST←OR(TEST,SHIFT_LEFT(TEST));
  TEST←AND(TEST,POS1MASK);
  /* Use carry to find rightmost sibling position */
  FLIP←COMPLEMENT(TEST)+1;
  /* Clear out everything but the final carry */
  FLIP←AND(FLIP,TEST);
  /* Make sure we adjust to the proper level */
  if not(AND(FLIP,MASK(TETRA))) then FLIP←SHIFT_RIGHT(FLIP);
  /* Flip the appropriate bit */
  CODE(TETRA)←XOR(CODE(TETRA),FLIP);
end;
```

3.3.5 Updating the Neighbor Mask

The four neighbor relations along with the new bit patterns for stop cases are summarized in the following table.

Current Bits	Neighbor 1	Neighbor 2	Neighbor 3	Neighbor 4
000	001	Cont 2	100	010
001	000	Cont 2	101	Cont 4
010	011	Cont 2	Cont 3	000
011	010	Cont 2	Cont 3	Cont 4
100	101	Cont 1	000	Cont 3
101	100	Cont 1	001	111
110	111	Cont 1	Cont 4	Cont 3
111	110	Cont 1	Cont 4	101

Figure 38: Table indicating how to proceed at each level when searching for the neighboring tetrahedron.

Notice that since neighbor types 3 and 4 switch whenever the child of the 3rd ancestor is child 1. 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 using the following segment of code.

```
procedure UPDATE_MASK(TETRA,FLIP);
```

```

/* Update the neighbor mask of location code TETRA. */
begin
  value pointer location_code TETRA;
  value path_array FLIP;

  if AND(FLIP,POS1MASK) then
    begin
      FLIP←SHIFT_LEFT(-FLIP);

      MASK(TETRA)←XOR(MASK(TETRA),FLIP);
    end;
end;

```

3.3.6 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.

Transitions at the top level are really no different than our previous discussion. We simply pick a new top level tetrahedron based on our table lookup.

3.4 Edge Neighbors

We have defined neighbor type 1 to be the neighbor that shares face $v_1v_2v_3$, neighbor type 2 to be the neighbor that shares face $v_1v_2v_4$, neighbor type 3 to be the neighbor that shares face $v_1v_3v_4$, and neighbor type 4 to be the neighbor that shares face $v_2v_3v_4$. The order of the vertices within each tetrahedron is fully determined based on the split rules which we use during bisection. This technique ensures that a given edge, for example v_2v_3 , in a given tetrahedron is in the same position (v_2v_3) for any neighbors along that edge. Remember that this was the case for face neighbors,

and so it makes sense that it would apply here to edge neighbors.

Knowing these properties, it is really quite simple to find all edge neighbors around a given edge. These edge neighbors can be found using an alternating sequence of two face neighbor operations. The specific two face neighbor types are determined by the edge in question. If we continue with the example v_2v_3 edge, we would use face neighbor types 1 and 4 to find all the edge neighbors. This is because these are the two face neighbors that contain the specified edge (they contain both v_2 and v_3).

The following table shows the sequence of face neighbors that would be required to find all neighbors around a given edge.

Edge	1/2 Pyramids	1/4 Pyramids	1/8 Pyramids
v_1v_2	1, 3, 1, 2	1, 2, 1, 2	1, 2, 1, 2
v_1v_3	1, 2, 1, 3	1, 3, 1, 3	1, 3, 1, 3
v_1v_4	2, 3, 2, 3, 2, 3	2, 3, 2, 3	2, 3, 2, 3, 2, 3, 2, 3
v_2v_3	1, 4, 1, 4, 1, 4, 1, 4	1, 4, 1, 4, 1, 4	1, 4, 1, 4, 1, 4, 1, 4
v_2v_4	2, 4, 2, 4, 2, 4, 2, 4	2, 4, 2, 4, 2, 4	2, 4, 2, 4
v_3v_4	3, 4, 3, 4	3, 4, 3, 4, 3, 4, 3, 4	3, 4, 3, 4, 3, 4

Figure 39: Neighbor sequences for all edges of the tetrahedra.

3.5 Extracting a Conforming Tetrahedral Mesh

In this Section, we discuss how we use the previously described neighbor finding technique to extract a conforming mesh done at a variable level of detail from an HT. A conforming tetrahedral mesh is a mesh with no “cracks”, meaning that the shared faces, edges, and vertices of adjacent tetrahedra must match exactly, for a specified error tolerance. The error associated with each tetrahedron is the maximum of

the absolute value of the difference between the interpolated value and the given field value at each internal point. The interpolated values for internal points are determined using linear interpolation of the field values at the four vertices of the tetrahedron. The resulting tetrahedral mesh should contain the minimum number of tetrahedra which are necessary to satisfy a given error threshold while ensuring that the tetrahedral mesh is conforming.

To generate a conforming tetrahedral mesh we need to ensure that the shared faces of neighboring tetrahedra match exactly. This will ensure that the shared edges of the tetrahedra match as well. This constraint ultimately restricts the size of neighbors to be at most one level different. When splitting any tetrahedron, all tetrahedra that share the bisected edge of the split tetrahedron must also be split in order to maintain the consistency of the tetrahedral mesh. We shall group all such tetrahedra into clusters based on the shared edge. The number of tetrahedra in a given cluster will primarily depend upon the orientation or alignment of the common edge.

There are basically three edge alignments that we will consider. An edge is called *axis-aligned* if it is parallel to one of the coordinate axes. An edge is only *plane-aligned* if it is parallel to one of the coordinate planes, but not to one of the coordinate axes. An edge is considered *non-aligned* otherwise.

3.5.1 Axis-aligned Clusters

Only the 1/4 pyramid splits on an axis-aligned edge. Actually, the longest edge of any 1/4 pyramid will always be an axis-aligned edge, and therefore will always be the edge which causes other neighboring tetrahedra to split. The maximum possible number of neighbors around an axis-aligned edge is 8, so at most 8 tetrahedra will split whenever a 1/4 pyramid is split. Unless we are dealing with a border case, exactly 8 tetrahedra must split simultaneously to maintain a conforming mesh. Therefore, the cluster in this case will contain at most 8 tetrahedra and all members of the cluster will be 1/4 pyramids (see Figure 40).

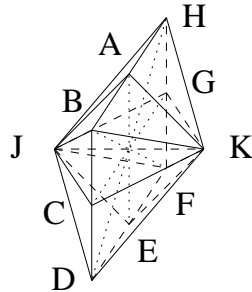


Figure 40: Axis-aligned cluster.

Using our neighbor finding techniques, all 8 tetrahedra can be found starting with any one of them by finding the following sequence of neighbors. Notice that this only requires constant time work.

3.5.2 Plane-aligned Clusters

Only the 1/2 pyramid splits on a plane-aligned edge. Actually, the longest edge of any 1/2 pyramid will always be a plane-aligned edge, and therefore will always be the edge which causes other neighboring tetrahedra to split. The maximum possible

Count	Neighbor Direction	Tetrahedron
1	Initial tetrahedron	$t_1 = [ABJK]$
2	Find neighbor type 4	$t_2 = [CBJK]$
3	Find neighbor type 3 (if it exists)	$t_3 = [CDJK]$
4	Find neighbor type 4	$t_4 = [EDJK]$
5	Find neighbor type 3 (if it exists)	$t_5 = [EFJK]$
6	Find neighbor type 4	$t_6 = [GFJK]$
7	Find neighbor type 3	$t_7 = [GHJK]$
8	Find neighbor type 4	$t_8 = [AHJK]$

Figure 41: Steps required to find an axis-aligned cluster.

number of neighbors around a plane-aligned edge is 4, so at most 4 tetrahedra will split whenever a 1/2 pyramid is split. Unless we are dealing with a border case, exactly 4 tetrahedra must split simultaneously to maintain a conforming mesh. Therefore, the cluster in this case will contain at most 4 tetrahedra and all members of the cluster will be 1/2 pyramids (see Figure 42).

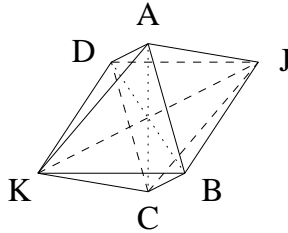


Figure 42: Plane-aligned cluster.

Using our neighbor finding techniques, all 4 tetrahedra can be found starting with any one of them by finding the following sequence of neighbors. Notice that this only requires constant time work.

Count	Neighbor Direction	Tetrahedron
1	Initial tetrahedron	$t_1 = [ABJK]$
2	Find neighbor type 3	$t_2 = [ADJK]$
3	Find neighbor type 4 (if it exists)	$t_3 = [CDJK]$
4	Find neighbor type 3	$t_4 = [CBJK]$

Figure 43: Steps required to find a plane-aligned cluster.

3.5.3 Non-aligned Clusters

Only the 1/8 pyramid splits on a non-aligned edge. Actually, the longest edge of any 1/8 pyramid will always be a non-aligned edge, and therefore will always be the edge which causes other neighboring tetrahedra to split. The maximum possible number of neighbors around a non-aligned edge is 6, so at most 6 tetrahedra will split whenever a 1/8 pyramid is split. Unless we are dealing with a border case (and it is impossible for non-aligned edges to touch the border), exactly 6 tetrahedra must split simultaneously to maintain a conforming mesh. Therefore, the cluster in this case will contain exactly 6 tetrahedra and all members of the cluster will be 1/8 pyramids (see Figure 44).

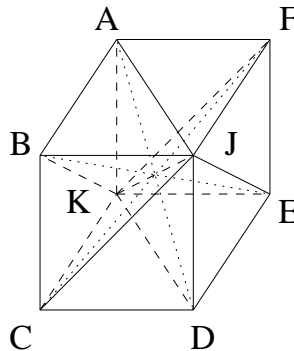


Figure 44: Non-aligned cluster.

Using our neighbor finding techniques, all 6 tetrahedra can be found starting with any one of them by finding the following sequence of neighbors. Notice that

this only requires constant time work.

Count	Neighbor Direction	Tetrahedron
1	Initial tetrahedron	$t_1 = [ABJK]$
2	Find neighbor type 3	$t_2 = [AFJK]$
3	Find neighbor type 4	$t_3 = [EFJK]$
4	Find neighbor type 3	$t_4 = [EDJK]$
5	Find neighbor type 4	$t_5 = [CDJK]$
6	Find neighbor type 3	$t_6 = [CBJK]$

Figure 45: Steps required to find a non-aligned cluster.

3.6 Algorithms for Selective Refinement

In this Section, we describe three algorithms for performing selective refinement on an HT, which are based on a depth-first, priority queue, and incremental approach, respectively. A *selective refinement* operation applied to a multi-resolution mesh M consists of extracting a conforming mesh Σ from M such that Σ covers the domain D of M , the resolution of Σ satisfies some user-defined error requirements, and Σ is the mesh with the smallest number of tetrahedra satisfying the above conditions (see [13]). In [8], a set of basic queries are defined for analysis and visualization of a volume data set, that are called *Level-Of-Detail (LOD)* queries, and it is shown that all of them are instances of selective refinement.

3.6.1 A Depth First Approach

The depth-first algorithm is a standard refinement algorithm (see, for instance, [50]). It starts from the initial mesh, consisting of the six top-level tetrahedra which subdivide the cube, and initializes the currently extracted mesh (that we call

the *current mesh*) with them. For any tetrahedron t , that does not satisfy the error requirements, we split the cluster c of tetrahedra which share their longest edge with t . If a tetrahedron t' in c does not exist in the current mesh, then we split the cluster associated with the parent of t' . This is applied recursively in order to guarantee a conforming mesh. The process continues until all tetrahedra in the current mesh satisfy the error requirements.

A pseudo-code description of the depth-first algorithm is given below. Predicate $\text{EXIST}(t)$ returns the value *true* if tetrahedron t is part of the current mesh, and the value *false* otherwise. Function $\text{PARENT}(t)$ deletes the last bit from the location code of t , thus returning the parent of t . Functions $\text{CHILD}_0(t)$ and $\text{CHILD}_1(t)$ add a 0-bit and a 1-bit to the end of the location code of t , respectively, thus returning the first and the second child of t , respectively. Function $\text{SPLIT}(t)$ replaces t in the current mesh with $\text{CHILD}_0(t)$ and $\text{CHILD}_1(t)$. Function $\text{CLUSTER}(t)$ returns the set of all tetrahedra sharing the splitting edge v_3v_4 with tetrahedron t . This set can be found by using the alternating sequence of face neighbors described in the previous section.

