International Journal of Computational Geometry
& Applications
Vol. 17, No. 6 (2007) 595–631

World Scientific
www.worldscientific.com

POINTERLESS IMPLEMENTATION OF HIERARCHICAL SIMPLICIAL MESHES AND EFFICIENT NEIGHBOR FINDING IN ARBITRARY DIMENSIONS*

F. BETUL ATALAY

Mathematics and Computer Science Department
Saint Joseph's University, Philadelphia, PA 19131, USA
fatalay@sju.edu

DAVID M. MOUNT

Department of Computer Science and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742, USA
mount@cs.umd.edu

Received 10 February 2005
Revised 24 May 2006
Communicated by JSB Mitchell

ABSTRACT

We describe a pointerless representation of hierarchical regular simplicial meshes, based on a bisection approach proposed by Maubach. We introduce a new labeling scheme, called an LPT code, which uniquely encodes the geometry of each simplex of the hierarchy, and we present rules to compute the neighbors of a given simplex efficiently through the use of these codes. In addition, we show how to traverse the associated tree and how to answer point location and interpolation queries. Our system works in arbitrary dimensions.

Keywords: Pointerless data structures; neighbor finding; hierarchical simplicial meshes.

1. Introduction

Hierarchical simplicial meshes have been widely used in various application areas such as finite element computations, scientific visualization and geometric modeling. There has been considerable amount of work in simplicial mesh refinement, particularly in 2- and 3-dimensions, and a number of different refinement techniques have been proposed.[1,2,3,4,5,6,7,8] Because of the need to handle data sets with temporal components, there is a growing interest in higher dimensional meshes. In this paper, we build on a bisection refinement method proposed by Maubach.[7]

*This material is based upon work supported by the National Science Foundation under Grant No. 0098151. This work was done while the first author was a graduate student at the University of Maryland, College Park.

A hierarchical mesh is said to be *regular* if the vertices of the mesh are regularly distributed and the process by which a cell is subdivided is identical for all cells. Maubach developed a simple bisection algorithm based on a particular ordering of vertices and presented a mathematically rigorous analysis of the geometric structure of the hierarchical regular simplicial meshes in any dimension d.[7] Each element of such a mesh is a d-simplex, that is, the convex hull of $d+1$ affinely independent points.[9] The mesh is generated by a process of repeated bisection applied to a hypercube that has been initially subdivided into $d!$ congruent simplices. The subdivision pattern repeats itself on a smaller scale at every d levels. Whenever a simplex is bisected, some of its neighboring simplices may need to be bisected as well, in order to guarantee that the entire subdivision is *compatible*. Intuitively, a *compatible* subdivision is a subdivision in which pairs of neighboring cells meet along a single common face. A compatible simplicial subdivision is also referred to as a *simplicial complex*.[10] (See Fig. 1 for a 2-dimensional example.) Compatibility is important, since otherwise, cracks occur along faces of the subdivision, which in turn present problems when using the mesh for interpolation.

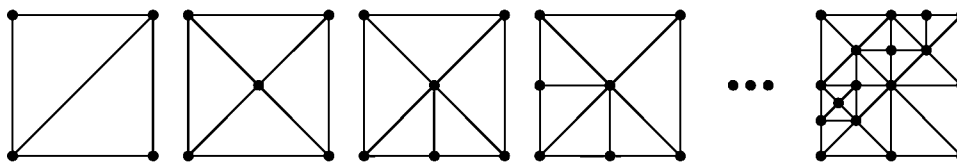

Fig. 1. Compatible simplicial mesh in the plane.

In computer graphics, adaptively refined regular meshes in 2- or 3-dimensions have been of interest for their use in realistic surface and volume rendering.[11,12,13] In many such applications, efficiency of various operations such as traversal and neighbor finding on the mesh is most desired. Based on the 3-dimensional version of Maubach's method, Hebert[14] presented a more efficient symbolic implementation of regular tetrahedral meshes by introducing an addressing scheme that allows unique labeling of the tetrahedra in the mesh, and he showed how to compute face neighbors of a tetrahedron based on its label. Hebert's addressing scheme could be generalized to higher dimensions, however the neighbor finding algorithms are quite specific to 3-dimensions, and a generalization to higher dimensions is a definite challenge. In this paper, we present such an algorithm that works in arbitrary dimensions.

Our interest in higher dimensions is motivated by another computer graphics application that accelerates ray-tracing[15] through multi-dimensional interpolation. In this application, rays in 3-space are modeled as points in a 4-dimensional parameter space, and each sample ray is traced through a scene to gather various geometric attributes that are required to compute an intensity value. (A ray is actually a 5-dimensional entity, but when rays are shot from a distant origin, it is more efficient to ignore the ray's origin and model it by its supporting line in space.)

Since tracing a ray through a complex scene can be computationally intensive, our approach is to instead collect and store a relatively sparse set of sample rays in a fast data structure and associate a number of continuous geometric attributes with each sample. We then interpolate among these samples to reconstruct the value at intermediate rays.[16,17] Because of variations in the field values, it is necessary to sample adaptively, with denser sampling in regions of high variation and sparser sampling in regions of low variation. An adaptively refined simplicial mesh is constructed over the 4-dimensional domain of interest, and the field values are sampled at the vertices of this subdivision. Given a query point, we determine which cell of the subdivision contains this point, and the interpolated value is an appropriate linear or multi-linear combination of the field values at the vertices of this cell.

For interpolation purposes, compatibly refined simplicial meshes are preferable over quadtree-like subdivisions, since they guarantee C^0 continuous interpolants. The problem with decompositions based on hyperrectangles (such as quadtrees and kd-trees) is the problem of "cracks." These arise when neighboring leaf cells are refined to different refinement levels, and new sample vertices are inserted along the boundary of one cell but not the other. Although it is possible to eliminate cracks by further subdividing a quadtree subdivision to produce a simplicial complex,[18,19,20] these approaches do not scale well to higher dimensions due to the exponential increase in the number of vertices in each cell and combinatorial explosion of different cases that need to be considered. Another advantage with simplicial decompositions is that interpolation is simpler and more efficient because linear interpolations can be used, which are based on a minimal number of samples ($d + 1$ samples for d-dimensional simplices) rather than multilinear interpolations (involving 2^d samples) for hyperrectangles. To illustrate these advantages more concretely, consider the images generated from our ray-tracing application in Fig. 2. Images (a) and (c) show the result of an interpolation based on kd-trees,[17] and images (b) and (d) show the results of using the hierarchical simplicial decomposition described in this paper.

In addition to our own motivation, higher dimensional meshes are of interest for visualization of time-varying fields, and efficient algorithms for performing traversals and neighbor finding is required.

Thus, our main objective in this paper is to present an efficient implementation of hierarchical regular simplicial meshes in any dimension d. Rather than representing the hierarchy explicitly as a tree using parent and child pointers, we use a *pointerless representation* in which nodes are accessed through an index called a *location code*. Location codes[21,22] have arisen as a popular alternative to standard pointer-based representations, because they separate the hierarchy from its representation, and so allow the application of very efficient access methods, such as hashing. The space savings realized by not having to store pointers can be quite significant for large multidimensional meshes. Each node would nominally be associated with $d + 4$ pointers. These are its pointers to its parent and each of its two children in the hierarchy, and pointers to each of its $d + 1$ face-neighboring simplices.

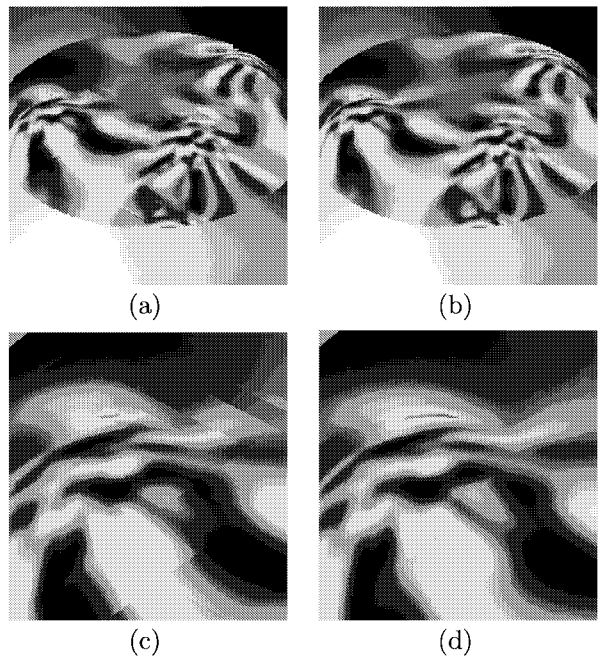

Fig. 2. Results of a ray-tracing application to produce an 800×800 image based on 4-dimensional interpolations using (a) a kd-tree based on 14,492 samples (96 CPU seconds) and (b) a simplex decomposition tree based on 6,072 samples (97 CPU seconds). Details of these images are shown in (c) and (d), respectively. Note the blocky artifacts in the kd-tree approach (c).

We store the mesh in a data structure called a *simplex decomposition tree*. We present a location code, called the *LPT code*, which can be used to access nodes of this tree. Our hierarchical decomposition is based on the same bisection method given by Maubach.[7] (Note that Maubach's representation is not pointerless.) In addition to efficient computation of neighbors, we show how to perform tree traversals, point locations, and answer interpolation queries efficiently through the use of these codes.

The remainder of the paper is organized as follows. In the next section we present prior work and discuss pointerless representations for hierarchical structures. In Section 3 we present basic notation and definitions. In Section 4 we introduce the LPT code and in Sections 5 and 6 we explain how to use the LPT code to perform tree traversals and compute neighbors.

2. Pointerless Representations and Prior Work

Regular subdivisions have the disadvantage of limiting the mesh's ability to adapt to the variational structure of the scalar field, but they provide a number of significant advantages from the perspectives of efficiency, practicality, and ease of use. The number of distinct element shapes is bounded (in our case by the dimension d),

and hence it is easy to derive bounds on the geometric properties of the cells, such as aspect ratios and angle bounds. The regular structure relieves us from having to store topological information explicitly, since this information is encoded implicitly in the tree structure. Additionally, the hierarchical structure provides a straightforward method for performing point location, which is important for answering interpolation queries.

One very practical advantage of regularity involves performance issues arising from modern memory hierarchies. It is well know that modern memory systems are based on multiple levels, ranging from registers and caches to main memory and disk (including virtual memory). The storage capacity at each level increases, and so too does access latency. There are often many orders of magnitude of difference between the time needed to access local data (which may be stored in registers or cache) versus global data (which may reside on disk).[23] Large dynamic pointer-based data structures are particularly problematic from this perspective, because node storage is typically allocated and deallocated dynamically and, unless special care is taken, simple pointer-based traversals suffer from a nonlocal pattern of memory references. This is one of the principal motivating factors behind I/O efficient algorithms[24,25] and cache conscious and cache oblivious data structures and algorithms.[23,26]

In contrast with pointer-based implementations, regular spatial subdivisions support *pointerless* implementations. Pointerless versions of quadtree and its variants have been known for many years.[27,21] The idea is to associate each node of the tree with a unique index, called a *location code*. Because of the regularity of the subdivision, given any point in space, it is possible to compute the location code of the node of a particular depth in the tree that contains this point. This can be done entirely in local memory, without accessing the data structure in global memory. Once the location code is known, the actual node containing the point can be accessed through a small number of accesses to global memory (e.g., by hashing).

Prior work on pointerless regular simplicial meshes has principally been in 2- and 3-dimensions. Lee and Samet presented a pointerless hierarchical triangulation based on a four-way decomposition of equilateral triangles.[22] Evans, Kirkpatrick and Townsend[28] considered triangulations based on bisection of right triangles. (This corresponds to the 2-dimensional case of the regular simplicial meshes we consider.) They developed a location code for this triangulation and provided an efficient neighbor finding method based on bit manipulation. Hebert presented a location code for bisection-based hierarchical tetrahedral meshes and a set of rules to compute neighbors efficiently in 3-space.[14] Lee, De Floriani and Samet developed an alternative location code for this same tetrahedral mesh, and presented algorithms for efficient neighbor computation.[29] Both approaches are based on an analysis of specific cases that arise in the 3-dimensional setting, and so do not readily generalize to higher dimensions.

Our interest is in hierarchical regular meshes, however, note that there is also significant interest in compact representation of irregular simplicial meshes.[30,31]

We introduce a new location code, which provides unique encoding of the sim-

plices generated by Maubach's[7] bisection algorithm. Unlike Hebert's approach, which only works in 3-dimensional space and relies on enumerations and look-up tables, our approach is fully algorithmic and works in arbitrary dimensions. We define the components required to develop a pointerless implementation based on our location code. The geometry of the simplices and the operations required for navigation in the associated tree can be computed easily based solely on the code of a simplex. Our location code and the definitions of various operations on the simplex tree depend on the particular vertex ordering. We have adopted a different ordering than Maubach's system, which we feel leads to simpler formulas. Our vertex ordering is a generalization of the vertex ordering used in Hebert's 3-dimensional system.

The most challenging operation on the tree is neighbor computation. Maubach's system computes the neighbors of a simplex during construction of the tree recursively,[32] and stores pointers to neighbors for each simplex. In contrast, we show how to compute an arbitrary neighbor of any simplex efficiently directly from its code, without storing any neighbor links, and without having to traverse the path to and from the root in order to compute neighbors. This is significant gain both in terms of storage, and computational efficiency, since our approach is local and runs in $O(d)$ time. In fact, it runs in $O(1)$ time, if the operations are encoded in lookup tables.

3. Preliminaries

Throughout, we consider real d-dimensional space, $\mathbb{R}^d$. We assume that the domain of interest has been scaled to lie within a unit *reference hypercube* of side length 2, centered at the origin, that is, $[-1,1]^d$. We shall denote points in $\mathbb{R}^d$ using lower-case bold letters, and represent them as d-element row vectors, that is, $\mathbf{v} = (v_1, v_2, \ldots, v_d) = (v_i)_{i=1}^d$. We let $\mathbf{e}_i$ denote the i-th unit vector. Note that we represent unit vectors as row vectors as well, for example, $\mathbf{e}_1 = (1,0,0)$ in 3-space. A d-simplex is represented as a $(d+1) \times d$ matrix whose rows are the vertices of the simplex, numbered from 0 to d. Of particular interest is the *base simplex*, denoted $S_\emptyset$, whose i-th vertex is $\sum_{j=1}^{i} \mathbf{e}_j - \sum_{j=i+1}^{d} \mathbf{e}_j$. For example, in $\mathbb{R}^3$ we have

$$S_\emptyset = \begin{bmatrix} -1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}. \tag{1}$$

Recall from basic geometry that two geometric objects are *congruent* if they are equivalent up to a rigid motion (translation, rotation and reflection). Coordinate permutations and coordinate reflections both preserve congruence. Two objects are *similar* if they can be made congruent by a nonzero uniform scaling.

