ABSTRACT

Title of Dissertation: SPATIAL DECOMPOSITIONS FOR GEOMETRIC INTERPOLATION AND EFFICIENT RENDERING

Fatma Betul Atalay-Satoglu, Doctor of Philosophy, 2004

Dissertation directed by: Professor David M. Mount
Department of Computer Science

Interpolation is fundamental in many applications that are based on multidimensional scalar or vector fields. In such applications, it is possible to sample points from the field, for example, through the numerical solution of some mathematical model. Because point sampling may be computationally intensive, it is desirable to store samples in a data structure and estimate the values of the field at intermediate points through interpolation. We present methods based on building dynamic spatial data structures in which the samples are computed on-demand, and adaptive strategies are used to avoid oversampling.

We first show how to apply this approach to accelerate realistic rendering through ray-tracing. Ray-tracing can be formulated as a sampling and reconstruction problem, where rays in 3-space are modeled as points in a 4-dimensional parameter space. Sample rays are associated with various geometric attributes, which are then used in rendering. We collect and store a relatively sparse set of sampled rays, and use inexpensive interpolation methods to approximate the attribute values for other rays. We present two data structures: (1) the *ray interpolant tree (RI-tree)*, which is based on a kd-tree-like sub-

division of space, and (2) the *simplex decomposition tree (SD-tree)*, which is based on a hierarchical regular simplicial mesh, and improves the functionality of the RI-tree by guaranteeing continuity.

For compact storage as well as efficient neighbor computation in the mesh, we present a pointerless representation of the SD-tree. An essential element of this approach is the development of a location code that enables efficient access and navigation of the data structure. For this purpose we introduce a location code, called an LPT code, that uniquely encodes the geometry of each simplex of the hierarchy. We present rules to compute the neighbors of a given simplex efficiently through the use of this code. We show how to traverse the associated tree and how to answer point location and interpolation queries. Our algorithms work in arbitrary dimensions. We also demonstrate the use of the SD-tree for rendering atmospheric effects. We present empirical evidence that our methods can produce renderings of good quality significantly faster than simple ray-tracing.

SPATIAL DECOMPOSITIONS FOR GEOMETRIC INTERPOLATION AND EFFICIENT RENDERING

by

Fatma Betul Atalay-Satoglu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

Advisory Committee:

Professor David M. Mount, Chair/Advisor
Professor William M. Goldman
Professor Leila De Floriani
Professor Amitabh Varshney
Professor Samir Khuller

DEDICATION

To my parents,

Yıldız and Beşir Atalay

To my sisters,

Tuba and İrem Atalay

And to my dear aunt,

Emine Fidan

ACKNOWLEDGEMENTS

Firstly, I would like to extend my deepest gratitude to my advisor, Dr. David Mount, the best advisor and teacher I could have wished for. His incredible support and guidance, and above all, his endless patience and kindness made this thesis possible. The time and care he puts into his students sets an example I hope to follow. I am grateful to Dr. Amitabh Varshney for his encouragement and helpful comments on this research. Special thanks to Dr. Leila De Floriani for her invaluable help, excellent geometric modeling class and especially for traveling such a long way to serve in my thesis committee. My sincere thanks also goes to the other members of my thesis committee, Dr. Samir Khuller and Dr. William Goldman.

Throughout this long long process, my life was richly blessed with many good friends without whom I cannot imagine my Maryland days. I am forever indebted to my dear friend Tikir who has always been there for me, seeing me through some of the hardest years of my life, listening to my complaints and frustrations for hours and always knowing the best thing to say to give comfort. I offer my heartfelt thanks to my dear friend and roommate Chiraz, who has been a source of emotional and practical support. Thank you for constantly reminding me that this is a matter of persistence and hard-work, for being such a cheerful person who brightened my days, and of course for providing the best coffee on campus. I am grateful to my dear friend Okan for his kindness and eagerness to help and for the countless times he came to my rescue throughout the grad

school till the very last day of submitting this thesis. Special thanks to my two dear friends with whom I have had some of the most fun times of my life here: to Sule, for sharing my loneliness and for teaching me so much with her faith, intellect and humor. I will surely miss our long lunch breaks and conversations; to Esin for her warmth and selflessness, for her moral support when most needed and for never saying “no”.

I would like to extend my sincere appreciation for many people who have helped me survive with their help and friendship starting from the earliest years in Maryland: Sahin Family, Berrin and Cengiz Celik, Cuneyt Akinlar, Ibrahim Korpeoglu, Yuce Family, Selda Kapan, Gamze Tunali, Ugur Cetintemel, Esin and Ismail Haritaoglu, Hatice Burakgazi and Erhan Yilmaz, Suheyla Aytac, Kevser and Gokturk Ozer and Gulsum Ozturk. Thank you Dilek Akar, Marat Fayzullin, Ahmed Elgammal, Dmitry Zotkin, Khaled Arisha, Burcu and Fazil Ayan, Fusun Yaman, Evren Sirin, Laura Bright, Mounya Elhilali, Cagdas Dirik, Akin Akturk, Rajiv Gandhi, Tamer Sharnouby, and Funda Ertunc for your friendship.

Greatest of thanks goes to my family: To my mother who has always been and will always be my ultimate role model for her character and determination; and for her excellent balance of a demanding career and perfect motherhood of three (fine) children. She stood by me at every single step of my life, and her love and inspiration has given me all the strength I have ever needed since as far as I can remember. I would not be here if it were not for her. To my father for teaching me the value of a life with dignity, and setting an excellent example of that, for his wisdom which keeps enlightening my path and for instilling in me the love of books and knowledge in my very early childhood. To my beloved sisters, Tuba and Irem, who are the most precious gifts God has given

me. They are the reason I maintained my sanity all these years. Through our long phone conversations—including the ones that woke them up in the middle of the night—they have endured all the fears and tears and raised my spirits all along. To my dearest aunt, who has been a second mother to me, thank you for your love and sacrifice, and your prayers for my success. To my in-laws for their prayers and for sharing my happiness and pride on completion of this dissertation.

And finally, but definitely not least, a very special thank you to my dear husband Mirat, my best friend and companion, whose enormous support, patience and optimism kept me going through the past years and made the completion of this thesis possible. Thank you for sharing my times of despair as well as times of joy and thank you for believing in me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Spatial data structures can play a vital role in achieving efficient computation for geometric applications. In this thesis we consider how spatial data structures can be used to improve the running time of algorithms used in the field of computer graphics for producing photo-realistic images. Computer graphics is concerned with all aspects of the process of creating images from 3-dimensional models, which is often called *rendering*. Given a scene modeled as a collection of objects and light sources, and viewing specifications for the camera, rendering algorithms generate images by simulating the propagation of light in the scene. Light rays originate from light sources and go through several interactions with the scene objects being reflected, transmitted, or absorbed until finally leaving the scene or reaching the camera.

Different graphics applications demand different levels of realism. At one end of the spectrum is *photo-realism* which aims to capture physically accurate, complex illumination effects such as reflections, specular highlights and shadows. Simulating these effects, even approximately, is a computationally demanding task. Many graphics systems

simplify the effects of light for fast rendering at the expense of realism. Fast hardware renderers use *local illumination models*, in which the color of the surface point is computed as a function of only the direct light coming from the light sources ignoring the inter-reflections from surface to surface. In contrast, *global illumination models* provide a more accurate approximation to reality by incorporating both direct lighting from light sources and indirect lighting from other scene objects.