```

procedure SPLIT_CLUSTER(TETRA);
/* Split all tetrahedra within the same cluster as TETRA. This includes splitting
  TETRA itself. */
begin
  value pointer location_code TETRA;
  pointer location_code NEIGHBOR;

  for NEIGHBOR in {CLUSTER(TETRA)} do
    begin
      if not(EXIST(NEIGHBOR)) then SPLIT_CLUSTER(PARENT(NEIGHBOR));

      SPLIT(NEIGHBOR);
    end;

```

```

end;

procedure CHECK_SPLIT(TETRA);
/* Check if TETRA satisfies the error requirement and split if necessary. */
begin
    value pointer location_code TETRA;

    if FAIL_ERROR_REQ(TETRA) then
        begin
            SPLIT_CLUSTER(TETRA);

            CHECK_SPLIT(CHILD_0(TETRA));
            CHECK_SPLIT(CHILD_1(TETRA));
        end;
    end;
end;

procedure EXTRACT_MESH();
/* Extract a conforming mesh where all tetrahedra satisfy the error requirement.
*/
begin
    location_code TETRA;

    for TETRA in {'0', '1', '2', '3', '4', '5'} do
        begin
            CHECK_SPLIT(TETRA);
        end;
    end;
end;

```

It can be easily seen that the worst-case time complexity of the algorithm is $O(k)$, where k is the number of tetrahedra in the hierarchy. Since the tetrahedra are only split in `SPLIT_CLUSTER` if forced by our consistency constraint, and `SPLIT_CLUSTER` is only called if one of the tetrahedra is beyond the error threshold, the algorithm never splits a tetrahedron unless it is necessary, and thus, it generates the minimum number of tetrahedra which are required to satisfy the error requirements.

3.6.2 A Priority Based Approach

The priority-based algorithm applies the splitting process to the tetrahedra according to an error-driven sequence. The tetrahedron with the largest error is split at each iteration. As in the depth-first approach, when a tetrahedron t is split, all the tetrahedra in the same cluster as t must be split as well. If a tetrahedron t' in the cluster is not in the current mesh, the splitting process is recursively applied to the parent of t' . The algorithm makes use of a priority queue of tetrahedra, in which the order is based on the errors associated with the tetrahedra. Tetrahedra are only added to the priority queue if they violate the error requirements. To reduce the number of insertions and the overall size of the priority queue, we simply insert only one of two children of a tetrahedron if they belong to the same cluster, namely the one with the largest error. Only tetrahedra on the queue, or tetrahedra which must be included in the current mesh to maintain consistency, are considered for splitting. It can be easily seen that the algorithm generates the minimum number of tetrahedra necessary to satisfy the error requirements, and that the time complexity of the algorithm is also linear in the number of tetrahedra in the hierarchy.

```
procedure CHECK_SPLIT2(TETRA);  
/* Check if TETRA satisfies the error requirement and, if not, check if it has a higher  
   error than its BUDDY. */  
begin  
  value pointer location_code TETRA;  
  pointer location_code BUDDY;  
  
  if FAIL_ERROR_REQ(TETRA) then  
    begin  
      BUDDY ← NEIGHBOR_3(TETRA);  
  
      if FAIL_VALUE(TETRA) > FAIL_VALUE(BUDDY) then ENQUEUE(TETRA);  
    end;
```

```

end;

procedure SPLIT_CLUSTER2(TETRA);
/* Split all tetrahedra within the same cluster as TETRA. This includes splitting
   TETRA itself. Add any children which do not satisfy the error requirement to the
   priority queue. */
begin
  value pointer location_code TETRA;
  pointer location_code NEIGHBOR;

  for NEIGHBOR in {CLUSTER(TETRA)} do
    begin
      if not(EXIST(NEIGHBOR)) then SPLIT_CLUSTER2(PARENT(NEIGHBOR));

      SPLIT(NEIGHBOR);

      CHECK_SPLIT2(CHILD_0(NEIGHBOR));
      CHECK_SPLIT2(CHILD_1(NEIGHBOR));
    end;
  end;
end;

procedure EXTRACT_MESH2();
/* Extract a conforming mesh where all tetrahedra satisfy the error requirement.
   */
begin
  location_code TETRA;

  for TETRA in {'0', '1', '2', '3', '4', '5'} do
    begin
      if FAIL_ERROR_REQ(TETRA) then ENQUEUE(TETRA);
    end;

    while DEQUEUE(TETRA) do
      begin
        if EXIST(TETRA) then SPLIT_CLUSTER2(TETRA);
      end;
    end;
end;

```

The priority-based approach produces an interruptible algorithm i.e., it generates a fairly good approximation of the solution, if time has expired, or the number of tetrahedra in the current mesh is above a predefined bound. On the other hand,

using a priority queue increases the storage cost, but we have found experimentally that the size of the queue is on average equal to 10 – 16% of the size of the output mesh.

Figure 46 shows the number of tetrahedron splits and cluster computations per second performed by the in-core and the out-of-core versions of the depth-first algorithm and by the out-of-core version of the priority based algorithm. The experiments were done on a Pentium III 650MHz machine with 384 Megs of Ram running a Linux OS. The results show that the in-core version of the depth-first algorithm is more than 50% faster than the out-of-core version, while the priority based algorithm is about 20% slower than its depth-first counterpart.

Algorithm	Tetrahedron Splits		Cluster Computations	
	Range	Average	Range	Average
Depth-first in-core	560000-695000	641000	115000-122000	118000
Depth-first out-of-core	250000-485000	402000	46000-85000	74000
Priority out-of-core	210000-465000	336000	39000-82000	61000

Figure 46: The second column shows the number of tetrahedron splits per second, the third column shows the number of cluster computations per second.

3.6.3 An Incremental Approach

In this Section, we describe an algorithm for selective refinement based on an *incremental* approach. The algorithm considers the current mesh resulting from a previous execution as the starting mesh and modifies such a mesh according to new error requirements. Thus, the current mesh not only may be refined by splitting tetrahedra in a cluster, but also may be coarsened by merging all tetrahedra incident

at a vertex. Such an approach may give sub-optimal solutions if the error does not decrease monotonically, but it is very useful in highly interactive environments, since it minimizes the work performed in updating the mesh.

A pseudo-code description of the incremental algorithm is given below. Function `MERGE(t)` replaces `CHILD_0(t)` and `CHILD_1(t)` in the current mesh with t . The other primitives used in the algorithm description have been introduced in the previous section. Procedure `UPDATE_MESH` considers each tetrahedron in the input, or in the current mesh. For any tetrahedron that does not satisfy the error requirements, a downward refinement operation is started. This is performed by `CHECK_SPLIT`. For any tetrahedron that is too refined, `CHECK_MERGE` is called to coarsen the mesh in a bottom-up direction.

Note that we never “force” a merge like we do in `SPLIT_CLUSTER` (where we force other splits). Merging the tetrahedra incident at a vertex can only be performed if all the tetrahedra to be merged are in the current mesh, and all the tetrahedra in the corresponding cluster satisfy the error requirements.

```
procedure MERGE_CLUSTER(TETRA);
/* Merge all tetrahedra within the same cluster as TETRA. This includes merging
   TETRA with its sibling. */
begin
  value pointer location_code TETRA;
  pointer location_code NEIGHBOR;

  for NEIGHBOR in CLUSTER(PARENT(TETRA)) do
    begin
      MERGE(NEIGHBOR);
    end;
end;

procedure CHECK_MERGE(TETRA);
/* Check if a merge is possible and then merge as appropriate. */
```

```

begin
  value pointer location_code TETRA;
  pointer location_code NEIGHBOR;

  for NEIGHBOR in CLUSTER(PARENT(TETRA)) do
    begin
      if not(EXIST(CHILD_0(NEIGHBOR))) then RETURN;
      if not(EXIST(CHILD_1(NEIGHBOR))) then RETURN;

      if FAIL_ERROR_REQ(NEIGHBOR) then RETURN;
    end;

    MERGE_CLUSTER(TETRA);

    CHECK_MERGE(PARENT(TETRA));
  end;

  procedure UPDATE_MESH();
  /* Update the current mesh so that the resulting tetrahedra satisfy the new error
  requirement. */
  begin
    location_code TETRA;