3.1. *Permutations and reflections*

Our decomposition will involve applying repeated affine transformations to the base simplex. Two basic operations involve permutations and reflections of the unit vectors. Let $\mathrm{Sym}(d)$ denote the *symmetric group* of all $d!$ permutations over $\{1, 2, \ldots, d\}$. We denote a permutation $\Pi \in \mathrm{Sym}(d)$ by a tuple of distinct integers $[\pi_1\ \pi_2\ \cdots\ \pi_d]$, where $\pi_i \in \{1, 2, \ldots, d\}$. We can interpret such a permutation as a linear function that maps the unit vector $\mathbf{e}_i$ to the $\mathbf{e}_{\pi_i}$, or equivalently as a coordinate permutation given by a $d \times d$ matrix whose i-th row is the unit vector $\mathbf{e}_{\pi_i}$. That is, Π is associated with the matrix

$$\begin{bmatrix} \mathbf{e}_{\pi_1} \\ \mathbf{e}_{\pi_2} \\ \vdots \\ \mathbf{e}_{\pi_d} \end{bmatrix}.$$

For example, the permutation $\Pi = [2\ 3\ 1]$ is associated with

$$\begin{bmatrix} \mathbf{e}_2 \\ \mathbf{e}_3 \\ \mathbf{e}_1 \end{bmatrix} = \begin{bmatrix} 0\ 1\ 0 \\ 0\ 0\ 1 \\ 1\ 0\ 0 \end{bmatrix}$$

and transforms the base simplex into a congruent simplex by mapping columns as follows $1 \to 2$, $2 \to 3$, and $3 \to 1$. Note that the transformation matrix is post-multiplied with the simplex.

$$S_\emptyset\Pi = \begin{bmatrix} -1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0\ 1\ 0 \\ 0\ 0\ 1 \\ 1\ 0\ 0 \end{bmatrix} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

It is well known that the collection of simplices $\{S_\emptyset\Psi : \Psi \in \mathrm{Sym}(d)\}$ gives a simplicial decomposition of the reference hypercube, and further that this subdivision is compatible (is a simplicial complex).[33] These $d!$ simplices form the starting point of our hierarchical simplicial mesh. The matrix associated with the *composition* of two permutations $\Pi \circ \Psi$, defined as $S(\Pi \circ \Psi) = (S\Psi)\Pi$ is given by the matrix product $\Psi\Pi$.

Note that the notation $[2\ 3\ 1]$ is not a vector in $\mathbb{R}^d$, but merely a convenient shorthand for a permutation matrix. Throughout, vectors will be denoted with parentheses, and square brackets will be used for objects that are to be interpreted as linear transformations, or equivalently a shorthand for a matrix.

Another useful class of transformations are coordinate reflections, which can be expressed as a d-tuple $R = [r_1\ r_2\ \cdots\ r_d]$ where $r_i \in \{\pm 1\}$, and is interpreted as a linear transformation represented by the diagonal matrix $diag(r_1, r_2, \ldots, r_d)$. It will simplify notation to combine the composition of a permutation and a reflection using a unified notation. We define a *signed permutation* to be a d-tuple of integers $[r_i\pi_i]_{i=1}^d$, where $[\pi_i]_{i=1}^d$ is a permutation and $[r_i]_{i=1}^d$ is a reflection. This is interpreted

as a linear transformation that maps the i-th unit vector to $r_i \mathbf{e}_{\pi_i}$. For example, in $\mathbb{R}^3$, the composition of the reflection $R = [-1\ -1\ +1]$ and the permutation $\Pi = [2\ 3\ 1]$ is expressed as the signed permutation $[-2\ -3\ +1]$, which is just a shorthand for the matrix product $R\Pi$, that is

$$R\Pi = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & +1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ +1 & 0 & 0 \end{bmatrix}.$$

An intuitive way to interpret the meaning of a signed permutation is as an operation involving a selective negation followed by a subsequent permutation of some of the components of a row vector or the columns of a matrix. For example the signed permutation $[-2\ -3\ +1]$ can be interpreted as negating the first and second components of a vector, and then mapping the first, second, and third components of the resulting vector to positions 2, 3, and 1, respectively. Thus, the image of (v_1, v_2, v_3) under this transformation is $(v_3, -v_1, -v_2)$.

We define the following functions that act on a signed permutation $\Pi = [\pi_i]_{i=1}^d$. The first, $\mathit{perm}(\Pi)$, extracts the permutation part of Π, the second, $\mathit{refl}(\Pi)$, extracts the (unpermuted) reflection part as a vector in $\{\pm 1\}^d$, and the third, $\mathit{orth}(\Pi)$, returns the permutation of $\mathit{refl}(\Pi)$ under Π. More formally,

$$\begin{aligned} \mathit{perm}(\Pi) &= [\ |\pi_i|\ ]_{i=1}^d \\ \mathit{refl}(\Pi) &= (\mathit{sign}(\pi_i))_{i=1}^d \\ \mathit{orth}(\Pi) &= \mathit{refl}(\Pi)\mathit{perm}(\Pi) = (\mathit{sign}(\pi_i^{-1}))_{i=1}^d, \end{aligned} \tag{2}$$

where π_i^{-1} denotes the i-th component of the inverse of Π. Note that, since Π is an orthogonal matrix, $\Pi^{-1} = \Pi^T = [r_1 \mathbf{e}_{|\pi_1|}^T\ r_2 \mathbf{e}_{|\pi_2|}^T \cdots r_d \mathbf{e}_{|\pi_d|}^T]$, where $(r_1, r_2, \ldots, r_d)$ denotes the reflection part of Π.

For example, if $\Pi = [-2\ -3\ +1]$ then $\mathit{perm}(\Pi) = [2\ 3\ 1]$, $\mathit{refl}(\Pi) = (-1, -1, +1)$, and $\mathit{orth}(\Pi) = (+1, -1, -1)$. Note that $\mathit{refl}(\Pi)$ and $\mathit{orth}(\Pi)$ are vectors. The associated transformation matrices are $\mathit{diag}(\mathit{refl}(\Pi))$ and $\mathit{diag}(\mathit{orth}(\Pi))$, respectively. The following technical lemma is an easy consequence of these definitions, and will be useful in some of our later proofs.

Lemma 1. *Let Π be a signed permutation. Then*

$$\Pi = \mathit{diag}(\mathit{refl}(\Pi))\mathit{perm}(\Pi) = \mathit{perm}(\Pi)\mathit{diag}(\mathit{orth}(\Pi)). \tag{3}$$

3.2. *The simplex decomposition tree*

Recall that the initial simplicial complex is formed from the $d!$ permutations of the base simplex, that is, $S_\emptyset \Psi$, for $\Psi \in \mathrm{Sym}(d)$. Simplices are then refined by a process of repeated subdivision, called *bisection*, in which a simplex is bisected by splitting one of its edges determined by the specific vertex ordering.[7] (Details will be given below.) The resulting *child* simplices are labeled 0 and 1. By applying

the process repeatedly, each simplex in this hierarchy is uniquely identified by its *path*, which is a string over $\{0,1\}$. The resulting collection of trees is called the *simplex decomposition tree*, or *SD-tree* for short. It consists of $d!$ separate binary trees, which conceptually are joined under a common super-root. Each simplex of this tree is uniquely identified by a *permutation-path pair* as $S_{\Psi,p}$, where Ψ is the initial permutation of the base simplex, and $p \in \{0,1\}^*$ is the path string. When starting with the base simplex (Ψ is the identity permutation) we may omit explicit reference to Ψ. By symmetry, it suffices to describe the bisection process on just the base simplex $S_\emptyset$. The ordering of the rows, that is, the numbering of vertices, will be significant.

Maubach[7] showed that with every d consecutive bisections, the resulting simplices are similar copies of their d-fold grandparent, subject to a uniform scaling by $1/2$. Thus, the pattern of decomposition repeats every d levels in the decomposition. Define the *level*, ℓ, of a simplex S_p to be the path length modulo the dimension, that is, $\ell = (|p| \bmod d)$, where $|p|$ denotes the length of p. The 0-child S_{p0} and 1-child S_{p1} of a simplex are computed as follows:

$$S_p = \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_{\ell-1} \\ \mathbf{v}_\ell \\ \mathbf{v}_{\ell+1} \\ \vdots \\ \mathbf{v}_d \end{bmatrix} \qquad S_{p0} = \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_{\ell-1} \\ (\mathbf{v}_\ell + \mathbf{v}_d)/2 \\ \mathbf{v}_{\ell+1} \\ \vdots \\ \mathbf{v}_d \end{bmatrix} \qquad S_{p1} = \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_{\ell-1} \\ (\mathbf{v}_\ell + \mathbf{v}_d)/2 \\ \mathbf{v}_\ell \\ \vdots \\ \mathbf{v}_{d-1} \end{bmatrix}. \quad (4)$$

A portion of the tree is illustrated in Fig. 3. Note that in both cases the first ℓ vertices are unchanged. The new ℓ-th vertex is the midpoint of the edge between the ℓ-th and last vertices. The remaining $d - \ell$ vertices are a subsequence of the original vertices, shifted by one position relative to each other.

Equivalently, we can define $S_{p0} = B_{\ell,0} S_p$ and $S_{p1} = B_{\ell,1} S_p$, where $B_{\ell,0}$ and $B_{\ell,1}$ are $(d+1) \times (d+1)$ matrices whose ℓth row (starting from row 0) has the value $1/2$ in columns ℓ and d (starting from column 0), and all other rows are unit vectors. For example, in dimension $d = 4$ and for $\ell = 2$ we have

$$B_{\ell,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad B_{\ell,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \quad (5)$$

Our bisection scheme is geometrically equivalent to the one defined by Maubach,[7] but we order the vertices differently from Maubach and reverse the names of the 0- and 1-children. Although the differences are theoretically insignificant, our ordering results in somewhat simpler and more regular formulas for computing descendents and neighbors.

3.3. *Reference simplices and the reference tree*

Since with every d consecutive bisections, the simplices are similar to, but half the size, of their d-fold grandparent, we can partition the decomposition tree into a collection of isomorphic, disjoint subtrees of height d. The roots of these subtrees are the nodes whose depths are multiples of d (where the root starts at depth 0). It suffices to analyze the structure of just one of these trees, in particular, the subtree of height d starting at the root. We call this the *reference tree*. Since the two children of any simplex are congruent, it follows that all the simplices at any given depth of the decomposition tree are congruent to each other. Thus, all the similarity classes are represented by d canonical simplices, called the *reference simplices*. These are defined to be $S_{(0^k)}$, for $0 \leq k < d$, and denoted by Δ_k. (See Fig. 3.) Although it is not a reference simplex, we also define $\Delta_d = S_{(0^d)}$, since it is useful in our proofs.

For example, in $\mathbb{R}^3$ the 3 reference simplices together with Δ_3 are

$$\underset{(S_\emptyset)}{\Delta_0} = \begin{bmatrix} -1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \underset{(S_0)}{\Delta_1} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \underset{(S_{00})}{\Delta_2} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \underset{(S_{000})}{\Delta_3} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}. \tag{6}$$

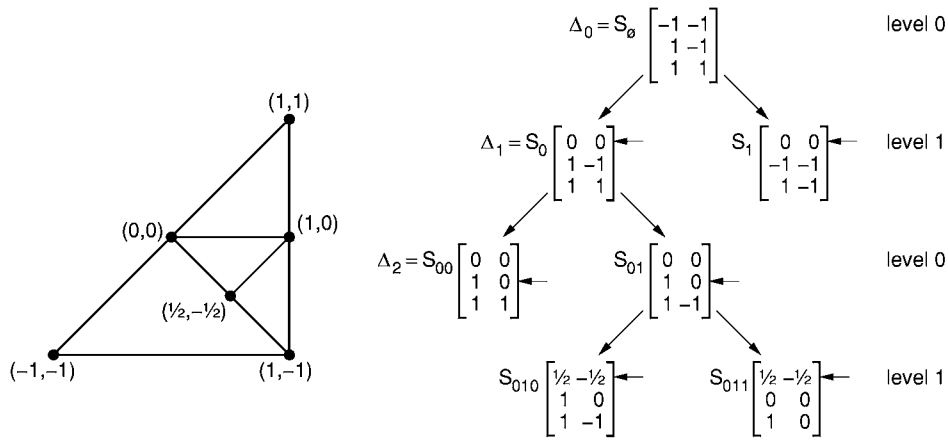

Fig. 3. The simplex decomposition tree. The corresponding bisected simplex is shown on the left. The newly created vertex is indicated by an arrow in each case. The reference simplices Δ_i are indicated as well.

4. The LPT code

So far we have defined a simplicial decomposition process and a tree structure that is naturally associated with this process. In order to provide pointerless implementation of the hierarchical mesh, we define a *location code*, which uniquely encodes

each simplex of the hierarchy. The most direct location code is the combination consisting of the initial permutation Ψ followed by the binary encoding of the tree path p. Unfortunately, it is not easy to compute basic properties of the simplex such as neighbors from this code. Nonetheless, Lee, De Floriani, and Samet showed how to compute neighbors from the path code in the 3-dimensional case.[29] Instead we modify an approach presented by Hebert[14] for the 3-dimensional case, by defining a location code that more directly encodes the geometric relationship between the each simplex and the reference simplex at the same level. We call this the *LPT code*, since, for each simplex, it encodes its *Level*, its signed *Permutation*, and its *Translation* relative to some reference simplex. We shall show that it is possible to compute tree relations (children and parents) as well as neighbors in the simplicial complex using this code.

Given any simplex $S_{\Psi,p}$ in the hierarchy, the *LPT code* is a 3-tuple (ℓ, Π, Φ), which consists of the following three components:

Level: $\ell = |p| \bmod d$ is the simplex's level.
Permutation: Π is a signed permutation relating $S_{\Psi,p}$ to the corresponding reference simplex at this level.
Translation: Φ is a list of vectors, called the orthant list, which is used to derive the translation relative to the reference simplex.

The permutation part $\Pi = \Pi_{\Psi,p}$ and orthant list $\Phi = \Phi_{\Psi,p}$ are defined below as functions of Ψ and p. Correctness will be established in Theorem 1 below.