This thesis is mainly motivated by the ray-tracing method which has long been the most popular global illumination algorithm. Ray-tracing can accurately capture *view-dependent* phenomena such as specular highlights, reflections and refractions. However, it remains a computationally very expensive technique. In traditional ray-tracing solutions, at least one ray is shot through each pixel of the image plane, and the intensity gathered by tracing the ray through the scene constitutes the color of that pixel. A critical part of any ray-tracer is the ability to determine the intersection of rays with objects of the model and how these rays may reflect off of and refract through the objects. This involves many expensive ray-object intersection computations, especially for scenes containing complex objects such as Bézier surfaces or NURBS.

In computer graphics, it is common to infer knowledge from samples. A ray-tracer, for example, is basically sampling rays at the pixel level (or at the sub-pixel level), where each sample demands high computational effort. Pixel level may not always be the most appropriate level at which to sample. The color of any given pixel in an image is a combination of many different phenomena, including the base color of the object, the intensity of the accumulated light at this point, the nature of the reflection function of the object's surface, the presence of reflection or transparency, and the scattering and obscuration

due to atmospheric effects such as smoke or clouds. Some of these elements may vary relatively slowly and smoothly over large areas of an image. These elements can be reproduced realistically with relatively sparse samples, since rays have to be sampled densely only in the regions where rapid changes occur with respect to the function that is of interest. This suggests adaptively collecting a set of sparse samples, and using inexpensive interpolation methods to approximate others.

To determine the appropriate level that we should be interpolating, we need data structures. These data structures should support efficient storage and efficient querying and interpolation of samples. The main problem we consider in this thesis is how to design efficient data structures for answering multidimensional interpolation queries. We show how to apply this approach for efficient ray-tracing by formulating ray-tracing as a sampling and reconstruction problem based on 4-dimensional fields of directed lines, where each directed line is associated with a set of vector-valued geometric attributes. We focus on accelerating the geometric component of ray-tracing by substituting accurate-but-slow intersection computations by approximate-but-fast interpolations.

Interpolation involves a weighted average of the field values of nearby sampled points. There are a number of approaches for determining which sample points to use and how to assign weights [Alf89, Sib81]. For the sake of efficiency in answering queries, we use a simple method. A spatial subdivision is constructed over the domain of interest, and the field values are sampled at the vertices of this subdivision. For a given query point, the interpolated value is an appropriate linear or multi-linear combination of the field values at the vertices of the cell that contains it. To avoid the complex computational issues associated with maintaining, accessing, and updating arbitrary multidimensional spatial

subdivisions, we ensure that the subdivision possesses some regular structure. This is possible since we have the flexibility of controlling the location of our samples, unlike some applications that are given a fixed set of samples in advance.

The requirement of adaptive sampling suggests that one way to approach answering interpolation queries is through data structures based on hierarchical subdivision of space, such as quadtrees and kd-trees [Sam90b]. We introduce the *RI-tree* data structure, which is a spatial decomposition based on a kd-tree-like subdivision of 4-dimensional space of directed lines. In higher dimensions, the storage requirements for representing a complete interpolation function by sampling the entire space is extremely high. To overcome this problem, we build a dynamic data structure in which samples are computed on demand, and only the most recently used samples are stored. We investigate tradeoffs between space and time used by this data structure and the accuracy of the interpolation results.

Another important concern when building data structures for interpolation purposes is the continuity of the interpolated surface. A significant problem with kd-tree-like subdivisions is that they do not guarantee any degree of continuity. *Cracks* may exist on the interpolating surface whenever cells fail to intersect along a *single* common face, hence not allowing even the lowest level of functional continuity. We address this issue by introducing the *SD-tree* data structure, which is based on a regular *simplicial* subdivision that is refined in a particular way to provide continuity. Simplicial decompositions are preferable also for other reasons that will be discussed further in the thesis. However, for them to be considered feasible alternatives, it is crucial that basic operations such as subdivision and point location can be performed rapidly since these operations may not be as trivial as they are for kd-trees. We address algorithmic issues involved in efficient imple-

mentation of hierarchical simplicial meshes in arbitrary dimensions, based on a bisection approach proposed by Maubach [Mau95].

In order to avoid cracks in the subdivision, the refinement algorithm must ensure that bisection of a simplex triggers the bisection of its facet neighbors. (See the 2-dimensional example in Figure 1.1.) This makes computation of facet neighbors of a simplex an essential operation, imposing the requirement for efficient neighbor finding algorithms. For space concerns, it is not preferable to store neighbor pointers ($d+1$ neighbors for each node of a d-dimensional subdivision), especially for large high-dimensional meshes. On the other hand, it is highly desirable to compute neighbors of a simplex in time independent of its depth in the hierarchy, that is, we do not want to traverse the path to and from the root in order to compute neighbors. Instead, we present a pointerless representation of hierarchical regular simplicial meshes, in which the nodes of the hierarchy are accessed through an index called a *location code*. We introduce a new location code, called an LPT code, that uniquely encodes each simplex of the hierarchy. We present rules to compute any neighbor of a given simplex directly from its code in constant-time.

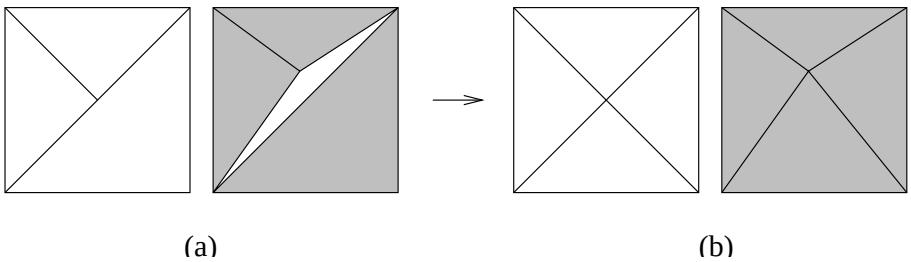


Figure 1.1: A simplicial mesh in the plane and the corresponding interpolating surface. (a) Crack in the interpolating surface due to bisection of one of the triangles. (b) Bisection of the neighbor triangle eliminates the crack.

We prove correctness of these rules. We show how to traverse the associated tree and how to answer point location and interpolation queries through the use of these codes. Our system works in arbitrary dimensions.

Finally, we discuss a number of issues in efficient use of the SD-tree data structure in another expensive rendering application, that is, rendering of atmospheric effects such as smoke. We propose a variation in the construction of the SD-tree to reconcile between the continuity requirement and the on-demand sampling requirement which normally conflict. This is a heuristic approach aiming to reduce the size of the data structure as well as to increase the performance.

The rest of this dissertation is organized as follows. Chapter 2 presents a survey of spatial data structures for multidimensional data, emphasizing simplicial refinement methods, and previously proposed pointerless representations of hierarchical simplicial meshes. Chapter 3 contains related work on accelerating ray-tracing and animations, as well as on image-based rendering methods, which are relevant to this thesis in the way they use interpolation. In Chapter 4, we introduce the *RI-tree* data structure and present empirical results to demonstrate its application in efficient ray-tracing. In Chapter 5, we introduce a pointerless representation for hierarchical regular simplicial meshes, called the *SD-tree*, in a theoretical setting which is applicable in arbitrary dimensions. We present theorems that are necessary to develop our labeling scheme, and to prove correctness of our neighbor finding rules. In Chapter 6, we illustrate the practical value of the *SD-tree* in rendering smoke based on experimental evidence. Finally, in Chapter 7, we conclude by summarizing our contributions, and proposing future research directions.