    for TETRA in CURRENT_MESH do
      begin
        if FAIL_ERROR_REQ(TETRA) then
          CHECK_SPLIT(TETRA);
        else
          CHECK_MERGE(TETRA);
        end;
      end;
    end;
  end;

```

Our experiments with the incremental algorithm, when the data structure is maintained out-of-core, show that the approximate number of tetrahedron splits per second is about 280000 on average, which corresponds to approximately 51000 cluster computations per second. The number of tetrahedron merges per second is approximately 300000 on average, which corresponds to approximately 54000 cluster computations per second.

Note that the algorithm described in [31] also performs incremental selective refinement, but it makes use of two priority queues containing candidate tetrahedra to be split or to be merged (see also [16]). The use of priority queues make the algorithm interruptible at the expense of extra storage.

3.7 Experimental Results

In this Section, we report performance statistics on LOD queries on the HT implemented with the depth-first algorithm. We have used two volume data sets of different sizes and characteristics:

- **Plasma64** (274,625 vertices, 1,572,864 tetrahedra), a large synthetic data set whose field values represent the 3D Perlin's noise (courtesy of Visual Computing Group, National Research Council, Pisa, Italy).
- **Buckyball** (2,146,689 vertices, 12,582,912 tetrahedra), a very large regular data set (courtesy of AVS Inc).

We show results on multi-resolution queries that extract a 3D mesh according to the following LOD criteria:

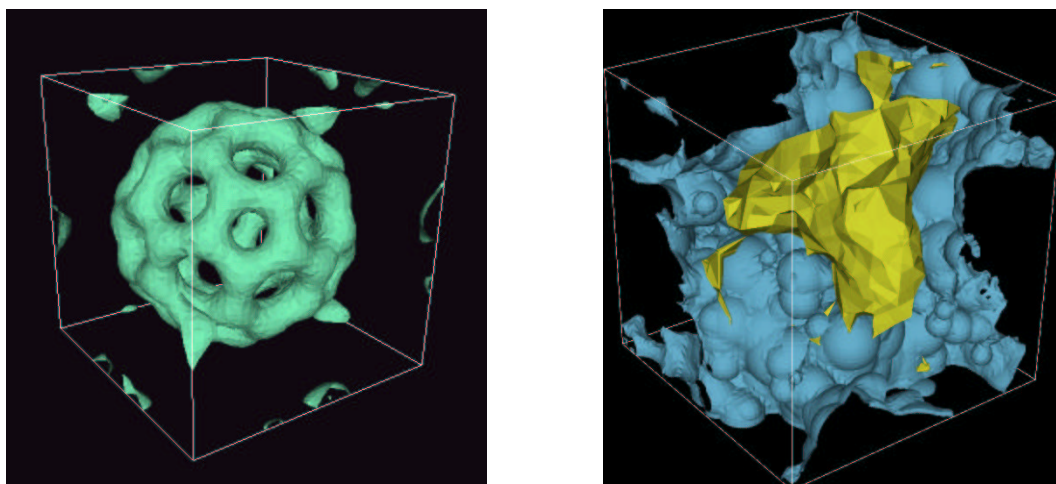
- *Uniform LOD*: extraction of a mesh satisfying a constant error threshold; in our experiments, the threshold varies from zero to the maximum error value.
- *Variable LOD in a box*: extraction of a mesh with an error below a given threshold inside a 3D box and no error constraint outside it; in our experi-

ments, the box is at a fixed position, and the threshold inside the box varies from zero to the maximum error.

- *Variable LOD based on a field value*: extraction of a mesh with an error below a certain threshold on the pentatopes intersecting the isosurface of the specified field value, and no error constraint elsewhere.

We have considered a uniform LOD query where the error is required to be smaller than a given threshold value over the whole domain (see Figure 47a), and, as an example of a variable LOD query, a query based on a field value (see Figure 47b). In this latter case we require a high accuracy in a specified part of the domain or in the proximity of a specified isosurface, and a lower one elsewhere. Our results are in terms of the number of tetrahedra. This quantity is directly related to the complexity of the queries as the execution time of the selective refinement algorithms depends on this parameter.

Figure 48 shows the number of tetrahedra in the extracted mesh for a uniform LOD query and the percentage of tetrahedra with respect to mesh at full resolution for both data sets. The error threshold is expressed as a percentage of the absolute value of the range of the field values in the data sets. Figure 49 reports the same statistics for a variable LOD query based on a ROI (in this case, an axis-aligned box) on the Buckyball data set. Different values of the error threshold have been selected inside the box, while an arbitrary large value of the error is allowed in the rest of the domain. Figure 50 reports the same statistics for a variable LOD query based on a field value for the Plasma data set. Different values of the error threshold



(a)

(b)

Figure 47: Uniform LOD extraction (a): error threshold equal to 5.0% of the field range of the whole domain. The isosurface for a field value equal to 100.0 is shown. Variable LOD extraction based on a field value (b): error threshold equal to 0.1% of the field range enforced near isosurface of value 1.27 (blue). The isosurfaces for field values equal to 1.27 and 1.45 are shown.

have been selected for the tetrahedra intersecting the isosurface, while an arbitrary large value of the error is allowed for the other tetrahedra.

For comparison purposes, we have also implemented a technique for extracting conforming meshes from a hierarchy of tetrahedra based on *error saturation*, as discussed in [54, 79]. First, all tetrahedra belonging to the same cluster are assigned the same error value, which is equal to the maximum of their original error values. Moreover, the approximation error associated with each tetrahedron is saturated to be greater than or equal to the error associated with its children. This implies that, during mesh extraction, if a tetrahedron is refined, then all tetrahedra belonging to the same cluster are refined.

Uniform LOD Query				
error (% of field range)	Plasma64		Buckyball	
	tetrahedra	% tetrahedra	tetrahedra	% tetrahedra
0.1	1,493,696	94.9%	9,276,978	73.7%
0.5	620,996	39.4%	2,792,664	22.1%
1.0	337,384	21.4%	1,352,728	10.7%
5.0	16,800	1.1%	247,760	1.9%
10.0	2,818	0.2%	95,400	0.7%

Figure 48: Number of tetrahedra in the meshes at *uniform* LOD and percentage with respect to the number of tetrahedra in the mesh at full resolution extracted from the HT representation of the Plasma and Buckyball data sets, respectively.

Variable LOD Based on a Region of Interest		
error (% of field range)	Buckyball	
	tetrahedra	% tetrahedra
0.1	405,860	3.22%
0.5	190,936	1.51%
1.0	102,430	0.81%
5.0	23,108	0.18%
10.0	10,382	0.08%

Figure 49: Number of tetrahedra in the meshes at *variable* LOD based on a region of interest and percentage with respect to the number of tetrahedra in the mesh at full resolution extracted from the HT representation of the Buckyball data set. The error within the region is specified in the left column.

Our experimental comparisons, that we have performed on the basis of LOD queries at a uniform resolution, have shown that the meshes extracted from a saturated HT have, on average, 5% more tetrahedra than those extracted with our method. On the other hand, the computing times of our depth-first algorithm are the same as those of the corresponding algorithm on a saturated HT (which simply performs a top-down traversal of the hierarchy without any neighbor finding computation).

Variable LOD Based on Field Value		
error (% of field range)	Plasma64 tetrahedra	% tetrahedra
0.0	19,070	1.2%
0.1	18,266	1.1%
0.5	12,100	0.8%
1.0	9,748	0.6%
5.0	1,912	0.1%
10.0	810	0.1%

Figure 50: Number of tetrahedra in the meshes at *variable* LOD and percentage with respect to the number of tetrahedra in the mesh at full resolution extracted from the HT representation of the Plasma data set. The error within the proximity of the isosurface is specified in the left column.

We have also compared an HT with a multi-resolution model based on unstructured meshes built through edge collapse [11], called an *Edge-based Multi-Tessellation (Edge-based MT)* on the basis of their selectivity on a set of LOD queries. The comparisons included uniform LOD queries, variable LOD queries based on a region of interest, and variable LOD queries based on a specified field value. The experiments on the queries that we performed showed that the HT performed better than the MT in terms of the number of tetrahedra in that there were fewer tetrahedra for the HT than for the MT except when using a uniform level of detail.

There are other methods which use techniques based on a variant of the typical location code for identifying tetrahedra. Hebert [34] introduces the idea of using symbolic algorithms to find parents, children, and neighbors. 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. This results in a neighbor finding algorithm that takes time proportional to the depth in the hierarchy.

Chapter 4

Four-Dimensional Hierarchies of Pentatopes

4.1 Pentatopic Decomposition

In this Section, we consider four-dimensional data sets represented as a hierarchy of four-dimensional simplexes, that we call *pentatopes*. The resulting hierarchy, that we call a *Hierarchy of Pentatopes (HP)*, forms the basis for producing adaptive decompositions of the domain of a four-dimensional scalar field.

The general decomposition strategy starts with a hypercube, which is subdivided into 24 pentatopes, all sharing an edge (the diagonal of the hypercube) which connects a pair of vertices of the hypercube which do not belong to the same face (cube, square or edge) in the hypercube. A pentatope is bounded by 5 0-simplexes (vertices), 10 1-simplexes (edges), 10 2-simplexes (triangles), and 5 3-simplexes (tetrahedra). Two of the five tetrahedral faces of each pentatope are contained by one of the eight cubic faces of the hypercube, while each of the remaining three faces is shared by two pentatopes in the subdivision of the hypercube.

The pentatopes at level 0 in an HP result from the initial subdivision of the hypercube. The pentatopes at level $i + 1$ are generated by bisecting pentatopes at level i . We need four bisection steps in order to create a pentatope at level i ($i > 3$) which is a factor of two smaller in all directions than its ancestor at level $i - 4$. Pentatopes at level i are congruent to their ancestors at level $i - 4$

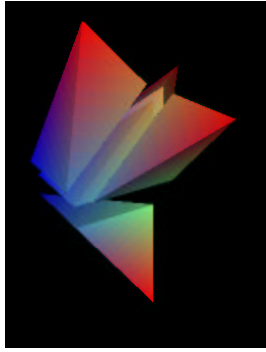
modulus reflections. In other words, the bisection rule generates four classes of congruent pentatopes. This result has been proven by Maubach [50] in the general d -dimensional case: the number of congruency classes generated through bisection is equal to d , independently of the level of refinement.

For clarity, we describe and classify the four simplicial shapes generated by the bisection process (we denote with h the initial hypercube) as follows:

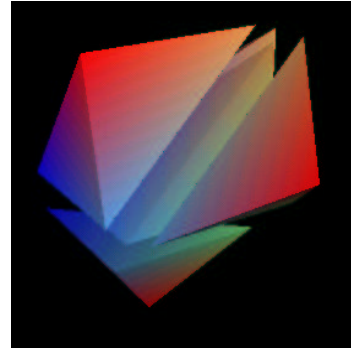
- *h-pentatope*: pentatope initially generated at level 0 by the subdivision of hypercube h .
- *c-pentatope*: pentatope initially generated at level 1 by splitting an h -pentatope along its longest edge, which is the diagonal of hypercube h .
- *s-pentatope*: pentatope initially generated at level 2 by splitting a c -pentatope along its longest edge, which is the diagonal of a cubic face of hypercube h .
- *e-pentatope*: pentatope initially generated at level 3 by splitting an s -pentatope along its longest edge, which is the diagonal of a square face of hypercube h .

Note that an h -pentatope is then generated at level 4 by the subdivision of an e -pentatope at level 3 along its longest edge, which is an edge of hypercube h . Thus, in an HP there are h -pentatopes at levels $4j$, c -pentatopes at levels $4j + 1$, s -pentatopes at levels $4j + 2$, and e -pentatopes at levels $4j + 3$, $j = 0, 1, \dots, i$.

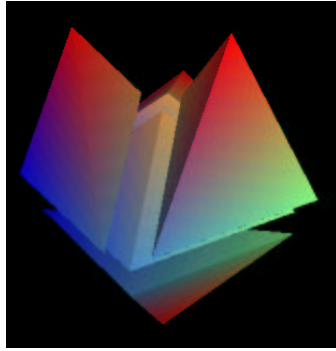
A hierarchy of pentatopes is generated from a grid of field values by top-down recursive bisection of the initial hypercubic domain. We denote as V the set of grid vertices at which the field value is known. An approximation error is computed



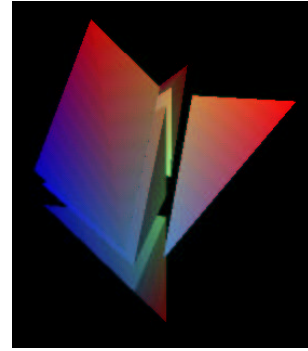
(a)



(b)



(c)



(d)

Figure 51: Example of (a) an h -pentatope, (b) a c -pentatope, (c) an s -pentatope and (d) an e -pentatope. The figures show the unfolding in 3D space of each pentatope by representing its five tetrahedral faces.

for each pentatope σ . We consider the *error* associated with σ as the maximum of the absolute value of the difference between the actual field value at the points of V inside σ and the field value at the same points linearly interpolated within σ . Thus, we could encode such errors by storing the full binary tree describing the HP as an array, where each element of the array corresponds to a pentatope σ , and it contains just the error associated with σ . Actually, there is no need to store the leaves of the hierarchy since all the corresponding pentatopes have a null error. The

storage requirements are equal in this case to $48n$ bytes, since a hypercube is split into 24 pentatopes, where n is the number of points in V , by assuming to encode the error in 2 bytes. On the other hand, we can avoid encoding the hierarchy, if we associate the error to the vertices in V . The error associated with a vertex p will be the maximum of the errors associated with the pentatopes that have been split by the introduction of vertex p . This will reduce the space requirements for encoding errors to $2n$ bytes.

4.2 Labeling Pentatopes in an HP

Each pentatope σ in a hierarchy of pentatopes, with the exception of those belonging to the subdivision of the initial hypercube, is labeled with one bit, depending on whether σ is the child 0 or child 1 of its parent. In this way, any pentatope in the hierarchy can be uniquely identified through a *location code*. A location code for a pentatope σ in an HP consists of a pair of numbers, in which the first number denotes the level of σ in the tree, while the second number denotes the path from the root of the tree to σ . This path is a sequence of bits each corresponding to a pentatope in the path from the root to σ .

Let $\sigma = [v_1, v_2, v_3, v_4, v_5]$ be a pentatope and v_m be the midpoint of the longest edge of σ . We denote with σ_0 and σ_1 child 0 and child 1, respectively, of σ . Eight cases arise depending on whether σ is an *h*-pentatope, a *c*-pentatope, an *s*-pentatope or an *e*-pentatope, and on the parent-child relations in the hierarchy. These cases are summarized in Figure 52.