Permutation Part: Maubach proved that the vectors defined by the simplices of any level of the tree are essentially related to one another by a coordinate permutation and reflection. The purpose of the signed permutation part of the code is to encode these two elements. The signed permutation $\Pi_{\Psi,p}$ is defined recursively as follows for a base permutation Ψ and binary tree path p:

$$\Pi_{\Psi,\emptyset} = \Psi \qquad \Pi_{\Psi,p0} = \Pi_{\Psi,p} \qquad \Pi_{\Psi,p1} = \Pi_{\Psi,p} \circ \Sigma_\ell = \Sigma_\ell \Pi_{\Psi,p}, \quad (7)$$

where Σ_ℓ is the permutation that cyclically shifts the last $d-\ell$ elements to the right and negates the element that is wrapped around. That is, $\Sigma_\ell = [1\ 2\ \cdots\ \ell\ (-d)\ (\ell+1)\ (\ell{+}2)\ \cdots\ (d{-}1)]$. A portion of the simplex decomposition tree, and the associated permutation values are shown in Fig. 4. For example, observe that S_1 is related to Δ_1 by the signed permutation $[-2\ +1]$, which negates the first column of Δ_1 and then swaps the two columns.

Orthant List: Recall that with every d levels of descent in the decomposition tree, the resulting simplices decrease in size by a factor of $1/2$. The bounding hypercube of the resulting descendent is one of the 2^d hypercubes that would result from a quadtree-like decomposition (indicated by broken lines on the left side of Fig. 4). Depending on the level within the tree, the translation of the descendent hypercube relative to its ancestor will be some power of $(1/2)$ times a d-vector over $\{\pm 1\}$. Such

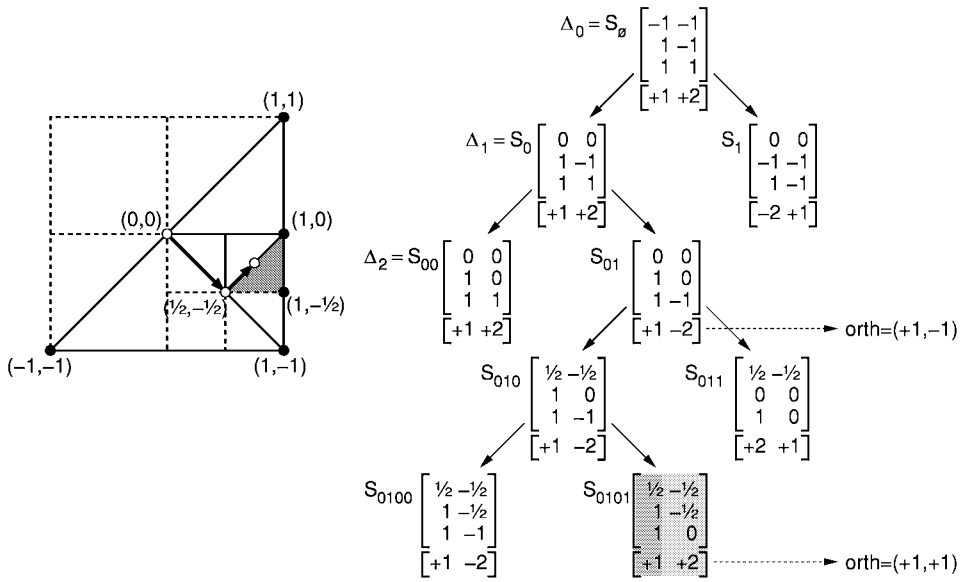

Fig. 4. The signed permutations $\Pi_{\Psi,p}$ associated with each simplex are shown below each simplex matrix, and the entries of the orthant list are shown for the shaded simplex S_{0101}. The LPT code for this simplex is $(0, [+1\ +2], \langle(+1,-1),(+1,+1)\rangle)$.

a vector defines the *orthant* containing the descendent hypercube relative to the central vertex of its ancestor. Consider, for example, the shaded simplex in Fig. 4. Its translation relative to the base simplex is $\frac{1}{2}(+1,-1)+\frac{1}{4}(+1,+1)$, indicated by the arrowed lines on the left side of the figure. The orthant list encodes these two vectors.

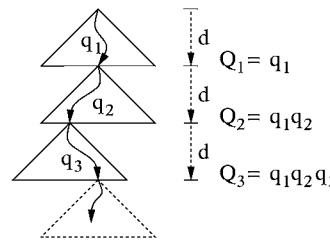

Fig. 5. Orthant List

To define the orthant list, we first remove the last ℓ symbols of p, leaving a multiple of d symbols (possibly empty). We then partition the remaining symbols into $L = \lfloor |p|/d \rfloor$ substrings, $q_1 q_2 \ldots q_L$, where $|q_i| = d$. (See Fig. 5.) Since the reference tree structure repeats every d levels, each q_i can be viewed as a complete path in one of these subtrees of height d. Let Q_i denote the concatenation of the

first i substrings. For $1 \leq i \leq L$, define $\Gamma_{\Psi,p}[i]$ to be the signed permutation for path Q_i, that is, Π_{Ψ,Q_i}. Define the *orthant list* for the pair (Ψ, p) to be the sequence of L vectors whose i-th element is $orth(\Gamma_{\Psi,p}[i])$, that is

$$\Phi_{\Psi,p} = \langle orth(\Gamma_{\Psi,p}[1]), orth(\Gamma_{\Psi,p}[2]), \ldots, orth(\Gamma_{\Psi,p}[L]) \rangle . \tag{8}$$

The orthant list can be computed incrementally along with the permutation part of the code as follows. Given the LPT code (ℓ, Π, Φ) for a simplex $S_{\Psi,p}$, first observe that the orthant list only changes for the children if the current level is $d-1$. If so, we compute the child's permutation Π' from Eq. (7) and append $orth(\Pi')$ to the current list. For example, consider the incremental computation of the orthant list for simplex S_{0101} in Fig. 4. We start with the root simplex. The orthant list of $S_\emptyset$ is empty. For S_0, the orthant list remains the same as its parent, thus, is empty. To compute the orthant list of S_{01}, we need to append $orth([+1\ -2]) = (+1, -1)$ to its parent's orthant list since the parent is at level 1. Therefore, the orthant list of S_{01} is $\langle(+1, -1)\rangle$. For S_{010}, the orthant list remains the same as its parent, thus, is $\langle(+1, -1)\rangle$. For S_{0101}, we need to append $orth([+1\ +2]) = (+1, +1)$ to its parent's orthant list since the parent is at level 1. Therefore, the orthant list of S_{0101} is $\langle(+1, -1), (+1, +1)\rangle$.

The computation of the LPT code is summarized in the procedure *LPTcode* shown in Fig. 6. The code for the simplex $S_{\Psi,p}$ is computed by the call $LPTcode(\Psi, p)$.

Input: A simplex defined by Ψ and p.
Output: The LPT code, (ℓ, Π, Φ), of the input simplex.
LPTcode(Ψ, p)
    // start with the LPT code of the root simplex,
    $\Pi \leftarrow \Psi, \quad \ell \leftarrow 0, \quad \Phi \leftarrow \emptyset$
    // incrementally update the LPT code along path p.
    Express p as $p_0 p_1 \cdots p_k$
    for $(0 \leq i \leq k)$
        $\ell \leftarrow (\ell + 1) \bmod d$
        if $(p_i = 1)$ $\Pi \leftarrow \Pi \circ \Sigma_\ell$
        if $(\ell = 0)$ $\Phi \leftarrow \Phi + orth(\Pi)$
    return (ℓ, Π, Φ)

Fig. 6. Procedure *LPTcode*, which computes the LPT code for the simplex $S_{\Psi,p}$.

We may now state the main result of this section, called the *LPT Theorem*, which establishes the geometric meaning of our LPT code by relating each simplex of the decomposition tree to its associated reference simplex. Hebert[14] proved the analogous result for his 3-dimensional bisection system. But first, we will prove the following technical lemmas. Recall from the definition of reference simplices that $\Delta_{\ell+1} = B_{\ell,0}\Delta_\ell$.

Lemma 2. *Given the reference simplex* Δ_ℓ, $0 \leq \ell < d$,

$$B_{\ell,0}\Delta_\ell = B_{\ell,1}\Delta_\ell\Sigma_\ell^{-1}, \tag{9}$$

where Σ_ℓ *is as defined in Section 4.*

Proof. $\Sigma_\ell^{-1} = \Sigma_\ell^T = [\mathbf{e}_1^T \ldots \mathbf{e}_\ell^T \quad -\mathbf{e}_d^T \quad \mathbf{e}_{\ell+1}^T \ldots \mathbf{e}_{d-1}^T]$, since Σ_ℓ is an orthogonal matrix. When a matrix is postmultiplied by Σ_ℓ^{-1}, the last column is negated, and then the last $(d-\ell)$ columns are cyclically shifted to the right. Consider the general form of a reference simplex and its two children as shown in Fig. 7. It can be observed that, if we negate the last column of the 1-child of Δ_ℓ, and cyclically shift the last $(d-\ell)$ columns to the right, we get the 0-child of Δ_ℓ. See Fig. 8 for an example where $d = 4$ and $l = 1$. □

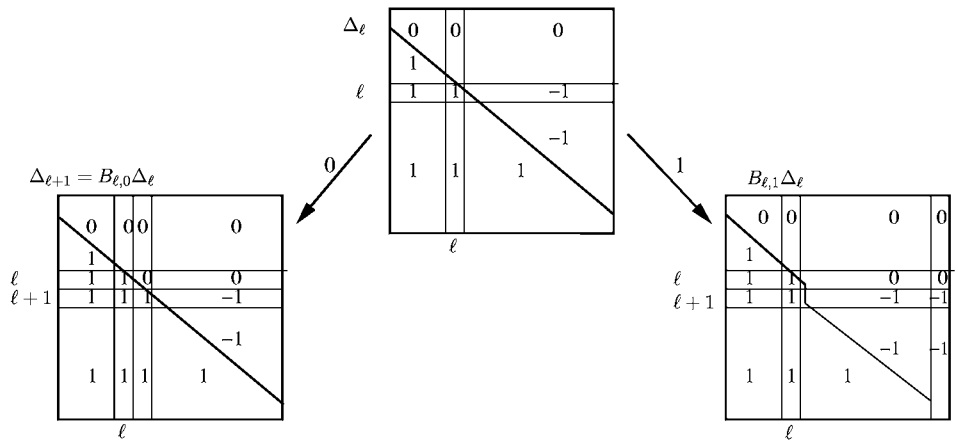

Fig. 7. The two children of a reference simplex.

Lemma 3. *Given the reference simplex* Δ_ℓ, $0 \leq \ell < d$, *and a signed permutation* $\Pi_{\Psi,p}$,

$$B_{\ell,1}\Delta_\ell\Pi_{\Psi,p} = \Delta_{\ell+1}\Pi_{\Psi,p1}. \tag{10}$$

Proof. By definition, $\Pi_{\Psi,p1} = \Sigma_\ell\Pi_{\Psi,p}$, that is, $\Pi_{\Psi,p} = \Sigma_\ell^{-1}\Pi_{\Psi,p1}$. Thus, $B_{\ell,1}\Delta_\ell\Pi_{\Psi,p} = \Delta_{\ell+1}\Sigma_\ell\Pi_{\Psi,p} = \Delta_{\ell+1}\Sigma_\ell\Sigma_\ell^{-1}\Pi_{\Psi,p1} = \Delta_{\ell+1}\Pi_{\Psi,p1}$. □

Let $\mathbf{1}_{d+1}^T$ denote a $(d+1)$-column vector of 1's. The following theorem makes use of the observation that, for any d-row vector $\mathbf{v}$, the matrix product $\mathbf{1}_{d+1}^T \cdot \mathbf{v}$ is a $(d+1) \times d$ vector whose rows are all equal to $\mathbf{v}$, and hence adding this to any simplex matrix is equivalent to a translation by $\mathbf{v}$.

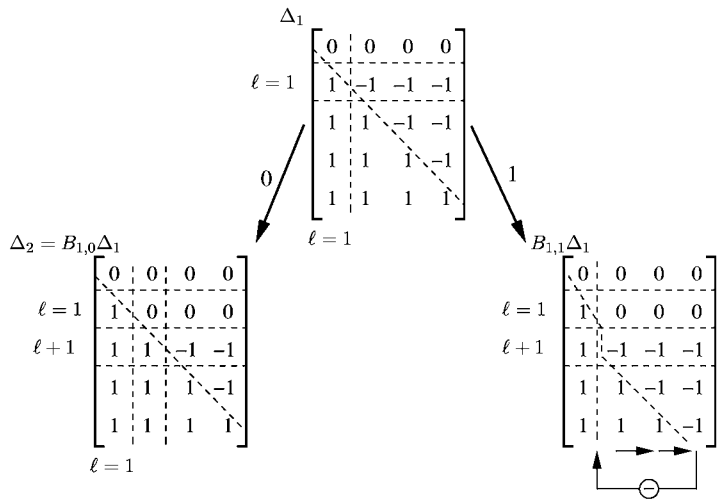

Fig. 8. The two children of reference simplex Δ_1 for $d = 4$. Notice that by negating the last column of the 1-child of Δ_1, and cyclically shifting the last 3 columns to the right, we get the 0-child of Δ_1.

Theorem 1. (LPT Theorem) *Let $S_{\Psi,p}$ be the simplex of the decomposition tree associated with some initial permutation Ψ and binary path p. Let (ℓ, Π, Φ) be the LPT code for this simplex, defined above. Then $S_{\Psi,p}$ is related to Δ_ℓ, the reference simplex at this level, by the following similarity transformation:*

$$S_{\Psi,p} = \frac{1}{2^L}\Delta_\ell \Pi + \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i}\Phi[i] \tag{11}$$

where $L = \lfloor\, |p|/d \rfloor$.

Proof. We will prove Theorem 1 by induction on the depth of the simplex in the decomposition tree. Recall that $\Pi = \Pi_{\Psi,p}$, $\Phi = \Phi_{\Psi,p}$, $l = |p| \bmod d$ and $L = \lfloor\, |p|/d \rfloor$.

For the basis case, consider any of the root simplices, $S_{\Psi,\emptyset}$. Since $L = \lfloor |p|/d \rfloor = 0$, and $\ell = 0$ at root level, we have

$$S_{\Psi,\emptyset} \;=\; \frac{1}{2^0}\Delta_0 \Pi_{\Psi,\emptyset} + \mathbf{1}_{d+1}^T \sum_{i=1}^{0} \frac{1}{2^i}\Phi_{\Psi,\emptyset}[i] \;=\; \Delta_0 \Pi_{\Psi,\emptyset}. \tag{12}$$

Next, assume that the induction hypothesis holds for $S_{\Psi,p}$, at level $\ell = |p| \bmod d$. We will show that, it holds for the 0- and 1-children of $S_{\Psi,p}$. We distinguish between two cases, based on whether the level ℓ is equal to $d - 1$.