Chapter 2

Hierarchical Data Structures for Multidimensional Data

The first large piece of literature relevant to this thesis is in the area of hierarchical data structures developed for efficient storage and retrieval of multidimensional data. Representation of spatial data has become important in many application areas such as computer graphics, visualization, image processing and geographic information systems, where efficient algorithms for manipulating data is crucial in the performance of the application. Many of the general purpose data structures that are commonly used are hierarchical, based on the principle of recursive decomposition of space. The most widely used ones of these data structures are quadtrees (octrees), kd-trees, and BSP-trees which differ in the way they choose the cutting plane(s), and in the branching factor (the number of sub-regions each region is decomposed into).

For *quadtrees*, even though a number of variations exist [FB74, Web84, Sam90b], the most common representation is based on recursively decomposing the square domain into four equal-sized squares. An octree is the 3-dimensional extension to the quadtree, subdividing the cubic domain into 8 equal-volume sub-cubes. Hence, in quadtrees and

their higher dimensional extensions, the split is performed in all dimensions. The *multi-dimensional binary search tree*, or *kd-tree* as it is more commonly known as [Ben75], are slower growing subdivisions compared to quadtrees, since they split one dimension at a time. Different methods have been proposed on how to choose the split dimension such as alternating dimensions or choosing the dimension corresponding to the longest side of the hyperrectangle being subdivided. A hyperrectangle is split into two sub-hyperrectangles by a cutting plane which is always perpendicular to the coordinate axis corresponding to the split dimension, but the positioning of the cut plane could differ depending on the application. In these respects, kd-trees are more flexible than quadtrees. The *binary space partition tree*, or *BSP-tree* [FKN80] for short, are also organized as binary search trees, however the cutting planes of arbitrary orientation are allowed as opposed to kd-trees. Hence, each node is an arbitrarily shaped convex polytope. For a good survey of these general data structures and various others for spatial data as well as their applications, we refer the reader to the well known reference books by Samet [Sam90a, Sam90b].

Our particular interest, however within the context of this thesis, is in simplicial decompositions. We will also refer to simplicial decompositions as *triangulations* regardless of the dimension—even though the term *triangulation* is generally only used for a 2-dimensional simplicial decomposition. In this chapter, we first survey a number of adaptive refinement methods for generating simplicial decompositions. Next, we review pointerless data structures developed for space-efficient representation of these simplicial meshes, and efficient methods to perform operations like traversal and neighbor finding based on the pointerless representation.

2.1 Simplicial Mesh Refinement

There has been a considerable amount of work in adaptive simplicial mesh refinement, mostly due to its importance in finite element methods for numerical solution of partial differential equations. The mesh serves as a discretization of the domain of a function, and by means of adaptive local refinements, it is used to improve the approximate solution of a partial differential equation locally. In addition, adaptively refined meshes have been widely used in various application areas in computer graphics, scientific visualization and geometric modeling. For example, 2-dimensional meshes are used for multiresolution terrain modeling and rendering [LKR+96, DWS+97, Paj98, Ger03]; 3-dimensional meshes for volume rendering of 3-dimensional scalar fields (such as medical datasets) [GR99, ZCK97], and 4-dimensional meshes for visualization of time-varying flow fields. Higher dimensional meshes are used in combinatorial algorithms to determine fixed points of functions as described in [Tod76], and to approximate solution manifolds of parameterized equations [Mau95].

Hierarchical simplicial meshes are obtained by starting with a coarse partition (triangulation) $\mathcal{T}_0$ of the domain, and generating a sequence $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_j$ of increasingly finer partitions by successive local mesh refinement, which is typically driven by a local error estimator. Each refinement step involves selecting certain simplices for refinement based on their local error estimate and refining those simplices and possibly some other simplices to preserve *compatibility*. It is usually desired that the sequence of partitions satisfy the conditions of *compatibility* and *stability*.

A simplicial mesh is compatible if the intersection of two neighboring simplices is a

single common sub-simplex (face). A compatible simplicial mesh is also called a *simplicial complex*. The compatibility condition is important since otherwise cracks may occur along the faces of the subdivision, which in turn causes discontinuities in the function.

The stability condition requires that the simplices generated during refinement do not degenerate. This usually means that, a certain shape measure ρ is bounded by a constant c for all simplices in the hierarchy, where the constant c is independent of the level of the simplex. A commonly used shape measure for a k-simplex is the aspect ratio, which is defined as the ratio of the length of the longest edge to the diameter of the largest inscribed k-ball. Liu and Joe [LJ94b] studied other shape measures and their relationship for tetrahedral meshes. Stability is important, since in many applications poorly-shaped simplices must be avoided. In scientific visualization, for example, thin, elongated tetrahedra will result in visual artifacts. In finite element methods, stability is not only desired, but is often essential to guarantee well-conditioned systems and numerical convergence.

Another desirable property for hierarchical simplicial meshes is that the number of congruency classes generated is finite, and in fact minimal. Two simplices are defined to be *congruent*, if they are equivalent up to a rigid motion (translation, rotation and reflection), and a nonzero uniform scaling.

The key element in an adaptive refinement algorithm is the basic subdivision step, which defines how a single simplex is subdivided into smaller simplices. Our emphasis here is on *regular* hierarchical simplicial meshes. We refer to a mesh as *regular*, if the process by which a simplex is subdivided is identical for all simplices. (A more restricted definition of regularity means that the vertices of a mesh are regularly distributed, such as the vertices of a grid). Regular hierarchical meshes satisfy the *nestedness* condition,

which states that each element $T \in \mathcal{T}_k$ is covered by exactly one element $T' \in \mathcal{T}_{k-1}$. We will also limit our survey to meshes where any vertex of T is either a vertex or an edge midpoint of T', i.e. new vertices are generated only at edge midpoints.

Various different refinement techniques have been proposed, particularly in 2- and 3-dimensions. Depending on the subdivision scheme, these techniques can be classified in two major groups: *red-green refinement* methods, which are based on subdividing a d-simplex into 2^d descendants and *bisection* methods, which subdivide a simplex into two descendants.

Red-Green Refinement Methods: Methods of this class typically consist of *regular* and *irregular* local refinement rules, which are combined in a global refinement algorithm to provide compatibility and stability. The *regular* (*red*) refinement rules subdivide the simplices with respect to a local error estimate and in a certain regular way. The *irregular* (*green*) rules are needed in case of adaptive refinements and they are only applied in order to guarantee compatibility by providing transition between different refinement levels. Irregular refinement is often referred to as the *green closure* or the *conforming closure* and is performed *after* the regular refinements. Only regular refinements introduce new vertices.

The first red-green refinement method was introduced by Bank in 2-dimensions [BSW83], and later was implemented into the multigrid code PLTMG [Ban98]. A triangle is subdivided into four congruent smaller triangles by connecting its edge midpoints as shown in Figure 2.1(b). Without adaptive refinements, since the generated triangles belong to a single congruency class, the triangulation will always be stable independent

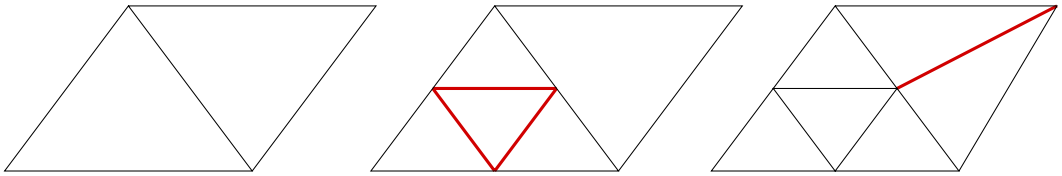


Figure 2.1: Red-green refinement in 2-dimensions (a) initial state (b) after red refinement (c) after green refinement.

of the depth of refinement. In case of adaptive refinements, compatibility is preserved by the following irregular refinement rules. If a single edge of a triangle T, is refined due to refinement of a neighboring triangle, then T is refined by connecting the midpoint of that edge to the opposite vertex as shown in Figure 2.1(c). Thus, T is bisected generating two descendants, but without introducing any additional vertices. If two or more edges of T are refined due to refinement of a neighboring triangle, then T is regularly refined into four descendants as explained above.