- σ is an h -pentatope: the two resulting c -pentatopes are $\sigma_0 = [v_m, v_1, v_2, v_3, v_5]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_4]$, where v_m is the midpoint of edge $[v_4, v_5]$ in σ .
- σ is a c -pentatope: If σ is child 0, then the two resulting s -pentatopes are $\sigma_0 = [v_m, v_1, v_2, v_3, v_5]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_4]$, where v_m is the midpoint of edge $[v_4, v_5]$ in σ .
- σ is a c -pentatope: If σ is child 1, then the two resulting s -pentatopes are $\sigma_0 = [v_m, v_1, v_5, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_4]$, where v_m is the midpoint of edge $[v_2, v_5]$ in σ .
- σ is an s -pentatope: If σ is child 0 and its parent child 0, then the two resulting e -pentatopes are $\sigma_0 = [v_m, v_1, v_2, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_5]$, where v_m is the midpoint of edge $[v_4, v_5]$ in σ .
- σ is an s -pentatope: If σ is child 1 and its parent child 0, then the two resulting e -pentatopes are $\sigma_0 = [v_m, v_1, v_2, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_5, v_4]$, where v_m is the midpoint of edge $[v_3, v_5]$ in σ .
- σ is an s -pentatope: If σ is child 0 and its parent child 1, then the two resulting e -pentatopes are $\sigma_0 = [v_m, v_1, v_2, v_5, v_3]$ and $\sigma_1 = [v_m, v_1, v_2, v_5, v_4]$, where v_m is the midpoint of edge $[v_3, v_4]$ in σ .
- σ is an s -pentatope: If σ is child 1 and its parent child 1, then the two resulting e -pentatopes are $\sigma_0 = [v_m, v_1, v_2, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_5, v_4]$, where v_m is the midpoint of edge $[v_3, v_5]$ in σ .

- σ is an e -pentatope: the two resulting h -pentatopes are $\sigma_0 = [v_m, v_1, v_2, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_5]$, where v_m is the midpoint of edge $[v_4, v_5]$ in σ .

Case	Shape	G	P	Split Edge	Resulting Children
1	h -pentatope			$[v_4, v_5]$	c -pentatopes: $\sigma_0 = [v_m, v_1, v_2, v_3, v_5]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_4]$
2	c -pentatope		0	$[v_4, v_5]$	s -pentatopes: $\sigma_0 = [v_m, v_1, v_2, v_3, v_5]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_4]$
3	c -pentatope		1	$[v_2, v_5]$	s -pentatopes: $\sigma_0 = [v_m, v_1, v_5, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_4]$
4	s -pentatope	0	0	$[v_4, v_5]$	e -pentatopes: $\sigma_0 = [v_m, v_1, v_2, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_5]$
5	s -pentatope	0	1	$[v_3, v_5]$	e -pentatopes: $\sigma_0 = [v_m, v_1, v_2, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_5, v_4]$
6	s -pentatope	1	0	$[v_3, v_4]$	e -pentatopes: $\sigma_0 = [v_m, v_1, v_2, v_5, v_3]$ and $\sigma_1 = [v_m, v_1, v_2, v_5, v_4]$
7	s -pentatope	1	1	$[v_3, v_5]$	e -pentatopes: $\sigma_0 = [v_m, v_1, v_2, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_5, v_4]$
8	e -pentatope			$[v_4, v_5]$	h -pentatopes: $\sigma_0 = [v_m, v_1, v_2, v_3, v_4]$ and $\sigma_1 = [v_m, v_1, v_2, v_3, v_5]$

Figure 52: Table with splitting rules. The second column denotes the shape of the pentatope σ which is split, the third column (G) indicates whether the parent of σ is child 0 or child 1 of the grandparent of σ , the fourth column (P) indicates whether σ is child 0 or child 1 of its parent, the fifth column shows the split edge of σ , the sixth column shows the pentatopes resulting from the split, and their vertices.

The table shows the shape of the pentatope which is split, its split edge and the two resulting pentatopes. Note that for a c -pentatope σ , two possible cases arise depending on whether σ is a child 0 or a child 1. For an s -pentatope σ , there are four cases which depend on the parent-child relation between σ and its parent, and between the parent and the grandparent of σ .

A hierarchy of pentatopes, when used as the domain decomposition of a four-dimensional scalar field, does not need to be explicitly stored either in a pointer-

based representation or through location codes. Location codes are computed on-the-fly and used in neighbor finding to locate the pentatopes sharing a given edge.

4.3 Neighbor Finding

In this Section, we describe how to find an equal-sized face neighbor of a pentatope. This is used during mesh generation to extract a conforming mesh from the HP so as to avoid any discontinuities in the approximation of the scalar field. The problem is to find the neighbors of a given pentatope along an edge, which reduces to the subproblem of finding the pentatope adjacent to a given pentatope along a specified tetrahedral face. The algorithm uses the approach defined in [67, 68]. We will not make use of the actual coordinate values of the pentatope 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 five neighbor directions based on the five tetrahedral faces of an arbitrary pentatope $t = [v_1, v_2, v_3, v_4, v_5]$. Neighbor of type 1 is the pentatope which shares face $v_1v_2v_3v_4$ with t . Neighbor of type 2 is the pentatope which shares face $v_1v_2v_3v_5$ with t . Neighbor of type 3 is the pentatope which shares face $v_1v_2v_4v_5$ with t . Neighbor of type 4 is the pentatope which shares face $v_1v_3v_4v_5$ with t . Neighbor of type 5 is the pentatope which shares face $v_2v_3v_4v_5$ with t . It should be clear that repeated application of a given neighbor type will continuously switch between the two neighbors which share the listed face.

4.3.1 Locating the Nearest Common Ancestor

Let σ be the given pentatope and σ' the k -neighbor of σ we want to find. We denote the nearest common ancestor of σ and its k -neighbor σ' with σ_A . The objective is to compute, from the location code of σ , the location code of the nearest common ancestor.

To find σ_A , the hierarchy of pentatopes must be ascended up from σ to σ_A by reversing the path from σ_A to σ . We stop when we identify σ_A . Since we are using a representation based on location codes, the bottom-up retrieval of the nearest common ancestor consists of scanning the bit string in the location code of σ from right to left, dropping the rightmost bit from the bit string at each step. This corresponds to moving to the parent of the current pentatope in the hierarchy. At the end of the process, we obtain the location code of the nearest common ancestor σ_A .

To identify the nearest common ancestor σ_A of σ and σ' , we observe that σ_A is the pentatope which is split by the tetrahedral face f_A containing the k -face f_k of σ . Thus, σ_A is the parent of the ancestor σ^* of σ bounded by face f_A . This does not have to be verified geometrically, but we can decide whether we need to continue the process or stop based on the shape of the current simplex τ (whether it is an e -pentatope, an h -pentatope, a c -pentatope or an s -pentatope) and on the type of neighbor k .

As an example, consider the location code 9010110. Since the depth is six, this location code refers to an s -pentatope. If we want to find neighbor type 4 (face

$v_1v_3v_4v_5$), then we must first find the nearest common ancestor using the previously described right to left scan. Since our neighbor direction forces us to cross face $v_1v_3v_4v_5$, we must look at the parent (901011). Keeping the same neighbor direction means that we must now cross face $v_2v_3v_4v_5$ of the parent. Again, we must look at the next ancestor (90101). Keeping the same neighbor direction means that we must now cross face $v_1v_2v_3v_4$ of the ancestor. Crossing face $v_1v_2v_3v_4$ is our stopping condition, so we stop at 90101. Officially, the nearest common ancestor (9010) is one level up, but we need to know which child contained our input pentatope in order to get the appropriate sibling for the neighbor.

4.3.2 Updating the Location Code

In this step we simply invert the one bit corresponding to the child of the nearest common ancestor σ_A . This works regardless of the original neighbor type which we were trying to find. No further work is necessary, since all neighbors' location codes differ by just this one bit.

If we continue with our example from the previous section, then we know that the nearest common ancestor is 9010. Since 90101 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 90101 is 90100, so the neighbor of 9010110 which shares face $v_1v_3v_4v_5$ is 9010010.

4.3.3 Extensions to the Entire Hypercube

Since we actually have 24 pentatopes as our first decomposition level of the hypercube, we need to make sure that our transitions work between the 24 top level pentatopes. The vertices for these 24 pentatopes have been initially labeled so that they imitate the labels of vertices at lower levels in the decomposition. Note that the labeling of these top 24 pentatopes themselves is not critical since we can simply use table lookup for 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 we are leaving the hypercube at this point, then we need to return an error. Otherwise, we know that a neighbor must exist, so we consider the entire hypercube 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 pentatope, 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 pentatope differ by only one bit. Therefore, we simply select the new top level bits based on a table lookup.

4.4 Constant-Time Neighbor Finding Algorithm

In this Section, we describe how to perform neighbor finding in worst-case constant time. For the sake of simplicity, we consider only the case in which the

input pentatope σ is an h -pentatope. In the case that σ is not an h -pentatope, we just need to move up in the hierarchy by at most four levels: either we find the nearest common ancestor (and, thus, the k -neighbor we are looking for) in a maximum of four steps (changes in level), or we find an h -pentatope, since the four shapes are cyclic on four levels.

We can apply the rules described in Figure 53 to each group of four consecutive bits in the location code of σ by proceeding right to left. We would do bit operations to identify the different bit patterns, but this is still a sequential search which does not achieve a constant time behavior. To this aim, we need to be able to predict the neighbor type we will be looking for in all groups of four bits at the same time, thus avoiding an iterative process.

Current Bits	Neighbor 1	Neighbor 2	Neighbor 3	Neighbor 4	Neighbor 5
0000	0001	Cont 2	Cont 3	0100	0010
0001	0000	Cont 2	Cont 3	0101	Cont 5
0010	0011	Cont 2	Cont 3	Cont 4	0000
0011	0010	Cont 2	Cont 3	Cont 4	Cont 5
0100	0101	Cont 2	1100	0000	Cont 4
0101	0100	Cont 2	1101	0001	0111
0110	0111	Cont 2	1110	Cont 5	Cont 4
0111	0110	Cont 2	1111	Cont 5	0101
1000	1001	Cont 1	Cont 5	Cont 4	1010
1001	1000	Cont 1	Cont 5	Cont 4	Cont 3
1010	1011	Cont 1	Cont 5	1110	1000
1011	1010	Cont 1	Cont 5	1111	Cont 3
1100	1101	Cont 1	0100	Cont 3	Cont 4
1101	1100	Cont 1	0101	Cont 3	1111
1110	1111	Cont 1	0110	1010	Cont 4
1111	1110	Cont 1	0111	1011	1101

Figure 53: The table indicates how to proceed at each level when searching for the neighboring pentatope.

We want to make use of the carry property of addition to find a neighbor

without specifically searching for the nearest common ancestor. In particular, we replace the step-by-step process mentioned previously by an arithmetic operation that takes constant time instead of time proportional to the depth of the tree. The algorithms make use of bit manipulation operations which can be implemented in hardware using a few machine language instructions. Of course, the constant time bound arises because the entire bit string which identifies the path in the location code is assumed to fit in one computer word. This, however, allows us to deal with data sets containing up to 256 points in each of the four dimensions, or over 10^9 total points.

Since our goal is to use bit operations in order to find the nearest common ancestor, we need to identify which bit patterns indicate that the nearest common ancestor is farther up in the tree (beyond the current set of four bits). Being able to identify these patterns in a logical notation allows for a conversion into bit operations that perform the same logical function over the entire location code in a fixed number of operations. This is regardless of the actual length of the location code, since bit operations can be performed over an entire computer word in a single operation.

Looking at the patterns found in Figure 53 we can see that it is possible to determine neighbor type 2 if bit 1 is 1, neighbor type 3 if bit 2 is 1, neighbor type 4 if bit 1 is equal to bit 3, and neighbor type 5 if bit 2 is equal to bit 4. Also note that once the neighbor can be identified, the neighbor type determines which bit needs to change in order to get the location code of the neighbor. Bit 4 changes for neighbor type 1, bit 1 changes for neighbor type 3, bit 2 changes for neighbor type 4, and bit

3 changes for neighbor type 5. These patterns, which are discussed in greater detail in [45], are the basis for the bit operations which lead to the constant-time behavior in the algorithms that follow.

4.4.1 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.