Case 1: $(0 \le \ell \le d - 2)$ For levels in this range we have $\lfloor |p0|/d \rfloor = \lfloor |p1|/d \rfloor = L$, which implies that there is no change in the orthant list, and hence no change in the translational component of the result. To simplify

the notation, let T denote this translational component, that is, $T = \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i} \Phi_{\Psi,p}[i]$. Note that,

$$T = \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i} \Phi_{\Psi,p0}[i] = \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i} \Phi_{\Psi,p1}[i]. \tag{13}$$

Note that all the rows of T are equal to each other. Premultiplying by either $B_{\ell,0}$ or $B_{\ell,1}$ has the effect of replacing each row of a matrix with either another row or the average of two rows. Thus it follows that $B_{\ell,0}T = B_{\ell,1}T = T$. With this in mind, let's consider the two children of $S_{\Psi,p0}$.

0-Child: $(S_{\Psi,p0})$ By definition, $\Pi_{\Psi,p0} = \Pi_{\Psi,p}$, from which we have

$$\begin{aligned} S_{\Psi,p0} &= B_{\ell,0} S_{\Psi,p} = B_{\ell,0} \left(\tfrac{1}{2^L} \Delta_\ell \Pi_{\Psi,p} + T \right) && \text{(by induc. hyp.)} \\ &= \tfrac{1}{2^L} \Delta_{\ell+1} \Pi_{\Psi,p} + T = \tfrac{1}{2^L} \Delta_{\ell+1} \Pi_{\Psi,p0} + T. \end{aligned} \tag{14}$$

1-Child: $(S_{\Psi,p1})$ By definition we have

$$\begin{aligned} S_{\Psi,p1} &= B_{\ell,1} S_{\Psi,p} = B_{\ell,1} \left(\tfrac{1}{2^L} \Delta_\ell \Pi_{\Psi,p} + T \right) && \text{(by induc. hyp.)} \\ &= \tfrac{1}{2^L} \Delta_{\ell+1} \Pi_{\Psi,p1} + T. && \text{(by Lemma 3)} \end{aligned} \tag{15}$$

Together, these complete the induction for Case 1.

Case 2: $(\ell = d - 1)$ This case is more complicated because the children of $S_{\Psi,p}$ will be at level 0, implying that we need to consider the effect of the new orthant list entry and the resulting translation. Again, we consider the two children separately.

0-Child: $(S_{\Psi,p0})$ By definition, $\Pi_{\Psi,p0} = \Pi_{\Psi,p}$, from which we have

$$\begin{aligned} S_{\Psi,p0} &= B_{d-1,0} S_{\Psi,p} \\ &= B_{d-1,0} \left(\tfrac{1}{2^L} \Delta_{d-1} \Pi_{\Psi,p} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i} \Phi_{\Psi,p}[i] \right) \\ &= \tfrac{1}{2^L} \Delta_d \Pi_{\Psi,p} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i} \Phi_{\Psi,p}[i] \\ &= \tfrac{1}{2^L} \Delta_d \Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i} \Phi_{\Psi,p0}[i] \\ &= \tfrac{1}{2^L} \Delta_d \Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \frac{1}{2^i} \Phi_{\Psi,p0}[i] - \frac{1}{2^{L+1}} \mathbf{1}_{d+1}^T \Phi_{\Psi,p0}[L+1] \end{aligned} \tag{16}$$

Since $\Delta_d = \frac{\Delta_0 + [\mathbf{1}]_{(d+1)\times d}}{2}$, where $[\mathbf{1}]_{(d+1)\times d}$ is a matrix of 1's, and

since $\Phi_{\Psi,p0}[L+1] = orth(\Pi_{\Psi,p0})$, we have

$$\begin{aligned} S_{\Psi,p0} &= \tfrac{1}{2^L}\tfrac{(\Delta_0+[\mathbf{1}]_{(d+1)\times d})}{2}\Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \tfrac{1}{2^i}\Phi_{\Psi,p0}[i] \\ &\quad - \tfrac{1}{2^{L+1}}\mathbf{1}_{d+1}^T orth(\Pi_{\Psi,p0}) \\ &= \tfrac{1}{2^{L+1}}\Delta_0\Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \tfrac{1}{2^i}\Phi_{\Psi,p0}[i] \\ &\quad + \tfrac{1}{2^{L+1}}([\mathbf{1}]_{(d+1)\times d}\Pi_{\Psi,p0} - \mathbf{1}_{d+1}^T orth(\Pi_{\Psi,p0})) \end{aligned} \tag{17}$$

Now, by applying Lemma 1 we have

$$\begin{aligned} \mathbf{1}_{d+1}^T orth(\Pi_{\Psi,p0}) &= \mathbf{1}_{d+1}^T refl(\Pi_{\Psi,p0}) perm(\Pi_{\Psi,p0}) \\ &= [\mathbf{1}]_{(d+1)\times d} diag(refl(\Pi_{\Psi,p0})) perm(\Pi_{\Psi,p0}) \\ &= [\mathbf{1}]_{(d+1)\times d}\Pi_{\Psi,p0}. \end{aligned} \tag{18}$$

From this it follows that the last term in the above expression of $S_{\Psi,p0}$ is 0, and hence

$$S_{\Psi,p0} = \frac{1}{2^{L+1}}\Delta_0\Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p0}[i]. \tag{19}$$

1-Child: ($S_{\Psi,p1}$) By definition we have

$$\begin{aligned} S_{\Psi,p1} &= B_{d-1,1}S_{\Psi,p} \\ &= B_{d-1,1}\left(\tfrac{1}{2^L}\Delta_{d-1}\Pi_{\Psi,p} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \tfrac{1}{2^i}\Phi_{\Psi,p}[i]\right) \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p1} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \tfrac{1}{2^i}\Phi_{\Psi,p1}[i] \qquad \text{(by Lemma 3)} \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p1} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \tfrac{1}{2^i}\Phi_{\Psi,p1}[i] - \tfrac{1}{2^{L+1}}\mathbf{1}_{d+1}^T\Phi_{\Psi,p1}[L+1] \end{aligned} \tag{20}$$

Applying the same derivations as in the previous case, this can be reduced to

$$S_{\Psi,p1} = \frac{1}{2^{L+1}}\Delta_0\Pi_{\Psi,p1} + \mathbf{1}_{d+1}^T \sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p1}[i]. \tag{21}$$

Because $\ell = d-1$, in both cases the child's level is $L+1$, and so Eqs. (19) and (21) complete the induction for the Case 2. □

Pointerless Implementation: We can now describe a pointerless implementation of a simplex decomposition tree. For each simplex $S_{\Psi,p}$ in the tree, we create a node that is indexed by an appropriate encoding of the associated LPT code. Theorem 1 implies that the geometry of this simplex is determined entirely from the LPT code, and, if desired, it can be computed from the code in time proportional to the code length. Note that, this node may also contain application-specific data. These

objects are then stored in any index structure that supports rapid look-ups, for example, a hash table.

There are a number of practical observations that can be made in how to encode LPT codes efficiently in low dimensional spaces. Let D denote the maximum depth of any node in the tree. Each of the $d!$ permutations of $\mathrm{Sym}(d)$ can be encoded as an integer with $\log_2(d!)$ bits.[34] A d-element reflection vector over $\{\pm 1\}$ can be represented as a d-element bit string (e.g., by the mapping $+1 \rightarrow 0$ and $-1 \rightarrow 1$). Thus, a signed permutation Π then can be encoded by a pair of integers. A convenient way to encode the vectors of the orthant list is to map them to bit strings and to store them as d separate lists, one for each coordinate. (The advantage of this representation will be discussed in Section 6.) The final code consists of the level ℓ, expressed with $\lceil \log_2 d \rceil$ bits, the permutation and reflection, represented using $\lceil \log_2(d!) \rceil + d$ bits, and finally the orthant list, represented using $d \cdot \mathit{length}(\Phi)$ bits, which is at most $d \lfloor D/d \rfloor \leq D$.

Given a simplex at level D, the total number of bits needed to represent the code for this simplex is $D + \log_2(d!) + O(d)$. This is asymptotically optimal in the worst case, since there are $2^D d!$ simplices at depth D in a full tree, which would require at least $D + \log_2(d!) = D + \Omega(d \log d)$ bits. Under the reasonable assumption that machine's word size is $\Omega((D/d) + \log_2(d!))$, the permutation part of the code can be stored in a constant number of machine words and the orthant lists can be stored in $O(d)$ machine words.

Also, note that for small d, the multiplication tables for the various signed permutations (such as Σ_ℓ of Eq. (7) and the neighbor permutations of Section 6 below) can be precomputed and stored in tables. This allows very fast evaluation of permutation operations by simple table look-up.

5. Decomposition Tree Operations

In this section we present methods for performing useful tree access operations based on manipulations of LPT codes, including tree traversal, point location and interpolation queries.

5.1. *Tree traversal*

Consider a simplex $S_{\Psi,p}$ of the tree whose LPT code is (ℓ, Π, Φ). Let us consider how to compute the children and parent of this simplex in the tree. The LPT codes of the children of this simplex can be computed in $O(d)$ time by applying the recursive rules used to define the LPT code, given in Section 4. We can compute the parent from the LPT code by inverting this process, but in order to do so we need to know whether the simplex is a left child, a right child, or the root. A root simplex is distinguished by having an empty orthant list and level $\ell = 0$. Otherwise, we make use of the following lemma. Given a simplex at level $\ell \neq 0$, the *nearest proper level-0 ancestor* is defined to be its ℓ-th ancestor. Given a nonroot simplex at level 0, it is defined to be its d-th ancestor.

Lemma 4. *Consider a nonroot simplex S of the decomposition tree with LPT code (ℓ, Π, Φ), and let S' be its nearest proper ancestor at level 0. Let $\Pi = [\pi_i]_{i=1}^d$ be the signed permutation of S, let $\mathbf{o} = (o_i)_{i=1}^d$ be the last entry of the orthant list of S', and let $\ell^* = 1+((\ell-1) \bmod d)$. Then S is a 0-child if and only if $sign(\pi_{\ell^*}) = sign(o_{|\pi_{\ell^*}|})$.*

This result is an easy corollary of the following more general lemma, which characterizes the child relations for a simplex's ancestors, up to the next 0th level. Lemma 4 follows by observing that $S_{\Psi,p}$ is a 0-child if and only if the last bit of the tree path (b_{ℓ^*} in the statement of the following lemma) is zero.

Lemma 5. *Let $S_{\Psi,p}$ be a nonroot simplex, and let $S_{\Psi,t}$ be its nearest proper ancestor of level 0. Let $\mathbf{o} = orth(\Pi_{\Psi,t})$ be the last orthant list entry of $S_{\Psi,t}$. Let $b_1 b_2 \ldots b_{\ell^*}$ denote the path from $S_{\Psi,t}$ to $S_{\Psi,p}$, where $\ell^* = 1+((\ell-1) \bmod d)$. Let $\Pi_{\Psi,p} = [\pi_i]_{i=1}^d$ and $\mathbf{o} = (o_i)_{i=1}^d$. Then*

$$b_i = \begin{cases} 0 \text{ if } & sign(\pi_i) = sign(o_{|\pi_i|}) \\ 1 \text{ if } & \text{otherwise.} \end{cases} \tag{22}$$

Proof. We do not know what $\Pi_{\Psi,t}$ is, but since we know $\mathbf{o}$, we know the signs of each coordinate axis in $\Pi_{\Psi,t}$. We can determine $b_1 b_2 \ldots b_{\ell^*}$ by finding out which axes changed signs as we go down the tree from $S_{\Psi,t}$ to $S_{\Psi,p}$.

Consider the step, when we descend from $S_{\Psi,tb_1\ldots b_{i-1}}$ down to $S_{\Psi,tb_1\ldots b_i}$. Let $\Pi_{\Psi,tb_1\ldots b_{i-1}}$ and $\Pi_{\Psi,tb_1\ldots b_i}$ denote the associated permutations. If $b_i = 0$, we follow the 0-path, and $\Pi_{\Psi,tb_1\ldots b_i}$ will be identical to $\Pi_{\Psi,tb_1\ldots b_{i-1}}$. Thus, the i-th entry in $\Pi_{\Psi,tb_1\ldots b_i}$ remains with its original sign. On the other hand, if $b_i = 1$, we follow the 1-path, and so the d-th entry in $\Pi_{\Psi,tb_1\ldots b_{i-1}}$ is negated and cyclically shifted to the i-th position in $\Pi_{\Psi,tb_1\ldots b_i}$. Thus, the i-th entry in $\Pi_{\Psi,tb_1\ldots b_i}$ has changed its original sign. Since the subsequent steps apply cyclical shifts only to the last $(d-i)$ entries of the permutation, the i-th location remains the same until we descend down to $S_{\Psi,p}$. And so, looking at whether the i-th entry in $\Pi_{\Psi,p}$ has changed its sign or not, we can determine b_i. □

$$\overbrace{[+1-4-3+2]}^{S_{\Psi,t}} \xrightarrow{1} [-2+1-4-3] \xrightarrow{0} [-2+1-4-3] \xrightarrow{1} \overbrace{[\underbrace{-2}_{b_1=1}\ \underbrace{+1}_{b_2=0}\ \underbrace{+3}_{b_3=1}\ -4]}^{S_{\Psi,p}}$$

To illustrate the proof more concretely, consider the above example where $\ell = 3$. Note that $(o_i)_1^d = (+1, +1, -1, -1)$. Since $o_2 = +1$, it follows that the 2 entry has a positive sign in $\Pi_{\Psi,t}$. On descending to the 1-child, this entry is negated and then shifted to the first element in $\Pi_{\Psi,p}$. After this its value is fixed. Next, on descending to its 0-child, the 1 entry remains positive and is then fixed in the second position. Finally, on descending to its 1-child, the 3 entry is negated and placed at the third location. And so, based on the final sequence, we can infer that the tree path leading from $S_{\Psi,t}$ to $S_{\Psi,p}$ is 101.