Triangles generated by irregular refinement are referred to as *irregular elements*. To avoid generating unstable meshes, irregular elements are never further refined. If they need to be refined due to high local error, then they are removed and their parents are regularly refined instead.

Bey [Bey95], Zhang [Zha95], and Liu and Joe [LJ96] generalized Bank's method to 3-dimensions. A tetrahedron is subdivided into eight smaller tetrahedra of equal volume. Consequently, Liu and Joe [LJ96] refer to this method as *8-subtetrahedron subdivision*. By connecting the edge midpoints of each triangular face as in the 2-dimensional case, we get four sub-tetrahedra at the corners and an octahedron in the middle as shown in Fig-

ure 2.2 [LJ96]. This octahedron is further subdivided into four sub-tetrahedra by adding one of its three possible diagonals. Note that the corner tetrahedra are all congruent to their parent, but the middle ones are in general not. Regarding which of the three diagonals to add when subdividing the octahedron, different strategies have been investigated. Different choices may lead to substantially different meshes with respect to the quality of the tetrahedra generated. Zhang [Zha95] proposed to always choose the shortest diagonal and showed that the measure of degeneracy is minimized in that case. Bey [Bey95], on the other hand, selects the diagonal based on a certain vertex ordering of the vertices of the tetrahedra, and proves that his method generates stable and compatible triangulations with at most three congruency classes, no matter how many refinement steps are performed.

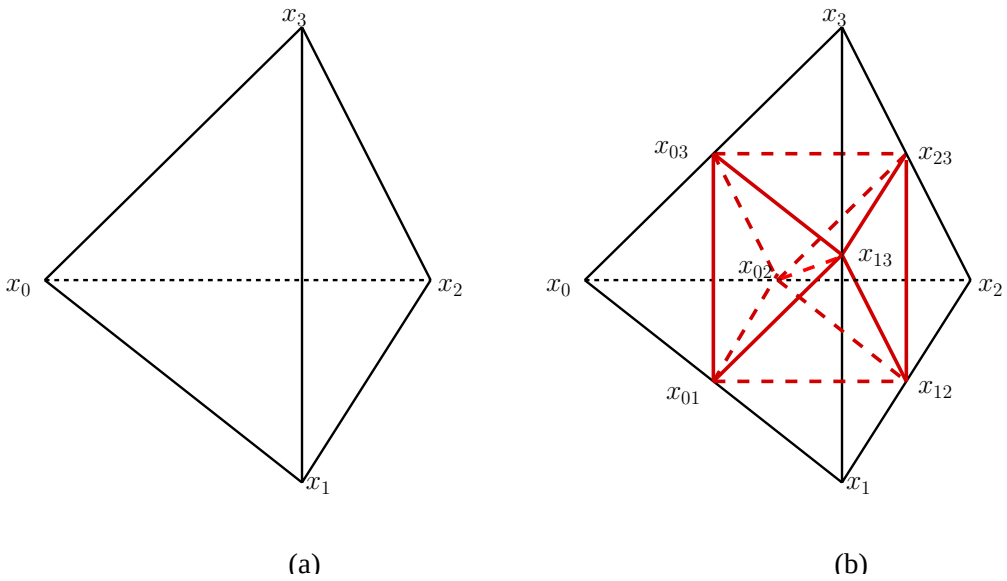


Figure 2.2: Red refinement in 3-dimensions (a) initial state (b) after red refinement.

In Bey's method [Bey95], the subdivision is summarized as follows: Assume that a tetrahedron T is given by an ordered sequence of its vertices, $\langle \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3 \rangle$. Let $\mathbf{x}_{ij}$

denote the midpoint of $\mathbf{x}_i$ and $\mathbf{x}_j$, $i \neq j$. Then, subdivision of T generates the eight sub-tetrahedra $T_i, 1 \leq i \leq 8$ with the following sequence of ordered vertices:

$$T_1 = \langle \mathbf{x}_0, \mathbf{x}_{01}, \mathbf{x}_{02}, \mathbf{x}_{03} \rangle , \qquad T_2 = \langle \mathbf{x}_{01}, \mathbf{x}_1, \mathbf{x}_{12}, \mathbf{x}_{13} \rangle ,$$

$$T_3 = \langle \mathbf{x}_{02}, \mathbf{x}_{12}, \mathbf{x}_2, \mathbf{x}_{23} \rangle , \qquad T_4 = \langle \mathbf{x}_{03}, \mathbf{x}_{13}, \mathbf{x}_{23}, \mathbf{x}_3 \rangle ,$$

$$T_5 = \langle \mathbf{x}_{01}, \mathbf{x}_{02}, \mathbf{x}_{03}, \mathbf{x}_{13} \rangle , \qquad T_6 = \langle \mathbf{x}_{01}, \mathbf{x}_{02}, \mathbf{x}_{12}, \mathbf{x}_{13} \rangle ,$$

$$T_7 = \langle \mathbf{x}_{02}, \mathbf{x}_{03}, \mathbf{x}_{13}, \mathbf{x}_{23} \rangle , \qquad T_8 = \langle \mathbf{x}_{02}, \mathbf{x}_{12}, \mathbf{x}_{13}, \mathbf{x}_{23} \rangle$$

The diagonal chosen to subdivide the octahedron is always the one between $\mathbf{x}_{02}$ and $\mathbf{x}_{13}$, meaning that it is determined by the vertex ordering, and not by means of any computation. Bey explains that Zhang's *shortest-interior* edge strategy is equivalent to this method when it is applied to initial tetrahedralizations with non-obtuse faces and a suitable vertex ordering [Bey00].

Bey also enumerated the irregular refinement rules for 3-dimensional red-green refinement. In 3-dimensions, there are $2^6 = 64$ possible edge refinement patterns. Two of these correspond to the regular refinement and empty refinement. The other 62 correspond to irregular cases, which can be classified into 9 groups. Bey restricts his algorithm to only four of these types that are shown in Figure 2.3 [Bey95]. Type (1) corresponds to three refined edges on the same face, type (2) correspond to exactly one refined edge,

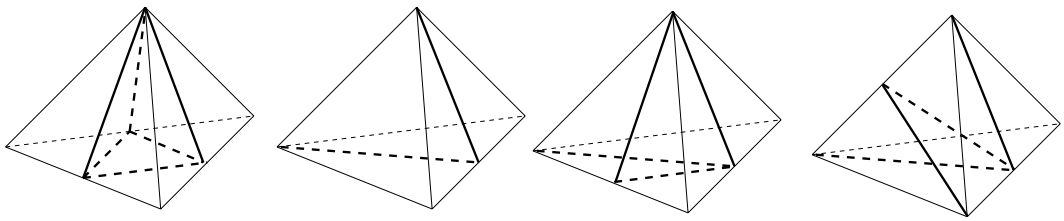


Figure 2.3: Irregular refinement in 3-dimensions.

types (3) and (4) correspond to two refined edges on the same face and on opposite edges, respectively. For all other cases where there are three or more refined edges that belong to different faces, the tetrahedron is refined regularly.