```
procedure NEIGHBOR_1(PENTA);
/* Determine the location code of the neighbor which shares face 1234 of the pentatope with location code PENTA. */
begin
  value pointer location_code PENTA;

  /* Flip the last bit */
  CODE(PENTA) ← XOR(CODE(PENTA), 1);
end;
```

4.4.2 Neighbor Type 2

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 4th ancestor (same shape, double size, sixteen times the volume) depending on which child of the 4th ancestor was needed to reach the input location code. This means that we need a carry if the child of the 4th ancestor was child 0, and no carry

if it was child 1.

```
procedure NEIGHBOR_2(PENTA);
/* Determine the location code of the neighbor which shares face 1235 of the pen-
tatope with location code PENTA. */
begin
  value pointer location_code PENTA;
  path_array TEST,FLIP;

  /* Identify positions where sibling can be determined */
  TEST←AND(CODE(PENTA),POS1MASK);
  /* Use carry to find rightmost sibling position */
  FLIP←COMPLEMENT(TEST)+1;
  /* Clear out everything but the final carry */
  FLIP←AND(FLIP,TEST);
  /* Make sure we adjust to the proper level */
  FLIP←SHIFT_LEFT(FLIP);
  /* Flip the appropriate bit */
  CODE(PENTA)←XOR(CODE(PENTA),FLIP);
end;
```

Notice that the first line of NEIGHBOR_2 locates the positions within the location code where the sibling can be determined because the relevant face is face $v_1v_2v_3$. This result is stored in TEST. Since we want carries where the sibling cannot be determined, we need to complement TEST and then do the addition. To isolate the one bit that needs to be flipped in the location code, we “and” the result of the addition with the value stored in TEST (the positions where we CAN determine the sibling). The bits are offset by one position at this point, so we shift the answer left by one position. Finally, we simply flip the appropriate bit in the location code, by using “xor” between the location code and the current bit mask (which contains only one bit marking the position where we found the sibling).

As an example, let us consider location code 710100110101100100011. We can determine the sibling at any position where the first bit (out of 4) is 1. The rightmost

1 tells us which bit needs to be inverted in order to get the correct neighbor. The sequence is shown in Figure 54.

Step	Operation	Results
0	Example Input	710100110101100100011
1	AND(\$0,POS1MASK)	010000000100000000000
2	COMPLEMENT(\$1)	101111111011111111111
3	ADD_ONE(\$2)	101111111100000000000
4	AND(\$3,\$1)	000000000100000000000
5	SHIFT_LEFT(\$4)	000000001000000000000
6	XOR(\$5,\$0)	710100111101100100011

Figure 54: Example of neighbor type 2.

4.4.3 Neighbor Types 3, 4, and 5

Finding neighbor types 3, 4, or 5 requires that we find the pentatope which shares the appropriate tetrahedral face, and may involve finding any of these three neighbor types at higher levels in the hierarchy if the requested face is contained by face $v_1v_2v_4v_5$, $v_1v_3v_4v_5$, or $v_2v_3v_4v_5$ in the 4th 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, and we also want this process to take into account any changes in neighbor type as we go up in the hierarchy. Therefore, we need an indicator to keep track of which neighbor type we want to find at each level (or at least at every fourth level). As in the 3D case, we will use a “neighbor mask” to store this information.

```
procedure NEIGHBOR_3(PENTA);
/* Determine the location code of the neighbor which shares face 1245 of the pen-
tatope with location code PENTA. */
```

```

begin
  value pointer location_code PENTA;
  path_array TEST,SELECT,FLIP;

  /* Apply all stop tests to entire location code */
  TEST←COMPLEMENT(CODE(PENTA));
  TEST←SHIFT_RIGHT(SHIFT_RIGHT(AND(TEST,POS12MASK)));
  TEST←XOR(TEST,CODE(PENTA));
  /* Get bits related to neighbor type 3 */
  SELECT←AND(MASK(PENTA),POS12MASK);
  SELECT←OR(SELECT,SHIFT_RIGHT(SHIFT_RIGHT(SELECT)));
  /* Select appropriate results from stop tests */
  SELECT←XOR(SELECT,SHIFT_RIGHT(AND(SELECT,POS2MASK)));
  SELECT←COMPLEMENT(OR(SHIFT_RIGHT(SELECT),POS1MASK));
  TEST←AND(TEST,SELECT);
  /* Use carry to find rightmost stopping position */
  FLIP←COMPLEMENT(TEST)+1;
  /* Clear out everything but the final carry */
  FLIP←AND(FLIP,TEST);
  /* Make sure we adjust to the proper level */
  FLIP←SHIFT_LEFT(FLIP);
  /* Flip the appropriate bit */
  CODE(PENTA)←XOR(CODE(PENTA),FLIP);
end;

```

The first step in finding neighbor types 3, 4, or 5, is identifying where (i.e., at what level in the location code) we can determine the neighbor. We want to construct the mask TEST so that it marks the locations where the neighbor can be determined. Since neighbors are determined before we reach the 4th ancestor (otherwise, we continue upwards in the hierarchy), we will examine the bits in sets of four, where the leftmost (or most significant) bit is called bit 1, the next bit is called bit 2, the next bit is called bit 3, and the rightmost (or least significant) bit is called bit 4.

There are three stop cases to consider, one for each of the three neighbor types. In Figure 53 we can see that neighbor type 3 stops with an answer if bit 2

is 1. Neighbor type 4 stops with an answer if bit 1 is equal to bit 3. And, neighbor type 5 stops with an answer if bit 2 is equal to bit 4. The mask TEST is constructed based on these patterns. The first three lines in the code compute and store the results of all three tests simultaneously.

The SELECT mask is constructed to select the appropriate result (from the three tests) for each level in the hierarchy. The “and” between TEST and SELECT keeps only the relevant stop results. We complement TEST before the addition because we want a carry to occur whenever we cannot identify the neighbor at a given level. The carry continues 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 flipped in the location code, we “and” the result of the addition with the value stored in TEST (the positions where we CAN determine the sibling). The bits are offset by one position at this point, so we shift the answer left by one position. Finally, we simply flip the appropriate bit in the location code, by using “xor” between the location code and the current bit mask (which contains only one bit marking the position where we found the sibling).

As an example, let us consider location code 710100110101100100011. We can determine the sibling at any position where our stop tests give a result of 1. The rightmost 1 tells us which bit needs to be inverted in order to get the correct neighbor. The sequence is shown in Figure 55.

The procedures for neighbor types 4 and 5 are almost identical to neighbor type 3. The only difference is the part of the code which gets the relevant bits from the neighbor mask. The complete code (for both neighbor types 4 and 5) is given

Step	Operation	Results
0	Example Input	710100110101100100011
1	COMPLEMENT(\$0)	001011001010011011100
2	AND(\$1,POS12MASK)	001001000010011001100
3	SHIFT_RIGHT(\$2) x 2	000010010000100110011
4	XOR(\$3,\$0)	710110100101000010000
5	Neighbor Mask	010111110011001100110
6	AND(\$5,POS12MASK)	010001100010001000100
7	SHIFT_RIGHT(\$6) x 2	000100011000100010001
8	OR(\$7,\$6)	010101111010101010101
9	AND(\$8,POS2MASK)	000000100010001000100
10	SHIFT_RIGHT(\$9)	000000010001000100010
11	XOR(\$10,\$8)	010101101011101110111
12	SHIFT_RIGHT(\$11)	001010110101110111011
13	OR(\$12,POS1MASK)	011011110101110111011
14	COMPLEMENT(\$13)	000100001010001000100
15	AND(\$14,\$4)	000100000000000000000
16	COMPLEMENT(\$15)	111011111111111111111
17	ADD_ONE(\$16)	111100000000000000000
18	AND(\$17,\$15)	000100000000000000000
19	SHIFT_LEFT(\$18)	001000000000000000000
20	XOR(\$19,\$0)	711100110101100100011

Figure 55: Example of neighbor type 3.

below.

```

procedure NEIGHBOR_4(PENTA);
/* Determine the location code of the neighbor which shares face 1345 of the pen-
tatope with location code PENTA. */
begin
  value pointer location_code PENTA;
  path_array TEST,SELECT,FLIP;

  /* Apply all stop tests to entire location code */
  TEST←COMPLEMENT(CODE(PENTA));
  TEST←SHIFT_RIGHT(SHIFT_RIGHT(AND(TEST,POS12MASK)));
  TEST←XOR(TEST,CODE(PENTA));
  /* Get bits related to neighbor type 4 */
  SELECT←AND(MASK(PENTA),POS34MASK);
  SELECT←OR(SELECT,SHIFT_LEFT(SHIFT_LEFT(SELECT)));
  /* Select appropriate results from stop tests */
  SELECT←XOR(SELECT,SHIFT_RIGHT(AND(SELECT,POS2MASK)));
  SELECT←COMPLEMENT(OR(SHIFT_RIGHT(SELECT),POS1MASK));

```



```

TEST←AND(TEST,SELECT);
/* Use carry to find rightmost stopping position */
FLIP←COMPLEMENT(TEST)+1;
/* Clear out everything but the final carry */
FLIP←AND(FLIP,TEST);
/* Make sure we adjust to the proper level */
FLIP←SHIFT_LEFT(FLIP);
/* Flip the appropriate bit */
CODE(PENTA)←XOR(CODE(PENTA),FLIP);
end;

procedure NEIGHBOR_5(PENTA);
/* Determine the location code of the neighbor which shares face 2345 of the pen-
tatope with location code PENTA. */
begin
    value pointer location_code PENTA;
    path_array TEST,SELECT,FLIP;

    /* Apply all stop tests to entire location code */
    TEST←COMPLEMENT(CODE(PENTA));
    TEST←SHIFT_RIGHT(SHIFT_RIGHT(AND(TEST,POS12MASK)));
    TEST←XOR(TEST,CODE(PENTA));
    /* Get bits related to neighbor type 5 */
    SELECT←SHIFT_LEFT(SHIFT_LEFT(AND(MASK(PENTA),POS34MASK)));
    SELECT←XOR(SELECT,AND(MASK(PENTA),POS12MASK));
    SELECT←OR(SELECT,SHIFT_RIGHT(SHIFT_RIGHT(SELECT)));
    /* Select appropriate results from stop tests */
    SELECT←XOR(SELECT,SHIFT_RIGHT(AND(SELECT,POS2MASK)));
    SELECT←COMPLEMENT(OR(SHIFT_RIGHT(SELECT),POS1MASK));
    TEST←AND(TEST,SELECT);
    /* Use carry to find rightmost stopping position */
    FLIP←COMPLEMENT(TEST)+1;
    /* Clear out everything but the final carry */
    FLIP←AND(FLIP,TEST);
    /* Make sure we adjust to the proper level */
    FLIP←SHIFT_LEFT(FLIP);
    /* Flip the appropriate bit */
    CODE(PENTA)←XOR(CODE(PENTA),FLIP);
end;

```

4.4.4 Updating the Neighbor Mask

In order to perform neighbor finding in constant-time, we not only need access to the neighbor type information in the neighbor mask, but we also need to be able to update the neighbor mask itself in constant time. This is accomplished by updating specific bit positions in the neighbor mask based on the bit that changes in the location code.

From Figure 53 we can verify that the following is true:

- If the first two bits are 00, neighbor type 3 continues as neighbor type 3, neighbor type 4 continues as neighbor type 4, and neighbor type 5 continues as neighbor type 5. Therefore, 00 means no change in our neighbor state.
- If the first two bits are 01, neighbor type 3 never continues (assume type 3), neighbor type 4 continues as neighbor type 5, and neighbor type 5 continues as neighbor type 4. Therefore, 01 means we swap 4 and 5 in our neighbor state.
- If the first two bits are 10, neighbor type 3 continues as neighbor type 5, neighbor type 4 continues as neighbor type 4, and neighbor type 5 continues as neighbor type 3. Therefore, 10 means we swap 3 and 5 in our neighbor state.
- If the first two bits are 11, neighbor type 3 never continues (assume type 5), neighbor type 4 continues as neighbor type 3, and neighbor type 5 continues as neighbor type 4. Therefore, 11 means we rotate types in our neighbor state.

If we consider all possible changes in the first two bits (both to and from each of the four options), we get a predictable set of related swaps which are shown in Figure 56.

From	To 00	To 01	To 10	To 11
00	No change	Swap 4 and 5	Swap 3 and 5	Not possible
01	Swap 4 and 5	No change	Not possible	Swap 3 and 5
10	Swap 3 and 5	Not possible	No change	Swap 3 and 4
11	Not possible	Swap 3 and 5	Swap 3 and 4	No change

Figure 56: Swaps performed as a result of various bit changes.

The positions of the two values that swap tell us which positions should be updated in the remaining parts of the neighbor mask. Since there are effectively three bit pairs, one for each of the three possible neighbor types, there are only three swap operations to consider (swap positions 1 and 2, swap positions 1 and 3, and swap positions 2 and 3). Swapping fixed positions can be accomplished in constant time. Thus, we can simply apply one of three constant-time swap operations in order to update the neighbor mask in constant time. Notice that swaps involving position 3 are really just a simple exclusive-or operation, since position 3 is only implied (by process of elimination) and not actually stored in the neighbor mask.