Lemma 4 can be applied as follows to determine the LPT code for the parent of a nonroot simplex S. Given S's LPT code, (ℓ, Π, Φ), we distinguish two cases, depending on its level. If ℓ is nonzero, then its parent's level is $\ell' = \ell - 1$ and otherwise its parent's level is $\ell' = d-1$. If ℓ is nonzero, then the orthant vector $\mathbf{o}$ of the lemma is the last entry of Φ. We apply this lemma to determine whether S is a 0- or 1-child. From Eq. (7) and Theorem 1 we know that, if it is a 0-child, it has the same permutation code as its parent, and otherwise its parent's permutation code is $\Pi \circ \Sigma_{\ell'}^{-1}$. Its parent has the same orthant list. On the other hand, if $\ell = 0$ then $\mathbf{o}$ is the second to last entry of Φ. Again we apply the lemma to determine whether S is a 0- or 1-child, and derive its parent's permutation code. The last entry of S's orthant list is removed to form the orthant list of its parent. This can be computed in $O(d)$ time.

5.2. *Point location and interpolation queries*

In this section we consider how to compute the LPT code of the leaf simplex of the decomposition tree that contains a given query point $\mathbf{q} = (q_i)_{i=1}^d$. We assume that $\mathbf{q}$ lies in the base hypercube, that is, $-1 \leq q_i \leq 1$. If $\mathbf{q}$ lies on a face between two simplices, we will choose one arbitrarily. Locating the leaf simplex that contains the query point is performed by a descent in the decomposition tree. Because the decomposition is based on repeated edge bisections, there is a very elegant way in which to locate the containing simplex for a query point by maintaining the query point's barycentric coordinates (defined below) with respect to its enclosing simplex.

The point-location process starts by determining the root simplex $S_{\Psi,\emptyset}$ that contains $\mathbf{q}$. It is easy to see that a point $\mathbf{q}$ in the base hypercube lies in the base reference simplex, Δ_0, if and only if its coordinate vector is sorted in decreasing order. It follows that determining the permutation Ψ of the root simplex reduces to sorting the coordinates of $\mathbf{q}$ in decreasing order and setting Ψ to the permutation that produces this sorted order. It is an easy matter to modify virtually any efficient sorting algorithm to produce a procedure *sortDescending* that computes this permutation in $O(d \log d)$ time.

Given the root simplex $S_{\Psi,\emptyset}$ containing the query point $\mathbf{q}$, let $\mathbf{v}_i$ denote the i-th vertex of this simplex. It is well known that $\mathbf{q}$ can be uniquely expressed as a convex combination of the vertices of $S_{\Psi,\emptyset}$, that is, $\mathbf{q} = \sum_i \alpha_i \mathbf{v}_i$ where $\alpha_i \geq 0$ and $\sum_i \alpha_i = 1$. These coefficients are called the *barycentric coordinates* of $\mathbf{q}$ with respect to this simplex.[35] Let $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$ denote this $(d+1)$-vector. To start the point-location process we compute the barycentric coordinates of $\mathbf{q}$ with respect to its containing root simplex, which we do with the aid of the following lemma.

Lemma 6. *Let $(q_1, q_2, \ldots, q_d)$ be the cartesian coordinates of a point $\mathbf{q}$ lying in the base reference simplex, Δ_0. Then, the barycentric coordinates of $\mathbf{q}$ with respect this*

simplex are $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$, *where*

$$\alpha_i = \begin{cases} (1 - q_1)/2 & \text{if } i = 0 \\ (q_i - q_{i+1})/2 & \text{if } 0 < i < d \\ (q_d + 1)/2 & \text{if } i = d. \end{cases} \tag{23}$$

Proof. To simplify notation, it is convenient to define $q_0 = 1$ and $q_{d+1} = -1$. Thus we have $\alpha_i = (q_i - q_{i+1})/2$ for $0 \le i \le d$. Because $\mathbf{q}$ lies in the base reference simplex, from the observations above we have

$$-1 = q_{d+1} \le q_d \le \ldots \le q_1 \le q_0 = 1. \tag{24}$$

It follows easily that $\alpha_i \ge 0$ for all i, and $\sum_i \alpha_i = 1$, which constitute the first two conditions for barycentric coordinates. To establish that $\mathbf{q}$ is the desired linear combination, recall from Section 3 that the i-th vertex of base reference simplex is $\mathbf{v}_i = \sum_{j=1}^{i} \mathbf{e}_j - \sum_{j=i+1}^{d} \mathbf{e}_j$. It follows that for $1 \le j \le d$, the j-th coordinate of the linear combination $\sum_i \alpha_i \mathbf{v}_i$ is

$$\frac{1}{2}\left(\sum_{0 \le i < j} -(q_i - q_{i+1}) + \sum_{j \le i \le d} (q_i - q_{i+1}) \right).$$

By simple telescoping we see that most of the terms of the two sums cancel leaving $\frac{1}{2}(-(q_0 - q_j) + (q_j - q_{d+1}))$. By substituting the definitions of q_0 and q_{d+1} we find that the j-th coordinate of the linear combination is q_j, as desired. □

Combining Lemma 6 with the observation above about using sorting to determine the root simplex containing $\mathbf{q}$, it follows easily that procedure *findRoot* shown in Fig. 9 correctly computes these initial barycentric coordinates.

After this initialization, we recursively descend the hierarchy until finding a leaf simplex. We use the barycentric coordinates of $\mathbf{q}$ relative to the current simplex to determine in which child it resides. Then we generate the barycentric coordinates of $\mathbf{q}$ with respect to this child. The remainder of the point-location algorithm is presented in procedure *search* given in Fig. 9. Its correctness is established formally in Lemma 7. To simplify the presentation, we have omitted the orthant list processing, but it is essentially the same as in the code block given in Fig. 6.

Lemma 7. *Consider a nonleaf simplex* $S_{\Psi,p}$ *of the hierarchy at level* ℓ *with the associated permutation code* $\Pi_{\Psi,p} = [\pi_i]_{i=1}^d$. *Suppose that* $\mathbf{q}$ *lies within this simplex with the barycentric coordinates* $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$.

- *If* $\alpha_\ell \le \alpha_d$, *then* $\mathbf{q}$ *lies in the 0-child. Let* $\boldsymbol{\alpha}'$ *be the* $(d+1)$*-vector that is identical to* $\boldsymbol{\alpha}$ *except that* $\alpha'_\ell = 2\alpha_\ell$ *and* $\alpha'_d = \alpha_d - \alpha_\ell$. *Then the barycentric coordinate vector of* $\mathbf{q}$ *relative to this child is* $\boldsymbol{\alpha}'$.
- *Otherwise,* $\mathbf{q}$ *lies in the 1-child. Let* Σ'_ℓ *be a* $(d+1)$*-permutation that shifts the last* $d+1-\ell$ *coordinates circularly one position to the right. Let* $\boldsymbol{\alpha}'$ *be the* $(d+1)$*-vector that is identical to* $\boldsymbol{\alpha}$ *except that* $\alpha'_d = 2\alpha_d$ *and* $\alpha'_\ell = \alpha_\ell - \alpha_d$. *Then the barycentric coordinate vector of* $\mathbf{q}$ *relative to this child is* $\boldsymbol{\alpha}'\Sigma'_\ell$.

Input: a query point q lying in the base hypercube.
Output:
Ψ: denotes the root simplex that contains q.
α: barycentric coordinates of q with respect to root simplex $S_{\Psi,\emptyset}$.
findRoot($\mathbf{q}$)
$\Psi \leftarrow sortDescending((q)_{i=1}^{d})$
$\alpha_0 \leftarrow (1 - q_{\psi_1})/2$
for $(0 < i < d)$ $\alpha_i \leftarrow (q_{\psi_i} - q_{\psi_{i+1}})/2$
$\alpha_d \leftarrow (q_{\psi_d} + 1)/2$
return$(\Psi, \boldsymbol{\alpha})$

Input:
q: query point.
(ℓ, Π): current simplex that contains q.
α: current barycentric coordinates of q with respect to (ℓ, Π).
Output: (ℓ, Π): the leaf simplex that contains q.
search($\mathbf{q}, (\ell, \Pi), \boldsymbol{\alpha}$)
if ((ℓ, Π) is a leaf) **return** (ℓ, Π)
$\boldsymbol{\alpha}' \leftarrow \boldsymbol{\alpha}$
if $(\alpha_\ell \leq \alpha_d)$
$\alpha'_\ell \leftarrow 2\alpha_\ell$; $\alpha'_d \leftarrow \alpha_d - \alpha_\ell$
return *search*$(\mathbf{q}, ((\ell + 1) \bmod d, \Pi), \boldsymbol{\alpha}')$
else
$\alpha'_d \leftarrow 2\alpha_d$; $\alpha'_\ell \leftarrow \alpha_\ell - \alpha_d$
return *search*$(\mathbf{q}, ((\ell + 1) \bmod d, \Pi \circ \Sigma_\ell), \boldsymbol{\alpha}'\Sigma'_\ell)$

Fig. 9. The procedures *findRoot* and *search*, which are used to locate a query point $\mathbf{q}$ in the hierarchy. The permutation Σ'_ℓ is defined in Lemma 7 and the permutation Σ_ℓ was given in Section 4, Eq. 7. Note that *search* is initially called as *search*$(q, (0, \Psi), \alpha)$ where $(\Psi, \alpha) \leftarrow$ *findRoot*(q).

Proof. Let $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^{d}$ denote $\mathbf{q}$'s barycentric coordinates with respect to $S_{\Psi,p}$. Let $\mathbf{v}_i$ denote the i-th vertex of $S_{\Psi,p}$. Recall that $\mathbf{m} = \frac{\mathbf{v}_\ell + \mathbf{v}_d}{2}$ is the newly created vertex that bisects this simplex and that

$$\begin{aligned} S_{\Psi,p} &= [\mathbf{v}_0 \ldots \mathbf{v}_\ell \ldots \mathbf{v}_d]^T, \\ S_{\Psi,p0} &= [\mathbf{v}_0 \ldots \mathbf{v}_{\ell-1}\ \mathbf{m}\ \mathbf{v}_{\ell+1} \ldots \mathbf{v}_d]^T, \\ S_{\Psi,p1} &= [\mathbf{v}_0 \ldots \mathbf{v}_{\ell-1}\ \mathbf{m}\ \mathbf{v}_\ell \ldots \mathbf{v}_{d-1}]^T \end{aligned}$$

And so, $\mathbf{v}_\ell = 2\mathbf{m} - \mathbf{v}_d$, and $\mathbf{v}_d = 2\mathbf{m} - \mathbf{v}_\ell$. Thus, $\mathbf{q}$ can be written in terms of

barycentric coordinates as,

$$\begin{aligned}\mathbf{q} &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_\ell\mathbf{v}_\ell + \ldots + \alpha_d\mathbf{v}_d \\ &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_{\ell-1}\mathbf{v}_{\ell-1} + \alpha_\ell(2\mathbf{m} - \mathbf{v}_d) + \alpha_{\ell+1}\mathbf{v}_{\ell+1} + \ldots + \alpha_d\mathbf{v}_d \\ &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_{\ell-1}\mathbf{v}_{\ell-1} + 2\alpha_\ell\mathbf{m} + \alpha_{\ell+1}\mathbf{v}_{\ell+1} + \ldots + (\alpha_d - \alpha_\ell)\mathbf{v}_d,\end{aligned}$$

and similarly,

$$\begin{aligned}\mathbf{q} &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_\ell\mathbf{v}_\ell + \ldots + \alpha_d\mathbf{v}_d \\ &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_{\ell-1}\mathbf{v}_{\ell-1} + \alpha_\ell\mathbf{v}_\ell + \ldots + \alpha_d(2\mathbf{m} - \mathbf{v}_\ell) \\ &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_{\ell-1}\mathbf{v}_{\ell-1} + 2\alpha_d\mathbf{m} + (\alpha_\ell - \alpha_d)\mathbf{v}_\ell + \ldots + \alpha_{d-1}\mathbf{v}_{d-1}.\end{aligned}$$

By the nonnegativity requirement of barycentric coordinates we see that if $(\alpha_d - \alpha_\ell) \geq 0$, then $\mathbf{q}$ lies in $S_{\Psi,p0}$, and otherwise it lies in $S_{\Psi,p1}$. From the above equations, we can also see the barycentric coordinates of $\mathbf{q}$ with respect to $S_{\Psi,p0}$ or $S_{\Psi,p1}$. □

In summary, the point-location process works as follows. Given a query point $\mathbf{q}$, the *findRoot* procedure is called to determine the root simplex Ψ containing $\mathbf{q}$ and its barycentric coordinates $\boldsymbol{\alpha}$. Then it invokes the recursive procedure *search*$(0, \Psi, \boldsymbol{\alpha})$ to locate $\mathbf{q}$ within the appropriate root simplex. The recursion stops when a leaf simplex has been reached.

Given the containing leaf simplex, interpolation queries can be answered as follows. We access the stored vector field values at each of the simplex vertices, and then weight these values according to the barycentric coordinates of $\mathbf{q}$. The result is a piecewise linear, continuous interpolant.

It is interesting to note that normally one would expect $O(d)$ time to determine the child containing the query point since it involves determining which side q lies relative to a $(d-1)$-dimensional hyperplane that splits the simplex. A nice feature of using barycentric coordinates (as seen in procedure *search*) is that this determination can be made based on just two of the barycentric coordinates at each level of the search, irrespective of the dimension.

This simple sequential search makes as many memory accesses to the decomposition tree as the depth of the final leaf simplex that contains $\mathbf{q}$. A more efficient procedure in terms of global memory accesses would be to employ a doubling binary search, which computes (using only local memory) the LPT codes for the simplices at depths 0, 1, 2, 4, 8, and so on, until first finding a depth whose simplex does not exist in the hierarchy. We then use standard binary search to determine the exact depth of the leaf simplex that contains $\mathbf{q}$. Although the computation of the LPT codes is performed sequentially in time linear in the depth of the final simplex, the number accesses to the simplex decomposition tree is only logarithmic in the final depth. Thus, the running time is $O(dD)$, where D is the maximum depth of the tree, and $O(\log D)$ global memory accesses are made.

6. Neighbors in the Simplicial Complex

As we mentioned earlier, when a simplex of the decomposition tree is bisected, it is necessary to bisect some of its neighbors in order to guarantee that the final subdivision is a simplicial complex. Henceforth, let us assume that the simplex tree decomposition has been constructed so that the underlying subdivision is a simplicial complex. In order to know what additional simplices must be bisected, it is necessary to compute neighbors within the complex. Two simplices are *neighbors* if they share a common $(d-1)$-dimensional face. In addition to this major need for fast neighbor computation, in general, computing facet neighbors of a simplex efficiently is of great interest for many applications that require moving along adjacent simplices, such as direct volume rendering and isosurface extraction techniques.