Liu and Joe [LJ96] provided a theoretical analysis of the quality of tetrahedral meshes generated similarly to Bey's method. They have shown that successive application of their refinement method to any initial tetrahedron T produces at most three classes of congruent tetrahedra, and will result in stable partitions such that for any tetrahedron T_i^n in the hierarchy, and the mean ratio η [LJ94a] as the tetrahedron shape measure, the inequality $0.5\eta(T) \leq \eta(T_i^n) \leq 2\eta(T)$ holds.

Bey [Bey00] explains that the 2-and 3-dimensional red-green refinement methods are special cases of Freudenthal's d-dimensional algorithm [Fre42], which was introduced within the context of fixed point computations, and is based on subdividing a simplex into 2^d sub-simplices of equal volume. Bey [Bey00] also proved that Freudenthal's algorithm generates at most $d!/2$ congruency classes for an initial d-simplex.

Bisection Refinement Methods: The second major class of refinement methods are based on dividing a simplex into two sub-simplices by bisection. A number of authors proposed bisection algorithms in 2- and 3-dimensions, as well generalizations in d-dimensions. There are two main approaches that differ in how they choose the edge to be bisected. The 2-dimensional bisection method by Rivara [Riv91] always chooses the longest edge for bisection, and consequently is referred to as the *longest-edge-bisection*. Rivara showed that, in 2-dimensions, the meshes constructed are guaranteed to be stable. This method can be applied to any initial compatible triangulation. Rivara and Levin

[RL92] presented a 3-dimensional extension of this method as well.

The well known *newest-vertex-bisection* of Sewell [Sew72] and Mitchell [Mit92] in 2-dimensions, chooses the edge opposite the newest vertex for refinement. In Sewell's terminology, one of the vertices of the triangle is designated as the *peak*, and the opposite edge as the *base*. To bisect the triangle, the peak is connected with the midpoint of the base. The new vertex created at the midpoint of the base is assigned to be the peak of the child triangles. Sewell showed that only four congruency classes arise from subdivision of a single triangle. Unlike red-green refinement methods which perform closure *after* regular refinement is completed, Mitchell's method maintains compatibility *during* the refinement process by subdividing two triangles simultaneously. This compatible refinement is a recursive process, but it is shown that the depth of the recursion is bounded [Mit88]. This method as well can be applied to any coarse triangulation. It generates stable triangulations, since the number of congruency classes is at most four times the number of triangles in the coarse triangulation. A major advantage of Mitchell's method is that the edge to be bisected can be determined without any computation.

Bänsch [Ban91], Liu and Joe [LJ95], and Arnold et al. [AML01] developed extensions of Mitchell's method to 3-dimensions. Arnold et al. describe a *marked tetrahedron* data structure, which simplifies the selection of the refinement edge and recursive compatible refinement. They also proved that the number of congruency classes is finite. Liu and Joe presented an equivalent method and have shown that the quality of the refined mesh is guaranteed. They also showed that the number of congruency classes is finite, but their bound exceeds that of Arnold, et al. These 3-dimensional algorithms as well, apply to any compatible coarse triangulation.

Maubach's d-dimensional Bisection Algorithm: Maubach [Mau95] extended Mitchell's algorithm to arbitrary dimensions, in the sense that it also makes use of a special ordering of vertices and chooses the bisection edge without computation or global communication. In 2-dimensions, Maubach's method is equivalent to Mitchell's but with a different vertex ordering. Even though it is applicable to any arbitrary compatible triangulation in 2-dimensions, Maubach's method in d-dimensions can satisfy compatibility only for special coarse simplicial meshes. Maubach has shown that the simplices can be properly ordered and the method can be applied to a simplicial grid G generated by reflections in a $k_1 \times \ldots \times k_n$ grid of d-cubes covering $[a_1, b_1] \times \ldots \times [a_n, b_n]$ as described in [Tod76], and in which each d-cube is initially subdivided into $d!$ congruent simplices. In addition, any simplicial grid that results from applying a nonsingular mapping to G is also acceptable as an initial partition, since this type of mapping does not affect the order of the vertices. Maubach presented a mathematically rigorous analysis of the geometric structure of this type of d-dimensional simplicial meshes. Even though his method is restrictive regarding initial meshes, his is one of the most well known refinement algorithms. This is because cubic domains are widely used in many applications such as direct volume rendering and isosurface extraction [ZCK97, GR99, GLE97], and multiresolution terrain modeling [LKR+96, DWS+97, Paj98, Ger03].

In Maubach's system, bisection is applied to each of the d simplices within each d-cube, and is defined by the codeblock BisectSimplex shown in Figure 2.4. Let T be described by its ordered sequence of vertices $\langle \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{d-1}, \mathbf{x}_d \rangle$, and let $\ell(T)$ define the level of T in the hierarchy, and T_0 and T_1 denote the two children of T. The coarse simplices are at level 0.

BisectSimplex(T)

$k \leftarrow d - \ell(T) \bmod d$;

$z \leftarrow \frac{1}{2}(\mathbf{x}_0 + \mathbf{x}_k)$;

$T_0 \leftarrow \langle \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{k-1}, z, \mathbf{x}_{k+1}, \ldots, \mathbf{x}_d \rangle$;

$T_1 \leftarrow \langle \mathbf{x}_1, \ldots, \mathbf{x}_k, z, \mathbf{x}_{k+1}, \ldots, \mathbf{x}_d \rangle$;

$\ell(T_0) \leftarrow \ell(T) + 1$;

$\ell(T_1) \leftarrow \ell(T) + 1$;

Figure 2.4: Procedure *BisectSimplex*.

To avoid incompatibilities, this basic bisection step is incorporated in a recursive compatible refinement algorithm, which triggers the bisection of neighboring simplices that share the bisected edge. Maubach proved a number of important properties of this subdivision.

1. The subdivision pattern repeats itself on a smaller scale at every d levels.

2. The descendants of the same level (*modulo* d) are congruent. Thus, exactly d congruency classes are generated for a grid G as described above.

3. For a compatibly refined mesh, two simplices are said to be *compatibly divisible* if their next subdivision will bisect the same edge. If T' shares the edge of T that will be bisected, then the following holds: Either $\ell(T') = \ell(T)$, or $\ell(T') = \ell(T) - 1$. In the first case T and T' are compatibly divisible, in the second case T is compatibly divisible with one of the children of T'.

4. The recursive compatible refinement algorithm terminates due to item (3).

5. For an arbitrary unstructured mesh consisting of N d-simplices refined by Maubach's method, the number of congruency classes is bounded from above by $2^d N$.

2.2 Pointerless Representations and Neighbor Finding

As mentioned above, regular meshes can be represented with nested models. *Trees* are the most straightforward method for representing meshes that are generated by recursive application of a basic subdivision step, since trees can easily describe the nested structure of these meshes. For example, the meshes based on bisection can be represented by a binary forest regardless of the dimension. Each coarse simplex serves as a separate root node. Each node in the tree corresponds to a simplex s in the subdivision and the children of the node associated with s correspond to the two sub-simplices generated by bisection of s. Another example is a 2-dimensional mesh generated by red refinement, which can be represented with a forest of quaternary trees, similar to quadtrees. (These meshes are also referred to as triangle quadtrees [LS00, DM02].)

To provide compatibility in tree-based representations, it is necessary to be able to compute the neighbors of a simplex in order to guarantee that the neighbors splitting the same edge are split simultaneously. Within the context of mesh extraction from multiresolution representations, other methods such as error saturation have also been proposed to provide compatibility without finding neighbors [ZCK97].)