```

procedure SWAP_1_2(PENTA,FLIP);
/* Swap positions 1 and 2 in neighbor mask of PENTA. */
begin
  value pointer location_code PENTA;
  value path_array FLIP;
  path_array TEMP;

  /* Modify FLIP so it can be used as a mask */
  FLIP←SHIFT_LEFT(-FLIP);
  /* Swap first two bits with last two bits */
  TEMP←SHIFT_LEFT(SHIFT_LEFT(AND(MASK(PENTA),POS34MASK)));
  TEMP←OR(TEMP,SHIFT_RIGHT(SHIFT_RIGHT(AND(MASK(PENTA),POS12MASK))));

```

```

    /* Clear out bits left of FLIP location */
    MASK(PENTA) ← AND(COMPLEMENT(FLIP));
    /* Store swapped bits into neighbor mask */
    MASK(PENTA) ← OR(AND(TEMP, FLIP), MASK(PENTA));
end;

procedure SWAP_1_3(PENTA, FLIP);
/* Swap positions 1 and 3 in neighbor mask of PENTA. */
begin
    value pointer location_code PENTA;
    value path_array FLIP;
    path_array TEMP;

    /* Modify FLIP so it can be used as a mask */
    FLIP ← SHIFT_LEFT(-FLIP);
    /* Get last two bits to be used for update step */
    TEMP ← AND(AND(MASK(PENTA), POS34MASK), FLIP);
    TEMP ← SHIFT_LEFT(SHIFT_LEFT(TEMP));
    /* Swap first two bits with implied third pair of bits */
    MASK(PENTA) ← XOR(MASK(PENTA), TEMP);
end;

procedure SWAP_2_3(PENTA, FLIP);
/* Swap positions 2 and 3 in neighbor mask of PENTA. */
begin
    value pointer location_code PENTA;
    value path_array FLIP;
    path_array TEMP;

    /* Modify FLIP so it can be used as a mask */
    FLIP ← SHIFT_LEFT(-FLIP);
    /* Get first two bits to be used for update step */
    TEMP ← AND(AND(MASK(PENTA), POS12MASK), FLIP);
    TEMP ← SHIFT_RIGHT(SHIFT_RIGHT(TEMP));
    /* Swap last two bits with implied third pair of bits */
    MASK(PENTA) ← XOR(MASK(PENTA), TEMP);
end;

```

4.4.5 Transitions Across the 24 Top Level Pentatopes

Transitions between the 24 top level pentatopes 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.

Transitions at the top level are really no different than our previous discussion. We simply pick a new top level pentatope based on our table lookup.

4.5 Clusters of Pentatopes in an HP

The bisection rule generates nested meshes, that in general are not conforming. To produce a conforming mesh, when applying pentatope bisection, all pentatopes that share a common edge with the pentatope that is being split, must be split at the same time to guarantee consistency. We call any set of pentatopes which share their longest edge a *cluster*. There are four types of clusters based on the four choices of orientation of the edge that is bisected, that we call *h-clusters*, *c-clusters*, *s-clusters*, and *e-clusters*, respectively. These four clusters correspond to the four geometrically similar simplicial shapes, and each cluster will contain only pentatopes with the same shape:

- an *h-cluster* is a hypercube formed by 24 *h*-pentatopes (the initial domain subdivision is an *h-cluster*), all sharing the diagonal of a hypercube as their longest edge, with 36 tetrahedral and 14 triangular faces.
- a *c-cluster* is formed by 12 *c*-pentatopes, all sharing the diagonal of a cube as their longest edge (which thus lies on a hyper-plane parallel to one of the four coordinate hyper-planes), with 18 tetrahedral and 8 triangular faces.
- an *s-cluster* is formed by 16 *s*-pentatopes, all sharing the diagonal of a square

as their longest edge (which thus lies on a plane parallel to one of the six coordinate planes), with 24 tetrahedral and 10 triangular faces.

- an *e-cluster* is formed by 48 *e*-pentatopes, all sharing an edge aligned with one of the four coordinate axes as their longest edge, with 72 tetrahedral and 26 triangular faces.

To produce a nested conforming subdivision, we need to be able to compute efficiently, for each pentatope σ that must be split along an edge e , all pentatopes which belong to the same cluster as σ . Given a pentatope σ and edge e of σ , the problem consists of computing the pentatopes sharing edge e with σ which form a cluster. This can be performed by traversing the pentatopes incident at e and moving from one pentatope σ to a pentatope adjacent to σ along a tetrahedral face, until all pentatopes in the cluster are found. Constant-time neighbor finding is used to locate adjacent pentatopes and thus to compute the necessary clusters.

4.6 A Depth First Algorithm for Selective Refinement

The depth-first algorithm is a standard refinement algorithm (see, for instance, [50]). It starts from the initial mesh, consisting of the 24 top-level pentatopes which subdivide the hypercube, and initializes the currently extracted mesh (that we call the *current mesh*) with them. For any pentatope t , that does not satisfy the error requirements, we split the cluster c of pentatopes which share their longest edge with t . If a pentatope t' in c does not exist in the current mesh, then we split the cluster associated with the parent of t' . This is applied recursively in order to

guarantee a conforming mesh. The process continues until all pentatopes in the current mesh satisfy the error requirements.

A pseudo-code description of the depth-first algorithm is given below. Predicate $\text{EXIST}(t)$ returns the value *true* if pentatope t is part of the current mesh, and the value *false* otherwise. Function $\text{PARENT}(t)$ deletes the last bit from the location code of t , thus returning the parent of t . Functions $\text{CHILD}_0(t)$ and $\text{CHILD}_1(t)$ add a 0-bit and a 1-bit to the end of the location code of t , respectively, thus returning the first and the second child of t , respectively. Function $\text{SPLIT}(t)$ replaces t in the current mesh with $\text{CHILD}_0(t)$ and $\text{CHILD}_1(t)$. Function $\text{CLUSTER}(t)$ returns the set of all pentatopes sharing the splitting edge with pentatope t . This set can be found by using the neighbor finding described previously.

```

procedure SPLIT_CLUSTER(PENTA);
/* Split all pentatopes within the same cluster as PENTA. This includes splitting
   PENTA itself. */
begin
  value pointer location_code PENTA;
  pointer location_code NEIGHBOR;

  for NEIGHBOR in {CLUSTER(PENTA)} do
    begin
      if not(EXIST(NEIGHBOR)) then SPLIT_CLUSTER(PARENT(NEIGHBOR));

      SPLIT(NEIGHBOR);
    end;
  end;

procedure CHECK_SPLIT(PENTA);
/* Check if PENTA satisfies the error requirement and split if necessary. */
begin
  value pointer location_code PENTA;

  if FAIL_ERROR_REQ(PENTA) then
    begin
      SPLIT_CLUSTER(PENTA);
    end;

```

```

        CHECK_SPLIT(CHILD_0(PENTA));
        CHECK_SPLIT(CHILD_1(PENTA));
    end;
end;

procedure EXTRACT_MESH();
/* Extract a conforming mesh where all pentatopes satisfy the error requirement.
*/
begin
    location_code PENTA;