Consider a simplex S in the complex defined by the decomposition tree. For $0 \leq i \leq d$, let $\mathbf{v}_i$ denote its i-th vertex. Exactly one $(d-1)$-face of S does not contain $\mathbf{v}_i$. If this face is not on the boundary of the base hypercube, its neighbor exists in the complex. If so, we define $N^{(i)}(S)$ to be the same depth neighboring simplex to S lying on the opposite side of this face. Let (ℓ, Π, Φ) denote the LPT code for S and let $(\ell^{(i)}, \Pi^{(i)}, \Phi^{(i)})$ denote the LPT code for $N^{(i)}(S)$. We present rules here for computing LPT codes of these neighbors.

The rules compute the LPT code for the neighbor simplex at the same depth as S, and hence $\ell^{(i)} = \ell$. Of course, this simplex need not be in the decomposition tree because its parent may not yet have been bisected. In fact, in a compatible subdivision, a $(d-1)$-face neighbor of S could also appear at one level higher or one level lower than S. We also show how to compute the LPT codes of those neighbors.

6.1. *Neighbor permutation code*

Each neighbor's permutation code is determined by applying one of a set of special signed permutations to Π. The permutation depends on whether S is a 0-child or a 1-child, which can be determined using the test given in Section 5.1. These permutations are illustrated in Fig. 10, and include the following:

- $\Gamma_{\text{NEG},1}$, negates the first element,
- $\Gamma_{\text{RGT},\ell}$, shifts the last $d-\ell$ elements cyclically one position to the right and negates the element that was wrapped around,
- $\Gamma_{\text{LFT},\ell}$ shifts the last $d-\ell$ elements cyclically one position to the left and negates the element that was wrapped around,
- $\Gamma_{\text{SWP},i}$, swaps elements i and $i+1$,
- $\Gamma_{\text{NSW},\ell}$, swaps and negates elements ℓ and d.

The neighbor rules are given in Theorem 2. A number of the rules involve the parent's level, and so to condense notation, we define $\ell^- = (\ell - 1) \bmod d$ and $\ell^* = \ell^- + 1$. Observe that $\ell^- = \ell - 1$ and $\ell^* = \ell$, except when $\ell = 0$, in which case they are larger by d. These can be computed in $O(d)$ time, and in fact in $O(1)$ time if permutations are encoded in look-up tables as described below.

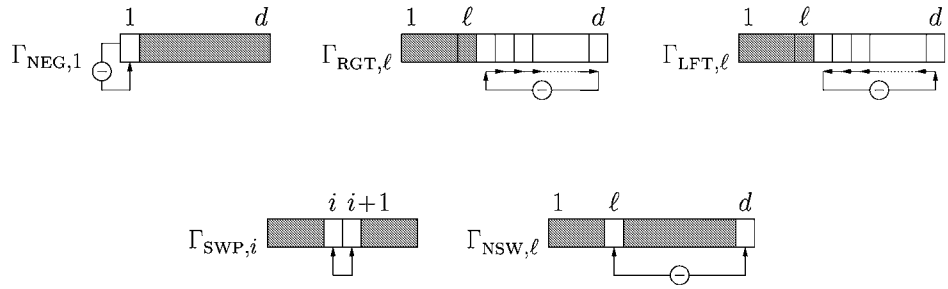

Fig. 10. Neighbor permutations. (The circle with a minus sign indicates that the element is negated.)

Theorem 2. *Let S denote a simplex at level ℓ, and let Π denote its associated permutation code. For $0 \leq i \leq d$, if the neighbor $N^{(i)}(S)$ exists, then its permutation code is given by the following rules:*

$$
\begin{array}{lll}
\textbf{\textit{if}}\ (S \text{ is a 0-child}): & N^{(0)}(S): & \Pi^{(0)} = \Pi \circ \Gamma_{\text{NEG},1} \\
& N^{(i)}(S): (0 < i < d) & \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i} \\
& N^{(d)}(S): & \Pi^{(d)} = \Pi \circ \Gamma_{\text{RGT},\ell^-} \\
\textbf{\textit{if}}\ (S \text{ is a 1-child}): & N^{(0)}(S): & \Pi^{(0)} = \Pi \circ \Gamma_{\text{NEG},1} \\
& N^{(\ell^*)}(S): & \Pi^{(\ell^*)} = \Pi \circ \Gamma_{\text{LFT},\ell^-} \\
& N^{(i)}(S): (0 < i < d, i \neq \ell^*) & \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i} \\
& N^{(d)}(S): (d \neq \ell^*) & \Pi^{(d)} = \Pi \circ \Gamma_{\text{NSW},\ell}
\end{array}
\tag{25}
$$

The proof of this theorem involves a straightforward but lengthy induction argument involving many cases. The proof is presented in the Section 6.5.

Implementation Issues: In our implementation, we treat the signed permutation component as a reflection and a permutation separately, as in the initial description given in Section 3.1. Recall that the reflection could be one of 2^d reflections, and the permutation could be one of $d!$ permutations. Both the reflection and the permutation are represented by a unique integer identifier. The operations defined on the permutation-reflection component such as cyclical shifts and swaps are performed through use of tables, which can be computed once the dimension d is given. Each possible operation is also given a unique integer identifier. We precompute two tables, one for permutations, and one for reflections. There is an entry for each possible permutation/reflection and each possible operation combination. The permutation/reflection integer identifier and the operation identifier could be used as indices to these tables to get the integer identifier of the resulting permutation/reflection. By these tables, all operations are performed in $O(1)$ time.

6.2. *Neighbor orthant list*

In order to compute the orthant list component of the neighbor from the LPT code of S, we distinguish three cases:

(i) If $\ell \neq 0$ or $1 \leq i < d$, $N^{(i)}(S)$, is in the same final orthant as S, and so $\Phi^{(i)} = \Phi$.
(ii) If $\ell = 0$, $N^{(d)}$ is in a different orthant than S, but, $N^{(d)}$ is the sibling of S in this case. Thus, Φ and $\Phi^{(d)}$ differ only in their last element, which is $orth(\Pi^{(d)})$ in $\Phi^{(d)}$. Thus the orthant list can be updated in $O(d)$ time in this case.
(iii) The only remaining case is $\Phi^{(0)}$. This case is the most complex because the final enclosing quadtree box of $N^{(0)}(S)$ is disjoint from S's final quadtree box. Further, it may be arbitrarily far away, in the sense that the least common ancestor of the two nodes may be the root of the tree. This case is described below.

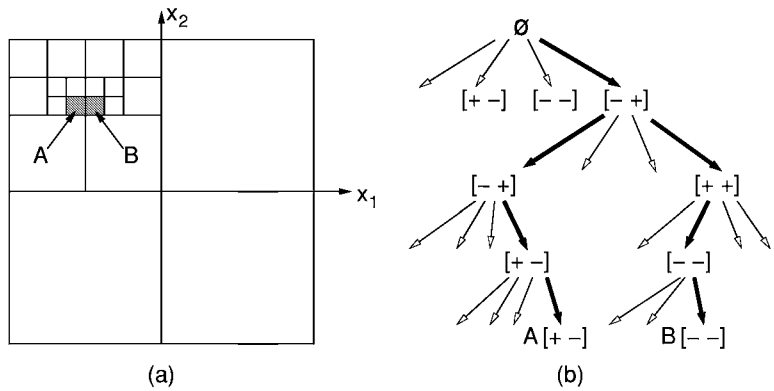

Fig. 11. Orthant B is a neighbor of orthant A in $+X_1$ direction. (a) The quadtree-like subdivision of space. (b) The corresponding tree representation.

To compute $\Phi^{(0)}$, we use a method similar to the one for computing neighbor quadrants in quadtrees.[21] In our representation, the path from the root to the orthant is the list of orthants in Φ. Consider the 2-dimensional example in Fig. 11(a). The orthants A and B are neighbors, and their associated orthant lists, written as column vectors are as follows. ($+1$ and -1 are denoted with their signs only, as $+$ and $-$, respectively.)

$$\Phi_A = \left\langle \begin{pmatrix} - \\ + \end{pmatrix} \begin{pmatrix} - \\ + \end{pmatrix} \begin{pmatrix} + \\ - \end{pmatrix} \begin{pmatrix} + \\ - \end{pmatrix} \right\rangle$$

$$\Phi_B = \left\langle \begin{pmatrix} - \\ + \end{pmatrix} \begin{pmatrix} + \\ + \end{pmatrix} \begin{pmatrix} - \\ - \end{pmatrix} \begin{pmatrix} - \\ - \end{pmatrix} \right\rangle.$$

It is easy to see that, paths to A and B have a common prefix corresponding to their common ancestors, that, is the orthant $(-,+)$ in the example. Orthant

entries are identical for the remainder of the paths except that one coordinate (in our example, X_1) is complemented. Fig. 11(b) illustrates the paths to A and B. The axis which has to be complemented depends on which neighbor we are looking for. This generalizes to a d-cube which is subdivided in a quadtree-like manner.

The problem of finding a neighbor orthant can be stated as follows: Given an orthant A whose path from the root is represented by Φ_A, and a direction defined as a 2-tuple (D, X_i) where X_i is the i-th coordinate axis, and $D \in \{-, +\}$ represents the direction of X_i, find the path to the neighbor orthant B of equal size located in the given direction with respect to A.

Similar to the algorithm described for quadtrees by Samet,[21] the algorithm for finding the path to the neighbor orthant B is a two-step process. Let the direction of the neighbor be (D, X_i). In terms of the tree representation, we first perform a bottom-up traversal starting from A, until we find the closest ancestor, C such that C is the parent of the lowest ancestor of A whose i-th coordinate is the complement of D. This is the desired common ancestor of A and B. If no such ancestor exists, then the desired neighbor B is outside the bounding box, and so, it does not exist. Otherwise, let the path from C to A be denoted as P_{CA}. In the next step, we complement the X_i coordinates in P_{CA}, to get the path from C to B, P_{CB}. Since the path from the root to C, Φ_C is common for both A and B, $\Phi_B = \Phi_C + P_{CB}$. Thus, finding the common ancestor C by bottom-up traversal corresponds to processing Φ_A back-to-front, complementing the X_i coordinate of each orthant, until we come across an orthant whose X_i coordinate is $-D$. We complement this coordinate as well. This completes the complementing part. Rest of the list remain the same. The resulting list is Φ_B.

Thus, in order to compute $\Phi^{(0)}$, which represents the path from the root to the final orthant of $N^{(0)}(S)$, all we need is to determine which direction $N^{(0)}(S)$ is located with respect to S. Consider the Π and $\Pi^{(0)}$ corresponding to S and $N^{(0)}(S)$ respectively. By the given neighbor rules, these two permutation-reflection codes differ only in the sign of their first element. This is the sign corresponding to the $X_{|\pi_1|}$ axis, given that $\Pi = [\pi_i]_1^d$ is the code for S. The sign of π_1 determines in which direction of $X_{|\pi_1|}$ axis S resides in its final orthant. And so, the neighbor $N^{(0)}(S)$ is also in that direction. Thus, the axis component of the direction is $X_{|\pi_1|}$, and the sign component of the direction is $sign(\pi_1)$.

Implementation Issues: This operation can be implemented very rapidly through a simple trick with bit manipulations. The neighbor computation[21] essentially involves an operation, which is applied to a bit string that consists of the i-th coordinate of each entry of the orthant list. Recall from our earlier discussion of implementation issues, that the orthant list is stored as d separate bit strings, one per coordinate, and packed into machine words as binary numbers. The key operation needed for the neighbor computation involves complementing a maximal trailing sequence of matching bits. For example, given a bit string of the form $w10^k$, for $w \in \{0,1\}^*$, the desired result is $w01^k$ (and vice versa). By packing these bits

into a single word, we can compute this function with a single arithmetic operation by subtracting (or adding) 1 from the resulting binary number. (Similar tricks has been applied elsewhere in the context of neighbor finding.[29]) Under the assumption that the machine's word size is $\Omega(D/d)$, where D is the maximum depth of any simplex, the orthant list for the neighbor can be computed in $O(1)$ time.

6.3. *Compatible refinement and the simplicial complex*

We have earlier mentioned that *compatibility* is important, since otherwise, cracks occur along faces of the subdivision, which in turn present problems when using the mesh for interpolation. In order to keep the subdivision compatible at all times, whenever a simplex is bisected a series of bisections will be triggered in other simplices. Hebert[14] and Maubach[7] describe the process for their systems. For completeness we include a short description here as well.

Consider a simplex S which is about to be bisected, and let e denote the next edge of S to be split. The simplices of the subdivision that share this edge, denoted $E_e(S)$, must be bisected as well. The rules given in Section 6 provide a means to locate same depth neighboring simplices that share a common $(d-1)$-face with S, that is, the same depth *facet neighbors* of S. Let $N_e(S)$ denote the same depth facet neighbors of S that contain the edge e. In order to access all the simplices of $E_e(S)$ we compute facet neighbors recursively. The algorithm was given by Maubach,[7] and is shown as the recursive function *compatBisect* in the codeblock shown in Fig. 12. The procedure *simpleBisect* performs the basic bisection step described in Section 3.2.

```
compatBisect(S)
    mark S as pending
    for (S' ∈ N_e(S))
        if ( S' does not exist )
            compatBisect(parent(S'))  // now S' exists
        if ( S' is a leaf and not marked as pending )
            compatBisect(S')  // bisect S' and its neighbors
    simpleBisect(S)
```

Fig. 12. Procedure *compatBisect*, which bisects simplex S and bisects surrounding simplices to maintain compatibility.

Maubach proved that in a compatible subdivision, the facet neighbors of S needed to be bisected in this refinement, either appear at the same depth as S or one level closer to the root.[7] For this reason, if the *compatBisect* procedure does not find a simplex S' in the tree, then it knows that its parent exists, and bisecting the parent will being S' into existence. Note that the bisection of the parent may trigger recursive bisections on levels $\ell - 1$ and $\ell - 2$, and so on.

6.4. *Neighbors at different depths*

The neighbor rules of Theorem 2 provide the LPT code for the same depth neighbors. However, as mentioned above, in a compatible subdivision, a neighbor could possibly appear one level closer or one level further from the root, that is, some neighbors of a simplex S_p at depth $|p|$, could appear at depths $|p|-1$ or $|p|+1$. We can categorize the neighbors of a simplex into two groups: neighbors that share the edge to be bisected and those that do not. A neighbor sharing the edge to be bisected is either at depth $|p|$ or at depth $|p|-1$, and such a neighbor at depth $|p|-1$ is the parent of the same depth neighbor which did not come into existence yet. And so, for a neighbor at depth $|p|-1$, we first compute the LPT code for the same depth neighbor by the rules of Theorem 2, and if the same depth neighbor does not exist in the tree, we compute its parent's LPT code as described in Section 5.