An alternative representation that has been applied both in in 2- and 3-dimensions within the context of multiresolution mesh representations is based on *Directed Acyclic Graphs (DAG)* [DM02, GDL$^+$02, LKR$^+$96]. Consider for example, the 2-dimensional

case for the meshes generated by bisection. A pair of triangles that must be split simultaneously to provide compatibility are referred to as a *diamond* (or a *cluster*) [DKP03]. Each diamond D is split into four triangles as shown in Figure 2.5 when the edge shared by the two triangles of the diamond is bisected. A *DAG of diamonds* (or a *DAG of vertex dependencies* as in [LKR$^+$96]) is constructed such that the root corresponds to the initial subdivision of the square into 2 triangles, which itself is a diamond (as described in Maubach's method [Mau95]). Each node is a diamond and each arc represents a parent-child relation such that parents of a node N correspond to those diamonds that have to be split before the diamond associated with N (because the splitting of parents creates the triangles of N). In 2-dimensions, each node has exactly two parents and four children except at the boundary cases. A portion of the described DAG is shown in Figure 2.6 [DM02]. In higher dimensions as well, the number of children and parents are bounded, but are more complicated to enumerate.

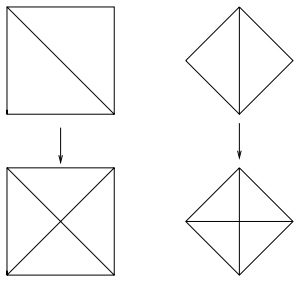


Figure 2.5: Two types of diamonds in 2-dimensional bisection-based mesh.

Explicit representation of trees and DAGs in general involve storing pointers to children, parent(s) and possibly neighbors. Thus, we refer to them as *pointer-based* representations. On the hand, there has been a considerable amount of research on *implicit* or *pointerless* representations for regular meshes, where the geometry (e.g., the vertices)

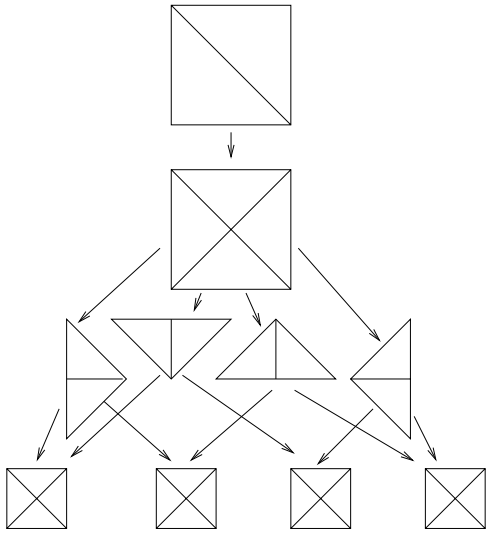


Figure 2.6: DAG representation corresponding to a 2-dimensional bisection-based mesh.

and the relationships (child, parent, neighbor) within the mesh are implicitly encoded, but not stored. This not only leads to more compact data structures in general, but also operations such as neighbor finding can be efficiently performed.

Pointerless versions of the quadtree and its variants have been long known. The most well known of these representations is the *linear quadtree* introduced by Gargantini [Gar82]. In a linear quadtree, each node is identified by a unique label called a *location code*, which consists of two components: the *depth* of the node in the tree, and the *path* from the root to that node. The path is constructed by concatenating the two-bit patterns corresponding to the child types, depending on the direction of the child within the parent quadtree block (00, 01, 10, 11) for each node along the path from the root to the node. Given the location code for a node, not only the codes for the parent and the children, but also the codes for the neighbors can be determined. Schrack [Sch92] introduced efficient neighbor finding methods in linear quadtrees by making use of bit operations. Equal sized neighbors can be found in constant-time regardless of the depth of the node in tree.

These ideas initially developed for quadtrees later provided the basis for many labeling schemes and pointerless representations developed for simplicial decompositions in 2- and 3-dimensions as described below.

Right Triangulated Irregular Networks(RTIN): The RTIN approach by Evans, Kirkpatrick and Townsend [EKT01] introduces a hierarchical data structure for representing height fields to provide approximations of the terrain at different levels-of-detail. It is based on a triangulation of the underlying two-dimensional space using right-angled triangles. This subdivision is basically equivalent to the 2-dimensional special case of Maubach's bisection scheme, where the initial coarse triangles are obtained by subdividing the square domain into two triangles by adding one of the diagonals of the square. The triangles are then recursively subdivided by connecting the right-angled vertex to the midpoint of the hypotenuse. (This is equivalent to both the new-vertex-bisection and the longest-edge bisection.) This hierarchy is also referred to as a *triangle bintree*. To avoid cracks on the approximating surface, a compatible triangulation is guaranteed by *propagation* of splits in a way conceptually equivalent to Mitchell's method [Mit92].

The main focus of the paper is on developing efficient data structures for representing this binary tree of right triangles and fast neighbor finding. As illustrated in Figure 2.7 [EKT01], each node (triangle) t in the hierarchy is labeled by the path code, which is constructed by concatenating the bit for the child type (0 for left child, 1 for right child) of each node on the path from the root to t. The (x, y) coordinates of the vertices of the triangulation do not have to be stored, as they can be computed easily from the label of the triangle. In addition, the representation is pointerless eliminating storage for the child

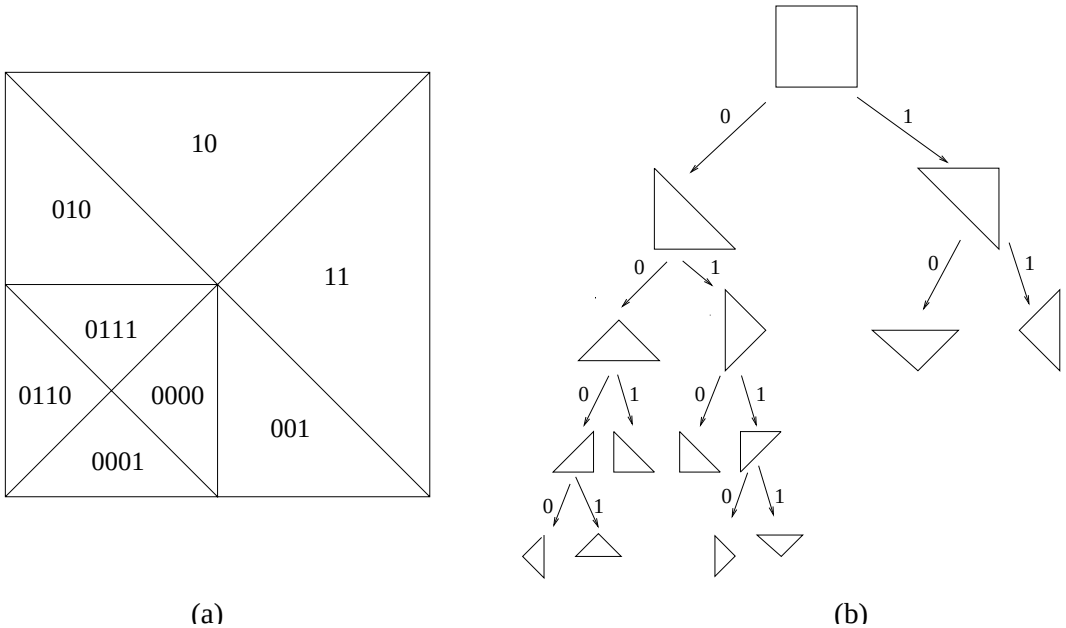


Figure 2.7: (a) 2-dimensional bisection-based mesh (b) corresponding tree representation.

and parent pointers, and thus, it is space efficient. The nodes are stored in an array (as linear quadtrees), and a node is accessed by indexing this array using the node's label, which is treated as the binary representation of an integer. Note that, the labels of the children and parent of a node can be easily determined from the label of the node. The height values associated with (x, y) coordinates are stored in a 2-dimensional array, since the full resolution of vertices corresponds to a uniform grid. This is more efficient than storing in each node the height value corresponding to the midpoint of its hypotenuse, since the straightforward way would lead to storing each height value twice.