    for PENTA in {'0', '1', ..., '23'} do
        begin
            CHECK_SPLIT(PENTA);
        end;
    end;
end;

```

It can be easily seen that the worst-case time complexity of the algorithm is $O(k)$, where k is the number of pentatopes in the hierarchy. Since the pentatopes are only split in SPLIT_CLUSTER if forced by our consistency constraint, and SPLIT_CLUSTER is only called if one of the pentatopes is beyond the error threshold, the algorithm never splits a pentatope unless it is necessary, and thus, it generates the minimum number of pentatopes which are required to satisfy the error requirements.

4.7 Experimental Results

In this Section, we show images and performance statistics obtained by performing multi-resolution queries on an HP. We have used subsets of the Ritchmyer Meshkov Instability data set from the ASCI team at Lawrence Livermore National Laboratories. This represents a simulation in which two gases are initially separated

by a membrane pushed against a wire mesh. The complete data set consists of 270 time steps, and each time step is simulated over a 2048x2048x1920 grid. Also, we have performed experiments by generating a 4D data set from the Buckyball data set. This 4D data set was generated by using the field values inside each 4x4x4 subgrid as the different values in the fourth dimension for the point corresponding to the subgrid. In both cases, the test data is within a 32x32x32x32 grid, resulting in 1,185,921 total points and 25,165,824 total pentatopes at maximum resolution.

We show results on multi-resolution queries that extract a 4D mesh according to the following LOD criteria:

- *Uniform LOD*: extraction of a mesh satisfying a constant error threshold; in our experiments, the threshold varies from zero to the maximum error value.
- *Variable LOD in a box*: extraction of a mesh with an error below a given threshold inside a 4D box and no error constraint outside it; in our experiments, the box is at a fixed position, and the threshold inside the box varies from zero to the maximum error.
- *Variable LOD based on a field value*: extraction of a mesh with an error below a certain threshold on the pentatopes intersecting the isosurface of the specified field value, and no error constraint elsewhere.

Figures 57–59 and 60–62 show meshes extracted by each of the queries for the 1% error threshold value for the Buckyball and Ritchmyer Meshkov Instability data sets, respectively. These meshes have been obtained by cutting the extracted 4D

mesh through a hyperplane perpendicular to the time axis (but our algorithm works for a cutting hyperplane parallel to any of the coordinate hyperplanes).

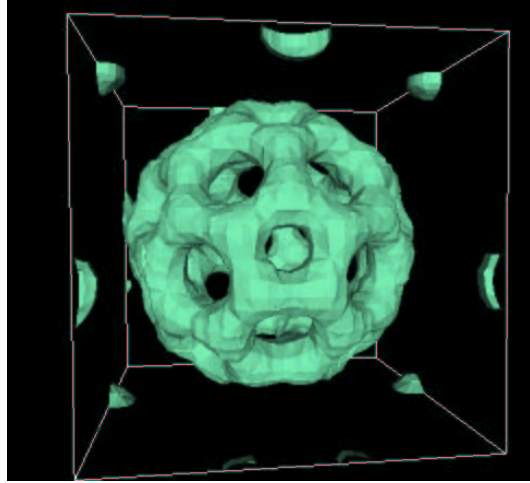


Figure 57: Uniform LOD extraction from the Buckyball data set: error threshold equal to 1% of the field range. The isosurface for timestep 4 and field value 100 is shown.

Sample results for each of the three queries with error threshold values of 0, 1, 5, and 10% of the field range are given in Figure 63 for the modified Buckyball data set and Figure 64 for the Ritchmyer Meshkov Instability data set. As expected, using higher error thresholds or increasing the selectivity for the query reduces the number of pentatopes which must be processed, and results in a lower level of detail in the corresponding time slice. It should also be noted that the selectivity can be used to focus on the time slice itself. In other words, we can extract a 4D mesh at a certain resolution only in the specified time slice. This can be seen in Figures 63 and 64 where the numbers of pentatopes are consistently less than 10% of the mesh size at full resolution (25,165,824). Looking at the uniform resolution we find that when using a 1% error value threshold restricted to the time slice of interest leads to a reduction in the number of pentatopes from 13,359,032 to 1,384,610 (as shown

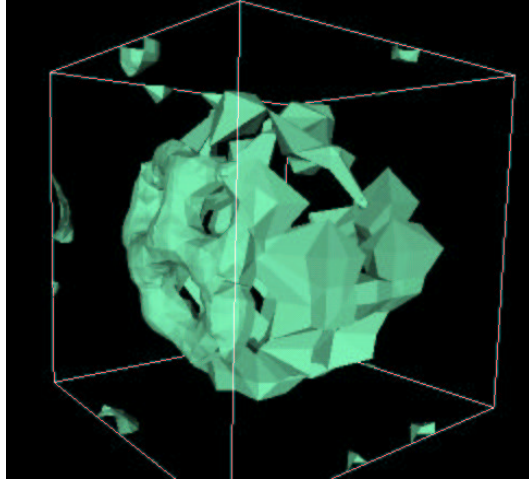


Figure 58: Variable LOD based on a region of space in the Buckyball data set: error threshold equal to 1% of the field range within the selected area, and arbitrarily large elsewhere. The isosurface for timestep 4 and field value 100 is shown.

in Figure 63) with no change in the number of tetrahedra for the time slice, which is 114,268 in both cases. It is interesting to note that the field value query allows a further reduction of the size.

For comparison purposes, we have also implemented a technique for extracting conforming meshes from a hierarchy of pentatopes based on *error saturation*, as discussed in [54, 79]. First, all pentatopes belonging to the same cluster are assigned the same error value, which is equal to the maximum of their original error values. Moreover, the approximation error associated with each pentatope is saturated to be greater than or equal to the error associated with its children. This implies that, during mesh extraction, if a pentatope is refined, then all pentatopes belonging to the same cluster are refined.

Our experimental comparisons, that we have performed on the basis of LOD queries at a uniform resolution, have not shown a major increase in the number of

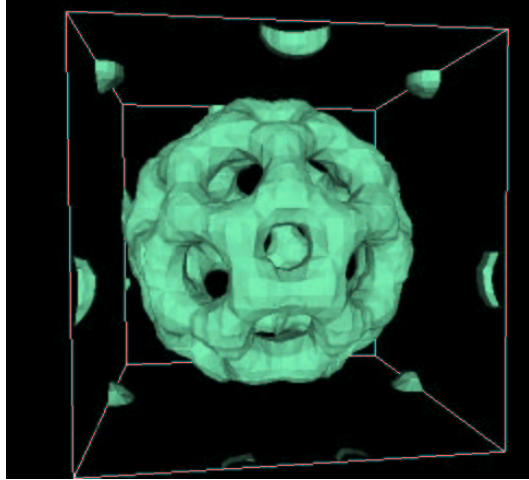


Figure 59: Variable LOD based on field value in the Buckyball data set: error threshold equal to 1% of the field range on the tetrahedra intersected by the isosurface with field value 100, and an error threshold arbitrarily large elsewhere. The isosurface for timestep 4 and field value 100 is shown. The number of pentatopes in the resulting 4D mesh is 70% of the number of pentatopes in the mesh obtained in the uniform LOD extraction (see Figure 57).

pentatopes in meshes extracted from a saturated HP. The resulting meshes have less than 1% more pentatopes than those extracted with our method. Regardless, the computation times of our depth-first algorithm are the same as those of the corresponding algorithm on a saturated HP (which simply performs a top-down traversal of the hierarchy without any neighbor finding computation).

Another method which uses a pointerless representation of hierarchical regular simplicial meshes is Atalay and Mount [2]. They extend the techniques of Hebert [34] by introducing a variation on the labeling scheme called an LPT code, and present rules to compute the neighbors of a given simplex efficiently. Their system uses the same bisection technique as Maubach [50] and is designed to work in arbitrary dimensions. Of course, as Atalay and Mount point out, Hebert's addressing scheme could already be generalized to higher dimensions. In fact, constant-time algorithms

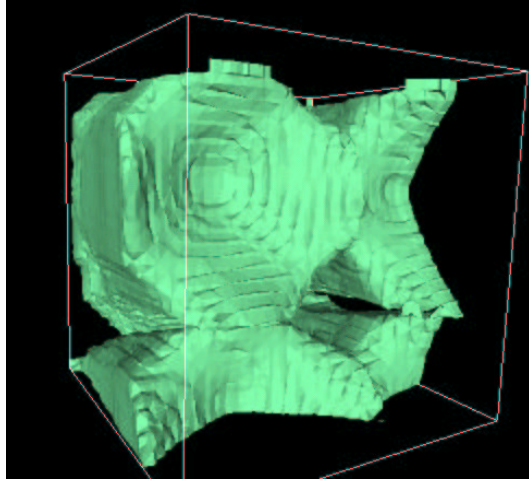


Figure 60: Uniform LOD extraction from the Ritchmyer data set: error threshold equal to 1% of the field range. The isosurface for timestep 4 and field value 100 is shown.

could be achieved by making use of bit operations on the symbolic location codes. Regardless, table lookups are important for efficient neighbor finding within the hierarchy, as is the case in [2]. While this may not technically make it impossible to create algorithms which are readily generalizable to higher dimensions, it does seem to make such a generalization fairly impractical. In any case, the vertex ordering of Atalay and Mount [2] is really just a generalization of the vertex ordering used by Hebert [34]. Our hierarchy of pentatopes uses a completely different strategy. We don't use symbolic location codes or LPT codes, which encode the *level*, *permutation*, and *translation* for each simplex relative to some other reference simplex. Instead, even with location codes which are based solely on the path to the pentatope within the hierarchy, we are able to compute neighbors in worst-case constant time.

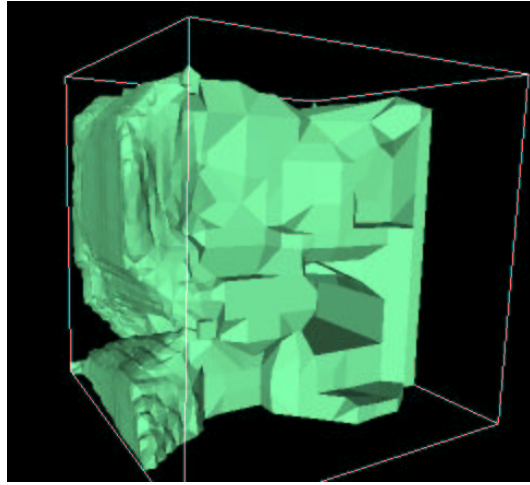


Figure 61: Variable LOD based on a region of space in the Ritchmyer data set: error threshold equal to 1% of the field range within the selected area, and arbitrarily large elsewhere. The isosurface for timestep 4 and field value 100 is shown.

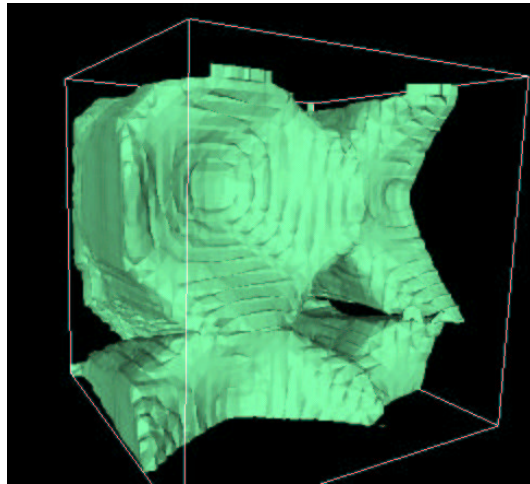


Figure 62: Variable LOD based on field value in the Ritchmyer data set: error threshold equal to 1% of the field range on the tetrahedra intersected by the isosurface with field value 100, and an error threshold arbitrarily large elsewhere. The isosurface for timestep 4 and field value 100 is shown. The number of pentatopes in the resulting 4D mesh is 92% of the number of pentatopes in the mesh obtained in the uniform LOD extraction (see Figure 60).

Modified Buckyball					
	error (% of field range)	4D mesh		3D time slice	
		pentatopes	% pentatopes	tetrahedra	% tetrahedra
Uniform	0.0	2,419,152	9.6%	196,608	100.0%
	1.0	1,384,610	5.5%	114,268	58.1%
	5.0	772,086	3.1%	60,968	31.0%
	10.0	493,358	2.0%	36,502	18.6%
Selected Box	0.0	421,072	1.7%	34,912	17.8%
	1.0	207,194	0.8%	16,986	8.6%
	5.0	102,736	0.4%	8,510	4.3%
	10.0	66,718	0.3%	5,878	3.0%
Field Value	0.0	1,139,996	4.5%	87,300	44.4%
	1.0	975,740	3.9%	78,538	39.9%
	5.0	712,942	2.8%	55,944	28.4%
	10.0	492,180	2.0%	36,422	18.5%

Figure 63: Number of pentatopes in the meshes along with the percentage with respect to the number of pentatopes at full resolution, and the number of tetrahedra in the time slice with value 4 along with the percentage with respect to the number of tetrahedra at full resolution.

Ritchmyer Meshkov					
	error (% of field range)	4D mesh		3D time slice	
		pentatopes	% pentatopes	tetrahedra	% tetrahedra
Uniform	0.0	2,419,152	9.6%	196,608	100.0%
	1.0	891,704	3.5%	81,368	41.4%
	5.0	738,622	2.9%	67,576	34.4%
	10.0	636,380	2.5%	57,364	29.2%
Selected Box	0.0	421,072	1.7%	34,912	17.8%
	1.0	164,670	0.7%	16,156	8.2%
	5.0	140,976	0.6%	14,036	7.1%
	10.0	125,430	0.5%	12,436	6.3%
Field Value	0.0	1,141,396	4.5%	88,006	44.8%
	1.0	821,822	3.3%	74,398	37.8%
	5.0	704,778	2.8%	64,374	32.7%
	10.0	625,996	2.5%	56,394	28.7%

Figure 64: Number of pentatopes in the meshes along with the percentage with respect to the number of pentatopes at full resolution, and the number of tetrahedra in the time slice with value 4 along with the percentage with respect to the number of tetrahedra at full resolution.

Chapter 5

Conclusions

We have described a triangle coding scheme that provides a new and worst-case constant time way 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. 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.

Our neighbor finding technique has a natural application [40] 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 multi-resolution surface capability [19, 36]. 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 [40].

We have also considered the natural and logical extension of these methods to three-dimensional data, where the basic shapes are now tetrahedra instead of triangles (e.g., [61, 69]). We have considered a multi-resolution representation of a 3D scalar field based on hierarchical tetrahedral meshes generated by tetrahedron bisection, that we call a Hierarchy of Tetrahedra (HT). We have developed a worst-case constant time neighbor finding algorithm for hierarchies of tetrahedra. We have proposed an implementation of an HT, and we have discussed how to extract conforming meshes from an HT by using such an algorithm. We have also performed theoretical and experimental comparisons with a LOD model based on multiresolution unstructured tetrahedral meshes [11].

For four-dimensional data, we have proposed a multi-resolution representation of the scalar field based on a recursive decomposition of its hypercubic domain into a hierarchy of nested pentatopes, called a Hierarchy of Pentatopes (HP), generated by bisecting each pentatope along its longest edge. To this aim, we have considered the problem of extracting conforming nested meshes from a hierarchy of pentatopes so as to avoid discontinuities in the corresponding approximation of the associated scalar field.

Generating conforming nested meshes requires computing clusters of pentatopes, which must be split at the same time. Clusters are extracted from the hierarchy by finding the neighbors of a pentatope along one of its five tetrahedral

faces. We have developed a labeling technique for nested pentatopes which enables us to identify a pentatope through its location code. We know how neighbors can be extracted by manipulating location codes, and we have neighbor finding algorithms which work in worst-case constant time. The constant-time behavior is achieved by using bit manipulation operations.

We have experimented with queries on a time-varying scalar field at different resolutions, and we have shown how such queries can be efficiently answered based on the multi-resolution model we have proposed. We have shown that a multi-resolution model is an effective and efficient tool for analyzing and visualizing time-varying volume data sets. The major ingredient for extracting topologically consistent simplified representations from a hierarchy of pentatopes is a neighbor finding algorithm based on location codes and arithmetic bit manipulation.