In addition, any (d-1)-face neighbor of S_p that does not share the edge to be bisected could possibly be at depth $|p|+1$. Specifically, same depth neighbors $N^{(\ell)}(S_p)$ and $N^{(d)}(S_p)$, might have been bisected without triggering bisection of S_p, and so, one of their children will now share a face with S_p. Moreover, the child of $N^{(\ell)}(S_p)$ or $N^{(d)}(S_p)$ that shares a face with S_p, is the same depth neighbor of one of the children of S_p. So, we can compute a neighbor at depth $|p|+1$ by computing the appropriate same depth neighbor of one of the children of S_p. Formally,

if($N^{(\ell)}(S_p)$ is a bisected simplex)
    $N^{(\ell)}(S_{p0})$ is the ℓ-th neighbor of S_p.
if($N^{(d)}(S_p)$ is a bisected simplex)
    $N^{(\ell)}(S_{p1})$ is the d-th neighbor of S_p.

It can be easily shown that these neighbors cannot exist at depths higher than $|p|+1$. Intuitively, same depth neighbor $N^{(\ell)}(S_p)$ (resp. $N^{(d)}(S_p)$) have exactly one vertex different from S_p. Let that vertex be $\mathbf{u}$. It can be shown that when $N^{(\ell)}(S_p)$ (resp. $N^{(d)}(S_p)$) is bisected, $\mathbf{u}$ is one of the endpoints of the bisected edge. So, one of the children of $N^{(\ell)}(S_p)$ (resp. $N^{(d)}(S_p)$) will have two vertices different from S_p, and cannot be a neighbor. The other child has exactly one vertex ($\mathbf{u}$) different from S_p, thus is a neighbor of S_p. If that child is further bisected however, its children will have an additional new vertex created by bisection of an edge which does not contain $\mathbf{u}$, hence these children at depth $|p|+2$ cannot be neighbors of S_p.

6.5. *Proof of neighbor-rules theorem*

In this section we present a proof of Theorem 2. The following notation will be used throughout the proof.

- S denotes any simplex.
- $S^{(i)} = N^{(i)}(S)$, i.e. the i-th neighbor of S.
- Π and $\Pi^{(i)}$ denote the signed permutation code associated with S and $S^{(i)}$ respectively.
- For $c \in \{0,1\}$:

- S_c denotes the c-child of S, and $S_c^{(i)}$ denotes the c-child of $S^{(i)}$.
- $(S_c)^{(i)}$ denotes the i-th neighbor of child S_c.
- Π_c, $\Pi_c^{(i)}$, and $(\Pi_c)^{(i)}$ denote the signed permutation codes for S_c, $S_c^{(i)}$, and $(S_c)^{(i)}$, respectively.

- $\mathbf{m}$ and $\mathbf{m}'$ are used to denote new vertices generated by bisection.
- $\mathbf{u}$ is used to denote the vertex that differs in the neighbor simplex.

Induction Hypothesis: Let $S = [\mathbf{v}_0 \dots \mathbf{v}_\ell \dots \mathbf{v}_d]^T$ be a simplex at level $\ell = |p| \bmod d$. Let $\ell^- = (\ell - 1) \bmod d$ and $\ell^* = \ell^- + 1$. Let $\mathbf{u}$ be the vertex of the neighbor simplex that is not a vertex of S. The rules of the theorem can be stated more explicitly as:

if (S is a 0-child)

$$\begin{aligned} S^{(0)} &= [\mathbf{u}\ \mathbf{v}_1 \dots \mathbf{v}_d]^T & \Pi^{(0)} &= \Pi \circ \Gamma_{\text{NEG},1} \\ S^{(i)} &= [\mathbf{v}_0 \dots \mathbf{v}_{i-1}\ \mathbf{u}\ \mathbf{v}_{i+1} \dots \mathbf{v}_d]^T, \ (0 < i < d) & \Pi^{(i)} &= \Pi \circ \Gamma_{\text{SWP},i} \\ S^{(d)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-}\ \mathbf{u}\ \mathbf{v}_{\ell^*} \dots \mathbf{v}_{d-1}]^T & \Pi^{(d)} &= \Pi \circ \Gamma_{\text{RGT},\ell^-} \end{aligned}$$

if (S is a 1-child)

$$\begin{aligned} S^{(0)} &= [\mathbf{u}\ \mathbf{v}_1 \dots \mathbf{v}_d]^T & \Pi^{(0)} &= \Pi \circ \Gamma_{\text{NEG},1} \\ S^{(\ell^*)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-}\ \mathbf{v}_{\ell^-+2} \dots \mathbf{v}_d\ \mathbf{u}]^T & \Pi^{(\ell^*)} &= \Pi \circ \Gamma_{\text{LFT},\ell^-} \\ S^{(i)} &= [\mathbf{v}_0 \dots \mathbf{v}_{i-1}\ \mathbf{u}\ \mathbf{v}_{i+1} \dots \mathbf{v}_d]^T, \ (0 < i < d, i \neq \ell^*) & \Pi^{(i)} &= \Pi \circ \Gamma_{\text{SWP},i} \\ S^{(d)} &= [\mathbf{v}_0 \dots \mathbf{v}_{d-1}\ \mathbf{u}]^T, \ (d \neq \ell^*) & \Pi^{(d)} &= \Pi \circ \Gamma_{\text{NSW},\ell} \end{aligned}$$

Basis: We show that the neighbor rules hold for the $d!$ root simplices. Root simplices are all at level 0, and the rules are the same whether the simplex is a 0-child, or a 1-child. Let S denote any root simplex whose $\Pi = [\pi_1 \dots \pi_i \ \ \pi_{i+1} \dots \pi_d]$.

- For all root simplices, $S^{(0)} = \emptyset$, and $S^{(d)} = \emptyset$, that is, the 0-th and the d-th neighbors do not exist. This is because the faces lying opposite these vertices are faces of the reference hypercube, and so the neighbors lying opposite these vertices are outside the reference hypercube.
- We will show that the other neighbors, $S^{(i)}$, for $0 < i < d$, are obtainable by swaps. Recall that the base simplex $S_\emptyset$ can be represented as,

$$S_\emptyset = [\mathbf{y}_1 \dots \mathbf{y}_d], \quad \text{where} \quad \mathbf{y}_i = \begin{bmatrix} y_{i,0} \\ \vdots \\ y_{i,i-1} \\ y_{i,i} \\ \vdots \\ y_{i,d} \end{bmatrix} = \begin{bmatrix} -1 \\ \vdots \\ -1 \\ 1 \\ \vdots \\ 1 \end{bmatrix},$$

and any root simplex S can be written as a column permutation π of $S_\emptyset$, that is, $S = [\mathbf{y}'_1 \dots \mathbf{y}'_d]$, where $\mathbf{y}'_{\pi_i} = \mathbf{y}_i$. Letting $j = \pi_i$ and $k = \pi_{i+1}$, we have $\mathbf{y}'_j = \mathbf{y}_i$ and $\mathbf{y}'_k = \mathbf{y}_{i+1}$, and so the associated permutation code is

$$\Pi = [\pi_1 \dots \pi_{i-1} \quad j \quad k \quad \pi_{i+2} \dots \pi_d]$$

It is easy to see that swapping columns $\mathbf{y}'_j$ and $\mathbf{y}'_k$ of S gives us another valid root simplex, S', that differs from S only in the i-th row, that is, the i-th vertex. Because it differs only in this one vertex, it follows that S' is the i-th neighbor of S, that is, $S' = S^{(i)}$. Let Π' denote the signed permutation for S'. Then,

$$\Pi' = [\pi_1 \dots \pi_{i-1} \quad k \quad j \quad \pi_{i+2} \dots \pi_d].$$

This shows that the i-th and the $(i+1)$st entries in Π are swapped to obtain Π', and so

$$\Pi^{(i)} = \Pi' = \Pi \circ \Gamma_{\text{SWP},i}.$$

This completes the basis.

Induction Step: Let S be a simplex at level ℓ^- such that the induction hypothesis holds. We will show that the induction hypothesis holds for the two children of S. We consider two cases, for the 0-child and the 1-child.

First, consider the 0-child of S, that is, S_0. Let ℓ denote the level of S_0. Recall that $\ell^- = (\ell - 1) \bmod d$, and $\ell^* = \ell^- + 1$. Letting i denote the neighbor number, there are a number of cases to be distinguished.

Case 1: $(i = 0)$

If $0 < \ell^- \leq d - 1$ then

$$\begin{aligned} S &= [\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, \quad S_0 = [\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \\ S^{(0)} &= [\,\mathbf{u} \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, \quad S_0^{(0)} = [\,\mathbf{u} \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T. \end{aligned}$$

Otherwise if $\ell^- = 0$ then

$$\begin{aligned} S &= [\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T, \quad S_0 = [\mathbf{m} \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T \\ S^{(0)} &= [\,\mathbf{u} \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T, \quad S_0^{(0)} = [\mathbf{m}' \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T. \end{aligned}$$

In either case, $(S_0)^{(0)} = S_0^{(0)}$. And so, $(\Pi_0)^{(0)} = \Pi_0^{(0)} = \Pi^{(0)} = \Pi \circ \Gamma_{\text{NEG},1} = \Pi_0 \circ \Gamma_{\text{NEG},1}$.

Case 2: $(i = d)$

By definition of the bisection rules, the d-th neighbor of S_0 is its sibling, S_1, and so, $(\Pi_0)^{(d)} = \Pi_1 = \Pi_0 \circ \Gamma_{\text{RGT},\ell^-}$.

Case 3: $(0 < i < \ell^-)$

$$\begin{aligned} S &= [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \quad \mathbf{v}_{i+1} \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, \\ S^{(i)} &= [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{u} \quad \mathbf{v}_{i+1} \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, \\ S_0 &= [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \quad \mathbf{v}_{i+1} \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T, \\ S_0^{(i)} &= [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{u} \quad \mathbf{v}_{i+1} \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T. \end{aligned}$$

In this case $(S_0)^{(i)} = S_0^{(i)}$, and so $(\Pi_0)^{(i)} = \Pi_0^{(i)} = \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i} = \Pi_0 \circ \Gamma_{\text{SWP},i}$.

Case 4: $(i = \ell^-, \ \ell^- \neq 0)$ We distinguish two cases, depending on whether S is a 0-child or 1-child.

Case 4a: If S is a 0-child then

$$\begin{aligned} S &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, & S_0 &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \dots \mathbf{v}_d]^T \\ S^{(\ell^-)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{y} \dots \mathbf{v}_d]^T, & S_0^{(\ell^-)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m}' \dots \mathbf{v}_d]^T. \end{aligned}$$

Then, $(S_0)^{(\ell^-)} = S_0^{(\ell^-)}$. And so, $(\Pi_0)^{(\ell^-)} = \Pi_0^{(\ell^-)} = \Pi^{(\ell^-)} = \Pi \circ \Gamma_{\text{SWP},\ell^-} = \Pi_0 \circ \Gamma_{\text{SWP},\ell^-}$.

Case 4b: If S is a 1-child then

$$\begin{aligned} S &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \\ S^{(\ell^-)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^*} \dots \dots \mathbf{v}_d \quad \mathbf{u}]^T \\ S_0 &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \\ S_1^{(\ell^-)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m}' \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T. \end{aligned}$$

Then, $(S_0)^{(\ell^-)} = S_1^{(\ell^-)}$.

$$\begin{aligned} \Pi_0 &= \Pi = [\pi_1 \dots \pi_d], \\ \Pi^{(\ell^-)} &= [\pi_1 \dots \pi_{\ell^- - 1} \quad \pi_{\ell^*} \dots \pi_d \quad -\pi_{\ell^-}], \\ \Pi_1^{(\ell^-)} &= [\pi_1 \dots \pi_{\ell^- - 1} \quad \pi_{\ell^*} \quad \pi_{\ell^-} \quad \pi_{\ell^*+1} \dots \pi_d]. \end{aligned}$$

And so, $(\Pi_0)^{(\ell^-)} = \Pi_1^{(\ell^-)} = \Pi_0 \circ \Gamma_{\text{SWP},\ell^-}$.

Case 5: $(\ell^- < i < d)$

$$\begin{aligned} S &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_d]^T \\ S^{(i)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-1} \quad \mathbf{y} \dots \mathbf{v}_d]^T \\ S_0 &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_d]^T \\ S_0^{(i)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{i-1} \quad \mathbf{y} \dots \mathbf{v}_d]^T. \end{aligned}$$

In this case $(S_0)^{(i)} = S_0^{(i)}$. Thus, $(\Pi_0)^{(i)} = \Pi_0^{(i)} = \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i} = \Pi_0 \circ \Gamma_{\text{SWP},i}$.

This completes the case of the 0-child. Next, consider the 1-child of S, that is, S_1. As before we let ℓ denote the level of S_1, and define $\ell^- = (\ell - 1) \bmod d$ and $\ell^* = \ell^- + 1$. Letting i denote the neighbor number, again, there are multiple cases.

Case 1: $(i = 0)$ We consider two cases depending on the value of ℓ^-.

Case 1a: $(\ell^- = 0)$

$$\begin{aligned} S &= [\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T, & S_1 &= [\mathbf{m} \quad \mathbf{v}_0 \dots \mathbf{v}_{d-1}]^T \\ S^{(d)} &= [\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{d-1} \quad \mathbf{u}]^T, & S_1^{(d)} &= [\mathbf{m}' \quad \mathbf{v}_0 \dots \mathbf{v}_{d-1}]^T. \end{aligned}$$

Then, $(S_1)^{(0)} = S_1^{(d)}$.

$$\begin{aligned} \Pi &= [\pi_1 \dots \pi_d], & \Pi_1 &= [-\pi_d \quad \pi_1 \dots \pi_{d-1}], \\ \Pi^{(d)} &= [\pi_1 \dots \pi_{d-1} \quad -\pi_d], & \Pi_1^{(d)} &= [\pi_d \quad \pi_1 \dots \pi_{d-1}]. \end{aligned}$$

And so, $(\Pi_1)^{(0)} = \Pi_1^{(d)} = \Pi_1 \circ \Gamma_{\text{NEG},1}$.