The other major focus of this paper is on providing an efficient neighbor finding scheme applicable to this hierarchy. This is important since many algorithms applied on terrains require traversing the approximation surface from one triangle to the adjacent one. The neighbors of a triangle are defined such that the i-*neighbor* is the one that shares the

edge opposite the i^{th} vertex. First, they have provided a recursive function that returns the label of the *same-size* i-neighbor of a triangle given its label. After the same-size neighbor is determined, the actual i-neighbor (which may or may not be the same-size neighbor) can be computed with a constant number of additional steps. The time complexity of the recursive algorithm is proportional to the depth of the triangle. They have also shown how these computations can be performed with a small number of arithmetic and bitwise logical operations in constant-time, provided that the path code fits in a single word.

Multiresolution Visualization and Compression of Global Topographic Data: Within the context of describing a compressed multiresolution hierarchy of the same triangle bintrees for height fields, Gerstner [Ger03] used similar methods to label triangles with bitcodes and manipulate bitcodes to find neighbors for determining the shared refinement vertex. In addition, an efficient mesh traversal scheme (corresponding to the depth-first traversal) based on this triangle numbering is described. Triangles are classified into up- and down-triangles, and a triangle's type can be identified from its bitcode. An up-triangle can only be followed by a triangle at the same level or one level higher. A down triangle can only followed by a triangle at the same level or one level lower. Encoding *stay on the same level* with 0 and a *change of level* with 1, the entire triangulation can be encoded by a starting triangle and one bit per each of the other triangles. The multiresolution DEM defined over this triangulation is stored in two one-dimensional arrays: one containing the height values in the order they appear in the tree traversal (taking special care to avoid duplicates), and one containing the bitcode of the triangulation. It is also shown how an adaptive triangulation can be extracted from the compressed representation.

Navigating through triangle meshes implemented as linear quadtrees: Lee and Samet [LS00] presented a pointerless representation of triangle quadtrees. They have provided algorithms to navigate between neighboring triangles of greater or equal size based on their location codes. For equal sized neighbors, the algorithms have worst-case constant time complexity, since they require only a few bit manipulation operations. The underlying surface is a sphere, which is approximated by an icosahedron whose 20 faces are equilateral triangles. (They have also considered octahedron and tetrahedron approximations to the sphere.) Each triangular face is then recursively subdivided as in red refinement of triangles generating a triangle quadtree for that face. Neighbor finding algorithms work with the same time complexity within each triangle quadtree (associated with a single face of the icosahedron), as well as for neighboring triangles that are in different base triangles of the icosahedron.

The 20 faces of the icosahedron are labeled using a 6-bit code ranging from 0 to 19. Each triangle in the decomposition has one of two orientations, tip-up and tip-down as shown in Figure 2.8 [LS00]. The children resulting from the subdivision use the bit patterns given in the figure, that is each child concatenates the corresponding two-bits to its parent's path to construct its own path.

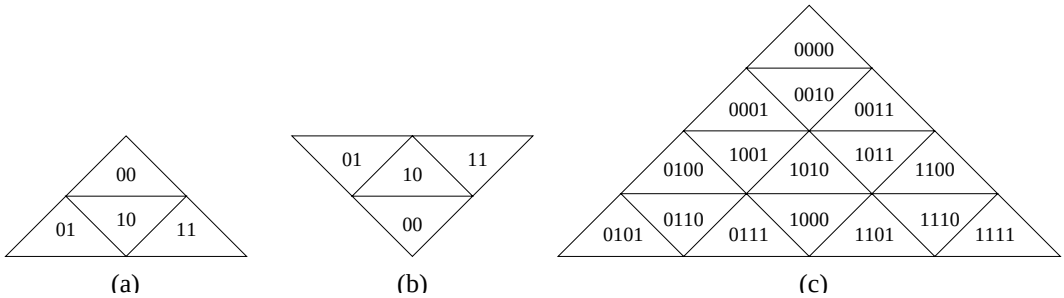


Figure 2.8: (a) tip-up (b) tip-down (c) triangle codes at depth 2.

First, they described how traditional neighbor finding as in quadtrees [Sam90a, Sam92] can be generalized to triangle quadtrees. This algorithm computes a neighbor of a triangle T in three steps. In step 1, the nearest common ancestor of T and its neighbor in given direction (left, right or vertical) is located by determining its path. Step 2 updates the path by concatenating the two-bits corresponding to that child of the nearest common ancestor that contains the neighbor. And, the last step updates the rest of the path to point to the neighbor. A number of relationships are encoded in look-up tables, which are then used in the algorithm. Next, they have explained how these operations can be performed in constant time by using the carry property of addition (subtraction) without searching the path code for the nearest common ancestor and updating the path code as much as its length (i.e. the iterative process is replaced by a few arithmetic operations). Note that the constant time complexity is based on the assumption that the path code fits into a single machine word. Algorithms for finding left, right and vertical neighbors are separately outlined and make use of bit masks to identify and alter bit positions depending on the different cases.

Other labeling methods for the same collections of triangle quadtrees have been proposed by Fekete [Fek90], and Goodchild and Shiren [GS92], but their neighbor finding algorithms have worst-case complexity proportional to the maximum depth of the tree.

Constant-time neighbor finding in hierarchical tetrahedral meshes: Lee, De Floriani and Samet [LDS01] extended the idea of labeling nodes with their binary path code, and constant-time neighbor finding techniques to regular hierarchical tetrahedral meshes generated by bisection. The hierarchy is generated by applying the basic longest edge

bisection to a unit cube initially subdivided into six tetrahedra. (This is the 3-dimensional instance of the refinement method described by Maubach.) They have introduced a particular vertex ordering that enables choosing the edge to be bisected without any computation by ensuring that the longest edge is always the one between the vertices numbered 3 and 4. The three different shapes of tetrahedra that can arise in this particular subdivision are called the $1/2$ *pyramid*, $1/4$ *pyramid* and $1/8$ *pyramid*, providing a very intuitive explanation to the geometry of the subdivision. Similar to the 2-dimensional approached described above, the path from the root to a tetrahedron is used as its label. However, the path does not consist entirely of binary digits; at the highest level the children are labeled from 0 to 5 corresponding to the six coarse tetrahedra.

They have described methods to compute the same-size neighbors of a tetrahedron. First, they have explained an algorithm that runs in time proportional to the length of the code. It works by first locating the nearest common ancestor by scanning the path code from right to left until the particular neighbor direction forces to pass a particular face which is always the one shared by sibling nodes. (the parent of these siblings is the nearest common ancestor.) Then, all that is needed is to invert the last bit to point to the sibling. Thus, regardless of the neighbor type, only one bit need to be inverted. This method is first described for tetrahedra within the same coarse tetrahedron, and then extended to the entire cube. Next, they show for each type of neighbor, how this algorithm can be implemented to run in constant-time by performing just a few bit manipulations. Similar to [LS00], they use bit masks to identify certain bit patterns, and they make use of the carry property of binary addition to determine which bit to invert.