There are several challenging issues related to this work. When the field values are uniform in large areas, or simply not available, we need to store a variable-resolution hierarchy of pentatopes, in which not all levels will be present, as, for instance, in a quadtree or an octree. Thus, a hierarchy of pentatopes can be encoded by associating with each pentatope its location code and its error value, and the resulting sorting sequence can be stored in a B-tree or in a hash table. An efficient representation for variable-resolution pentatopic hierarchies is also the basis for developing simplicial multi-resolution representations of implicit static and dynamic shapes described through adaptive 4D distance fields. Modeling operations can be efficiently performed by exploiting our efficient neighbor finding algorithm. Finally, we can compute both variable-resolution (according to an LOD criterion)

and multi-resolution representations of interval volumes between pairs of isosurfaces of a 3D scalar field by generating a multi-resolution model of the corresponding 4D field (defined in [6, 7]) as a hierarchy of pentatopes.

BIBLIOGRAPHY

- [1] D. J. Abel. A B^+ -tree structure for large quadtrees. *Computer Vision, Graphics, and Image Processing*, 27(1):19–31, July 1984.
- [2] F. Atalay and D. Mount. Pointerless implementation of hierarchical simplicial meshes and efficient neighbor finding in arbitrary dimensions. In *Proceedings of the 13th International Meshing Roundtable*, pages 15–26, Williamsburg, VA, September 2004.
- [3] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. In R. Stepleman, M. Carver, R. Peskin, W. F. Ames, and R. Vichnevetsky, editors, *Scientific Computing, IMACS Transactions on Scientific Computation*, volume 1, pages 3–17. North-Holland, Amsterdam, The Netherlands, 1983.
- [4] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. In *Proceedings of the 31st IEEE Annual Symposium on Foundations of Computer Science*, pages 231–241, St. Louis, MO, October 1990.
- [5] P. Bhaniramka, R. Wenger, and R. Crawfis. Isosurfacing in higher dimensions. In *Proceedings IEEE Visualization 2000*, pages 267–273, October 2000.
- [6] P. Bhaniramka, R. Wenger, and R. Crawfis. Isosurface construction in any dimension using convex hulls. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):130–141, 2004.
- [7] P. Bhaniramka, C. Zhang, D. Xue, R. Crawfis, and R. Wenger. Volume interval segmentation and rendering. In *Proceedings Volume Visualization Symposium*, 2004.
- [8] P. Cignoni, L. De Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):29–45, 2004.
- [9] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.
- [10] A. da Silva and H. Duarte-Ramos. A progressive trimming approach to space decomposition. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, volume 2, pages 951–960, Zurich, Switzerland, July 1990.
- [11] E. Danovaro, L. De Floriani, M. Lee, and H. Samet. Multiresolution tetrahedral meshes: an analysis and a comparison. In *Proceedings of the International Conference on Shape Modeling 2002*, pages 83–91, Banff, Canada, May 2002.

- [12] L. De Floriani and M. Lee. Selective refinement on nested tetrahedral meshes. In G. Brunett, B. Hamann, and H. Mueller, editors, *Geometric Modeling for Scientific Visualization*. Springer Verlag, 2003.
- [13] L. De Floriani and P. Magillo. Multi-resolution mesh representation: Models and data structures. In M. Floater, A. Iske, and E. Quak, editors, *Principles of Multi-resolution in Geometric Modeling*, Lecture Notes in Mathematics, Springer Verlag, Berlin (D), 2002.
- [14] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization '97*, pages 103–110, Phoenix, AZ, Oct 1997.
- [15] L. De Floriani, P. Marzano, and E. Puppo. Multiresolution models for topographic surface description. *The Visual Computer*, 12(7):317–345, August 1996.
- [16] 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, October 1997.
- [17] G. Dutton. Geodesic modelling of planetary relief. *Cartographica*, 21(2&3):188–207, Summer & Autumn 1984.
- [18] G. Dutton. Locational properties of quaternary triangular meshes. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, volume 2, pages 901–910, Zurich, Switzerland, July 1990.
- [19] N. Dyn, D. Levin, and J. A. Gregory. A Butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9:160–169, April 1990.
- [20] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of the SIGGRAPH'95 Conference*, pages 173–182, Los Angeles, August 1995.
- [21] D. Eppstein. Approximating the minimum weight triangulation. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 48–57, Orlando, FL, January 1992.
- [22] W. Evans, D. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 2001.
- [23] G. Fekete. Rendering and managing spherical data with sphere quadtrees. In A. Kaufman, editor, *Proceedings IEEE Visualization'90*, pages 176–186, San Francisco, October 1990.

- [24] 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 TR-1355.
- [25] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, April 1986.
- [26] I. Fujishiro, Y. Maeda, H. Sato, and Y. Takeshima. Volumetric data exploration using interval volume. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):144–155, 1996.
- [27] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.
- [28] T. Gerstner and M. Rumpf. Multiresolutional parallel isosurface extraction based on tetrahedral bisection. In *Proceedings 1999 Symposium on Volume Visualization*, 1999.
- [29] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.
- [30] M. F. Goodchild and S. Yang. A hierarchical spatial data structure for global geographic information systems. *CVGIP: Graphical Models and Image Understanding*, 54(1):31–44, January 1992.
- [31] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proceedings IEEE Visualization 2002*, Boston, MA, October 2002.
- [32] G. Greiner and R. Grosso. Hierarchical tetrahedral-octahedral subdivision for volume visualization. *The Visual Computer*, 16:357–369, 2000.
- [33] 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*, pages 387–394, Phoenix, AZ, October 1997.
- [34] D. J. Hebert. Symbolic local refinement of tetrahedral grids. *Journal of Symbolic Computation*, 17(5):457–472, May 1994.
- [35] H. Hoppe. View-dependent refinement of progressive meshes. In *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH'97)*, pages 189–198, Los Angeles, August 1997.
- [36] H. Hoppe, T. De Rose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. Piecewise smooth surface reconstruction. In *Proceedings of the SIGGRAPH'94 Conference*, pages 295–302, Orlando, FL, July 1994.

- [37] G. M. Hunter. *Efficient computation and data structures for graphics*. PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.
- [38] L. Ibarria, P. Linstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n -dimensional scalar fields. *Computer Graphics Forum*, 22(3), 2003.
- [39] G. Ji, H. W. Shen, and R. Wenger. Volume tracking using higher dimensional isosurfacing. In G. Turk, J. van Wijk, and R. Moorhead, editors, *Proceedings IEEE Visualization 2003*, pages 209–216, October 2003.
- [40] S. Junkins. Constant time neighbor finding for subdivision surfaces. Technical report, Intel Architecture Labs, Hillsboro, OR, May 1999.
- [41] M. R. Kaplan. Space-tracing: a constant time ray-tracer. Tutorial Notes of the ACM SIGGRAPH Conference on the Uses of Spatial Coherence in Ray-Tracing, San Francisco, July 1985.
- [42] A. Kela, R. Perucchio, and H. Voelcker. Toward automatic finite element analysis. *Computers in Mechanical Engineering*, 5(1):57–71, July 1986.
- [43] A. Klinger. Patterns and search statistics. In J. S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 303–337. Academic Press, New York, 1971.
- [44] M. Lee, L. De Floriani, and H. Samet. Constant-time neighbor finding in hierarchical tetrahedral meshes. In *Proceedings of the International Conference on Shape Modeling & Applications*, pages 286–295, Genova, Italy, May 2001.
- [45] M. Lee, L. De Floriani, and H. Samet. Constant-time navigation in four-dimensional nested simplicial meshes. In *Proceedings of the International Conference on Shape Modeling 2004*, pages 221–230, Genova, Italy, June 2004.
- [46] M. Lee and H. Samet. Traversing the triangle elements of an icosahedral spherical representation in constant-time. In T. K. Poiker and N. Chrisman, editors, *Proceedings of the 8th International Symposium on Spatial Data Handling*, pages 22–33, GIS Lab, Department of Geography, Simon Fraser University, Burnaby, British Columbia, Canada, July 1998. International Geographical Union, Geographic Information Science Study Group.
- [47] M. Lee and H. Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Transactions on Graphics*, 19(2):79–121, April 2000.
- [48] 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, August 1996.

- [49] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proceedings IEEE Visualization 2001*, pages 363–370, San Diego, CA, October 2001.
- [50] J. M. Maubach. Local bisection refinement for n -simplicial grids generated by reflection. *SIAM Journal on Scientific Computing*, 16(1):210–227, January 1995.
- [51] N. Max. Consistent subdivision of convex polyhedra into tetrahedra. *Journal of Graphics Tools*, 6(3):29–36, 2002.
- [52] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. , IBM Ltd., Ottawa, Canada, 1966.
- [53] G. M. Nielson and J. Sung. Interval volume tetrahedralization. In *Proceedings IEEE Visualization '97*, pages 221–228, 1997.
- [54] M. Ohlberger and M. Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 56(4):365–385, 1997.
- [55] M. Ohlberger and M. Rumpf. Adaptive projection operators in multiresolution scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):74–93, 1999.
- [56] E. J. Otoo and H. Zhu. Indexing on spherical surfaces using semi-quadcodes. In D. Abel and B. C. Ooi, editors, *Advances in Spatial Databases — 3rd International Symposium, SSD'93*, pages 510–529, Singapore, June 1993.
- [57] 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, October 1998.
- [58] V. Pascucci. Slow growing subdivision (SGS) in any dimension: towards removing the curse of dimensionality. *Computer Graphics Forum*, 21(3), 2002.
- [59] V. Pascucci. Multi-resolution indexing for hierarchical out-of-core traversal of rectilinear grids. In G. Farin, H. Hagen, and B. Hamann, editors, *Hierarchical and Geometrical Methods for Scientific Visualization*, Springer Verlag, Heidelberg, Germany, 2003.
- [60] V. Pascucci and C. L. Bajaj. Time critical isosurface refinement and smoothing. In *Proceedings IEEE Symposium on Volume Visualization*, pages 33–42, Salt Lake City, UT, October 2000.
- [61] R. Perucchio, M. Saxena, and A. Kela. Automatic mesh generation from solid models based on recursive spatial decompositions. *International Journal for Numerical Methods in Engineering*, 28(11):2469–2501, November 1989.

- [62] M. Rivara and C. Levin. A 3D refinement algorithm for adaptive and multigrid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.
- [63] J. C. Roberts and S. Hill. Piecewise linear hypersurfaces using the marching cube algorithm. In R. Erbacher and A. Pang, editors, *Visual Data Exploration and Analysis VI, Proceedings of SPIE Visualization 2000*, pages 170–181. SPIE, 1999.
- [64] T. Roxborough and G. Nielson. Tetrahedron-based, least-squares, progressive volume models with application to freehand ultrasound data. In *Proceedings IEEE Visualization 2000*, pages 93–100, October 2000.
- [65] H. Samet. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, 18(1):37–57, January 1982.
- [66] H. Samet. Implementing ray tracing with octrees and neighbor finding. *Computers & Graphics*, 13(4):445–460, 1989. Also University of Maryland Computer Science TR-2204.
- [67] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.
- [68] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
- [69] M. Saxena and R. Perucchio. Element extraction for automatic meshing based on recursive spatial decompositions. Department of mechanical engineering, University of Rochester, Rochester, NY, June 1989.
- [70] G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Understanding*, 55(3):221–230, May 1992.
- [71] J. P. Snyder. *Map Projections - A Working Manual*. United States Government Printing Office, Washington, DC, 1987.
- [72] P. Sutton and C. D. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (TBON). In *Proceedings IEEE Visualization '99*, pages 147–154, 1999.
- [73] M. Tamminen, O. Karonen, and M. Mäntylä. Ray-casting and block model conversion using a spatial index. *Computer-Aided Design*, 16(4):203–208, July 1984.
- [74] W. Tobler and Z. T. Chen. A quadtree for global information storage. *Geographical Analysis*, 18(4):360–371, October 1986.
- [75] C. Weigle and D. Banks. Complex-valued contour meshing. In *Proceedings IEEE Visualization 1996*, pages 173–180, October 1996.

- [76] C. Weigle and D. Banks. Extracting iso-valued features in 4-dimensional scalar fields. In *Proceedings IEEE Visualization 1998*, pages 103–110, October 1998.
- [77] 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, October 1994.
- [78] M. A. Yerry and M. S. Shephard. A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, 3(1):39–46, January–February 1983.
- [79] Y. Zhou, B. Chen, and A. Kaufman. A multiresolution tetrahedral framework for visualizing regular volume data. In R. Yagel and H. Hagen, editors, *Proceedings IEEE Visualization '97*, pages 135–142, Phoenix, AZ, October 1997.