Case 1b: $(\ell^- \neq 0)$

$$\begin{aligned} S &= [\mathbf{v}_0 \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T \\ S^{(0)} &= [\,\mathbf{y} \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T \\ S_1 &= [\mathbf{v}_0 \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T \\ S_1^{(0)} &= [\,\mathbf{y} \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T. \end{aligned}$$

Then, $(S_1)^{(0)} = S_1^{(0)}$.

$$\begin{aligned} \Pi &= [\pi_1 \ldots \pi_d], & \Pi_1 &= [\pi_1 \ldots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \ldots \pi_{d-1}], \\ \Pi^{(0)} &= [-\pi_1 \ldots \pi_d], & \Pi_1^{(0)} &= [-\pi_1 \ldots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \ldots \pi_{d-1}]. \end{aligned}$$

And so, $(\Pi_1)^{(0)} = \Pi_1^{(0)} = \Pi_1 \circ \Gamma_{\text{NEG},1}$.

Case 2: $(0 < i < \ell^-)$

$$\begin{aligned} S &= [\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{v}_i \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T \\ S^{(i)} &= [\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{y} \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T \\ S_1 &= [\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{v}_i \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T \\ S_1^{(i)} &= [\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{y} \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T. \end{aligned}$$

Then, $(S_1)^{(i)} = S_1^{(i)}$.

$$\begin{aligned} \Pi &= [\pi_1 \ldots \pi_d], \\ \Pi_1 &= [\pi_1 \ldots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \ldots \pi_{d-1}], \\ \Pi^{(i)} &= [\pi_1 \ldots \pi_{i-1} \quad \pi_{i+1} \quad \pi_i \quad \pi_{i+2} \ldots \pi_d], \\ \Pi_1^{(i)} &= [\pi_1 \ldots \pi_{i-1} \quad \pi_{i+1} \quad \pi_i \ldots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \ldots \pi_{d-1}]. \end{aligned}$$

And so, $(\Pi_1)^{(i)} = \Pi_1^{(i)} = \Pi_1 \circ \Gamma_{\text{SWP},i}$.

Case 3: $(i = \ell^-)$ We consider two cases depending on whether S is a 0-child or a 1-child.

Case 3a: If S is a 0-child then

$$\begin{aligned} S &= [\mathbf{v}_0 \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T \\ S^{(d)} &= [\mathbf{v}_0 \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{y} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T \\ S_1 &= [\mathbf{v}_0 \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T \\ S_0^{(d)} &= [\mathbf{v}_0 \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m}' \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T. \end{aligned}$$

Then, $(S_1)^{(\ell^-)} = S_0^{(d)}$.

$$\begin{aligned} \Pi &= [\pi_1 \ldots \pi_d], \\ \Pi_1 &= [\pi_1 \ldots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \ldots \pi_{d-1}], \\ \Pi_0^{(d)} &= \Pi^{(d)} = [\pi_1 \ldots \pi_{\ell^- - 1} \quad -\pi_d \quad \pi_{\ell^-} \ldots \pi_{d-1}]. \end{aligned}$$

And so, $(\Pi_1)^{(\ell^-)} = \Pi_0^{(d)} = \Pi_1 \circ \Gamma_{\text{SWP},\ell^-}$.

Case 3b: If S is a 1-child then

$$\begin{aligned} S &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T \\ S^{(d)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1} \quad \mathbf{u}]^T \\ S_1 &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \\ S_1^{(d)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m}' \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T. \end{aligned}$$

Then, $(S_1)^{(\ell^-)} = S_1^{(d)}$.

$$\begin{aligned} &\Pi = [\pi_1 \dots \pi_d], \\ &\Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}], \\ &\Pi^{(d)} = [\pi_1 \dots \pi_{\ell^- - 1} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1} \quad -\pi_{\ell^-}], \\ &\Pi_1^{(d)} = [\pi_1 \dots \pi_{\ell^- - 1} \quad -\pi_d \quad \pi_{\ell^-} \dots \pi_{d-1}]. \end{aligned}$$

And so, $(\Pi_1)^{(\ell^-)} = \Pi_1^{(d)} = \Pi_1 \circ \Gamma_{\text{SWP},\ell^-}$.

Case 4: ($i = \ell^*$) By definition, the (ℓ^*)-th neighbor of S_1 is its sibling, S_0, and $(\Pi_1)^{(\ell^*)} = \Pi_0 = \Pi_1 \circ \Gamma_{\text{LFT},\ell^-}$.

Case 5: ($\ell^* < i \leq d, \ell \neq 0$) We consider two subcases, depending on i.

Case 5a: ($\ell^* < i < d$, $\ell \neq 0$)

$$\begin{aligned} S &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-2} \quad \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T, \\ S^{(i-1)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-2} \quad \mathbf{u} \quad \mathbf{v}_i \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T, \\ S_1 &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-2} \quad \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_{d-1}]^T, \\ S_1^{(i-1)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-2} \quad \mathbf{u} \quad \mathbf{v}_i \dots \mathbf{v}_{d-1}]^T. \end{aligned}$$

Then, $(S_1)^{(i)} = S_1^{(i-1)}$.

$$\begin{aligned} &\Pi = [\pi_1 \dots \pi_{\ell^-} \dots \pi_{i-1} \quad \pi_i \dots \pi_d], \\ &\Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{i-1} \quad \pi_i \dots \pi_{d-1}], \text{since} \quad i - 1 > \ell^- \\ &\Pi^{(i-1)} = [\pi_1 \dots \pi_{i-2} \quad \pi_i \quad \pi_{i-1} \quad \pi_{i+1} \dots \pi_d], \\ &\Pi_1^{(i-1)} = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{i-2} \quad \pi_i \quad \pi_{i-1} \quad \pi_{i+1} \dots \pi_{d-1}]. \end{aligned}$$

And so, $(\Pi_1)^{(i)} = \Pi_1^{(i-1)} = \Pi_1 \circ \Gamma_{\text{SWP},i}$.

Case 5b: ($i = d$, $\ell \neq 0$)

$$\begin{aligned} S &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-2} \quad \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T, \\ S^{(d-1)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-2} \quad \mathbf{u} \quad \mathbf{v}_d]^T, \\ S_1 &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-2} \quad \mathbf{v}_{d-1}]^T, \\ S_1^{(d-1)} &= [\mathbf{v}_0 \dots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-2} \quad \mathbf{u}]^T. \end{aligned}$$

Then, $(S_1)^{(d)} = S_1^{(d-1)}$.

$$\begin{aligned} &\Pi = [\pi_1 \dots \pi_{\ell^-} \dots \pi_{d-1} \quad \pi_d], \\ &\Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}], \quad \text{since} \quad d - 1 > \ell^- \\ &\Pi^{(d-1)} = [\pi_1 \dots \pi_{d-2} \quad \pi_d \quad \pi_{d-1}], \\ &\Pi_1^{(d-1)} = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_{d-1} \quad \pi_{\ell^*} \dots \pi_{d-2} \quad \pi_d]. \end{aligned}$$

And so, $(\Pi_1)^{(d)} = \Pi_1^{(d-1)} = \Pi_1 \circ \Gamma_{\text{NSW},\ell^*} = \Pi_1 \circ \Gamma_{\text{NSW},\ell}$.

7. Concluding Remarks

We have presented a representation of hierarchical regular simplicial meshes based on Maubach's[7] simplex bisection algorithm. Unlike Maubach's approach, which requires the use of recursion or an explicit tree structure, our representation is pointerless, that is, the simplices of the mesh are uniquely identified through a location code, called the LPT code. We have shown how to use this code to traverse the hierarchy, compute neighbors, and to answer point location and interpolation queries.

The space savings realized by not having to store pointers (to the two children, the parent, and $d+1$ neighbor simplices) is significant for large multidimensional meshes. If desired, the vertices of a simplex need not be stored either, and can be computed entirely from the code of the simplex. For example, for a 4-dimensional SD-tree consisting of 13.2 million cells, the storage requirements when storing pointers and vertices is 708MB, whereas it is 354MB without pointers, and 222MB without pointers and vertices (in fact pointers to vertices) within the cell. (Note that these numbers also include application specific data associated with vertices.)

Processing of LPT codes is quite efficient. Given a tree of maximum depth D in dimension d, we showed that, under the reasonable assumption that the machine's word length is $\Omega((D/d)+\log_2 d!)$, it is possible to pack the LPT code into words so that all traversal and neighbor-finding operations can be performed in $O(d)$ time through the use of standard integer arithmetic and bit masking and shifting. In fact, by precomputing multiplication tables for the small number of possible operations defined on codes these operations can be performed in $O(1)$ time. (Computing the orthant list component of the code for children or parent has worst-case $O(d)$ time complexity, however the amortized cost is $O(1)$, since orthant list is updated only at every d levels.) In addition, point location can be performed with $O(\log D)$ global memory accesses with the pointerless representation, in contrast with $O(D)$ global memory accesses with the pointer-based one.

Acknowledgements

We would like to acknowledge valuable discussions with David Kirkpatrick and Leila De Floriani on the subject of compatible hierarchical triangulations. We also thank anonymous reviewers for their suggestions which helped improve this paper.

References

1. R. E. Bank, A. H. Sherman and A. Weiser, Refinement algorithms and data structures for regular local mesh refinement, *J. Sci. Comput.* (1983) 3–17.
2. W. F. Mitchell, Optimal multilevel iterative methods for adaptive grids, *SIAM J. Sci. Stat. Comp.* **13** (1992) 146–167.
3. D. N. Arnold, A. Mukherjee and L. Pouly, Locally adapted tetrahedral meshes using bisection, *SIAM J. Sci. Comput.* **22**(2), (2001) 431–448.
4. J. Bey, Tetrahedral grid refinement, *Comput.* **55** (1995) 355–378.

5. A. Liu and B. Joe, Quality local refinement of tetrahedral meshes based on bisection, *SIAM J. Sci. Comput.* **16** (1995) 1269–1291.
6. A. Liu and B. Joe, Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision, *Math. Comput.* **65**(215) (1996) 1183–1200.
7. J. M. Maubach, Local bisection refinement for N-simplicial grids generated by reflection, *SIAM J. Sci. Stat. Comp.* **16** (1995) 210–227.
8. M. C. Rivara, Local modification of meshes for adaptive and/or multigrid finite-element methods, *J. Comput. Appl. Math.* **36** (1991) 79–89.
9. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, EATCS Monographs on Theoretical Computer Science, Vol. 10 (Springer-Verlag, 1987).
10. J. R. Munkres, *Topology: A First Course* (Prentice Hall, Englewood Cliffs, NJ, 1975).
11. Y. Zhou, B. Chen and A. Kaufman, Multiresolution tetrahedral framework for visualizing regular volume data, *Proc. IEEE Visualization'97*, 1997, pp. 135–142.
12. T. Gerstner and M. Rumpf, Multiresolutional parallel isosurface extraction based on tetrahedral bisection, *Proc. Symp. Volume Visualization*, 1999.
13. M. Ohlberger and M. Rumpf, Hierarchical and adaptive visualization on nested grids, *Computing* **56** (1997) 365–385.
14. D. J. Hebert, Symbolic local refinement of tetrahedral grids, *J. Symb. Comput.* **17** (1994) 457–472.
15. A. S. Glassner, *An Introduction to Ray Tracing* (Academic Press, San Diego, 1989).
16. F. B. Atalay and D. M. Mount, Ray interpolants for fast ray-tracing reflections and refractions, *J. WSCG* **10**(3) (2002) 1–8, *Proc. Int. Conf. Central Europe on Computer Graphics, Visualization and Computer Vision.*
17. F. B. Atalay and D. M. Mount, Interpolation over light fields with applications in computer graphics, *Proc. 5th Workshop on Algorithm Engineering and Experiments (ALENEX 2003)* (SIAM, 2003), pp. 56–68.
18. M. Bern, D. Eppstein and J. Gilbert, Provably good mesh generation, *J. Comput. Syst. Sci.* **48** (1994) 384–409.
19. R. Pajarola, Overview of quadtree-based terrain triangulation and visualization, Technical report, UCI-ICS-02-01, Information & Computer Science, University of California Irvine, 2002.
20. R. Sivan and H. Samet, Algorithms for constructing quadtree surface maps, *Proc. 5th Int. Symp. Spatial Data Handling*, 1992, pp. 361–370
21. H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS* (Addision-Wesley, 1990).
22. M. Lee and H. Samet, Navigating through triangle meshes implemented as linear quadtrees, *ACM Trans. Comput. Graph.* **19** (2000) 79–121.
23. T. Chilimbi, M. D. Hill and J. R. Larus, Cache-conscious structure layout, in *Programming Languages Design and Implementation*, 1999.
24. A. Aggarwal and J. S. Vitter, The input/output complexity of sorting and related problems, *Commun. ACM* **31** (1988) 1116–1127.
25. L. Arge, External memory data structures, in *Handbook of Massive Data Sets*, eds. J. Abello, P. M. Pardalos and M. G. C. Resende (Kluwer Academic Publishers, 2002), pp. 313–358.
26. E. D. Demaine, Cache-oblivious algorithms and data structures, in *Lecture Notes from the EEF Summer School on Massive Data Sets*, 2002.
27. I. Gargantini, An effective way to represent quad-trees, *Commun. ACM* **25**(12) (1982) 905–910.
28. W. Evans, D. Kirkpatrick and G. Townsend, Right-triangulated irregular networks, *Algorithmica* **30**(2) (2001) 264–286.

29. M. Lee, L. De Floriani and H. Samet, Constant-time neighbor finding in hierarchical tetrahedral meshes, *Proc. Int. Conf. Shape Modelling*, 2001, pp. 286–295.
30. D. K. Blandford, G. E. Blelloch, D. E. Cardoze and C. Kadow, Compact representations of simplicial meshes in two and three dimensions, *Proc. 12th Int. Meshing Roundable*, 2003.
31. A. Szymczaka and J. Rossignac, Grow & fold: compressing the connectivity of tetrahedral meshes, *Comput. Aided Desig.* **32** (2000) 527–537.
32. J. M. Maubach, The efficient location of neighbors for locally refined n-simplicial grids, *Proc. 5th Int. Meshing Roundable*, 1996.
33. E. Allgower and K. Georg, Generation of triangulations by reflection, *Utilitas Mathematica* **16** (1979) 123–129.
34. D. E. Knuth, *Sorting and Searching*, The Art of Computer Programming, Vol. 3 (1973).
35. S. R. Lay, *Convex Sets and Their Applications* (John Wiley & Sons, New York, NY, 1982).
36. F. B. Atalay and D. M. Mount, Pointerless implementation of hierarchical simplicial meshes and efficient neighbor finding in arbitrary dimensions, *Proc. 13th Int. Meshing Roundable*, 2004, pp. 15–26.