Zhou et al. [ZCK97] described a multiresolution hierarchy of tetrahedra for visual-

izing regular volume data using the same underlying mesh described in [LDS01]. They also represented the binary as an array, without storing any child or parent pointers. The subdivision is performed using only vertex indices without use of actual coordinates.

Symbolic Local Refinement of Tetrahedral Grids: Hebert [Heb94] introduced a labeling scheme for 3-dimensional tetrahedral meshes, which are generated by Maubach's bisection algorithm. His scheme more directly encodes the geometry of the tetrahedra, and also leads to symbolic algorithms to find same-size neighbors. It is based on the fact that the local geometric structure repeats itself on a smaller scale at every three levels of the hierarchy. Basically, each 3-level subtree rooted at level $3m, m \geq 0$ is a scaled copy of the 3-level subtree rooted at level 0. Each tetrahedron can be described by a unique expression of translations, permutations, rotations and scalings, which can be encoded symbolically. This symbolic label of a tetrahedron consists of octal digits. The first three octal digits consist of a permutation, reflection and descendant number that determine the unique position of the tetrahedron within an initial 3-level subtree (rooted at level 0), which repeats itself at level $3m$ subject to a scaling by a factor of $1/2^m$. The remaining digits encode the location of the *lattice origin* of the tetrahedron, which is the center of the smallest enclosing octree box that contains the tetrahedron, and is shared by all the tetrahedra in the 3-level subtree.

The basic bisection step, the compatible refinement algorithm and neighbor finding methods are described by symbolic algorithms manipulating these labels, and can be performed with only integer and logical operations. No neighbor, child and parent links need to be stored as the above methods. In addition, the vertices of a tetrahedron can be

computed by decoding its label, thus, the vertices need not be stored, either.

Neighbor finding methods enumerate different cases for different neighbor types, and for different permutation, reflection and descendant numbers. Three of the four same-size neighbors share the same lattice origin as they are within the same 3-level subtree, hence, only the other three components of the label have to be computed for the neighbor. This is done by following the decision tree based on all possible cases, and can also be implemented efficiently by table-lookups. The remaining neighbor is outside the smallest enclosing octree box, and could be arbitrarily far away. Thus, its lattice origin has to be computed as well. In this paper, it is not described how this can be done efficiently but, this possibly can be done in constant-time by bit manipulation.

Both the formulation of the labeling scheme and the neighbor-finding methods require use of tables and enumerations, which makes the method complicated. In fact, using similar ideas, simpler formulations are possible. Our labeling method for arbitrary dimensions is conceptually a generalization of Hebert's 3-dimensional method, but is formulated in a much simpler way. Hebert's neighbor finding methods, however are not readily generalizable to higher dimensions. We provide neighbor finding methods in arbitrary dimensions with a very compact representation and using very few special cases.

Chapter 3

Efficient Methods for Rendering

In this chapter, we survey methods for efficient rendering that are relevant to the work presented in this thesis. First, we focus on methods for accelerating ray-tracing in particular, as well as methods for accelerating animations. Next, we discuss image-based rendering. We will concentrate on light field methods due their relevance to the work of this thesis.

3.1 Ray-tracing Acceleration Techniques

Ray-tracing is among the most popular techniques for generating complex illumination effects such as shadows, specular highlights, reflection and refraction. The standard Whitted ray-tracer [Whi80] computes global illumination by simulating the path of light rays through the scene. The image is generated by tracing a ray from the viewpoint through each pixel on the image. The color of each pixel is calculated as follows. The viewing ray is intersected with every object in the scene, and the point of intersection closest to the ray origin is determined. The diffuse and specular components of the radiance at this point are computed by a local illumination model, incorporating the contributions of each

visible light source. Then, the reflected and refracted rays are traced recursively, if they exist, and their contributions are added to the local radiance.

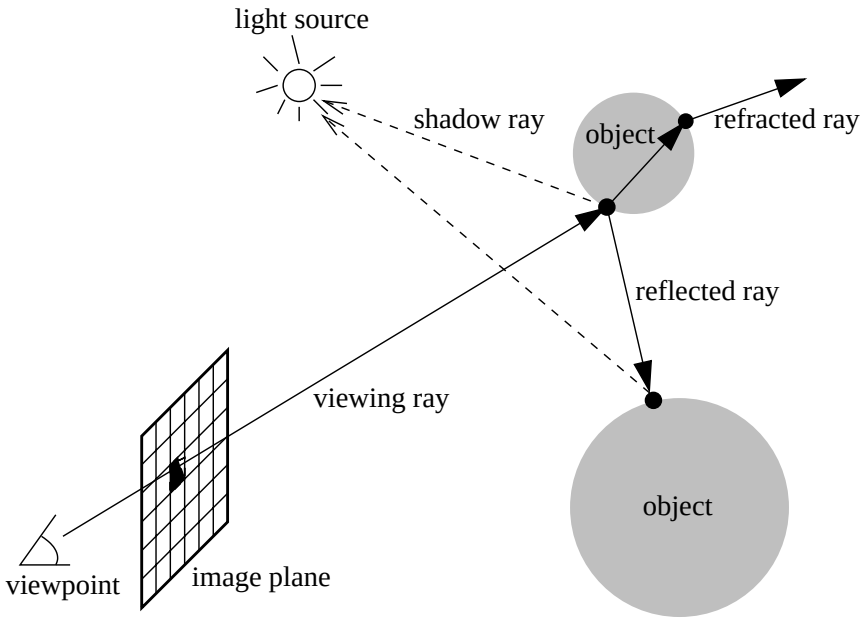


Figure 3.1: Ray-tracer (Whitted)

Ray-tracing is a computationally intensive technique. A major expense lies in the intersection calculations, particularly for scenes that contain complex objects, and in case of multiple reflections and/or refractions. Early research concentrated on accelerating ray-tracing by reducing the cost for intersection computations using the following methods.

Bounding Volume Hierarchies: In these methods [KK86, RW80], each object is enclosed by a simpler volume such as a sphere or a box, allowing for a simpler intersection check. Only those rays that intersect the bounding volume are checked for intersection with the object itself. If most of the rays do not pass close to the object, this results in an overall gain in performance. Furthermore, bounding volumes can be organized in hierarchies—a number of bounding volumes are enclosed by larger

ones—, and so, by a single intersection check, many objects can be eliminated from further intersection checks.

Space Partitioning Techniques: The goal of space partitioning is the same as bounding volume hierarchies, to focus only on a smaller percentage of the scene to determine the closest intersection. However, space partitioning techniques work top-down by subdividing the entire volume containing the scene into smaller volumes. Nonuniform data structures such as BSP-Trees [Kap85] and Octrees [Gla84], or uniform decompositions such as 3-dimensional Grids [FTI86] have been proposed to subdivide the space. In these methods, the candidates for intersection are the objects that lie in the subregions pierced by the ray.

Ray Coherence Techniques: Ray coherence, which is one of the key elements of our work, is exploited by various ray tracing acceleration techniques as well. Ray coherence means that similar rays are likely to follow similar paths in the scene, and so, they are likely to intersect same set of objects at similar points. Beam tracing [HH84] and cone tracing [Ama84] rely on the assertion that, since a bundle of rays follow a similar path, it is more efficient to trace them as a group rather than individually. In cone tracing, Amanatides [Ama84] generalized rays to circular cones represented by an apex, centerline and spread angle, and objects are intersected with cones. To compute reflection/refraction of a cone, the new centerline is computed by reflection/refraction of the centerline of the cone. Beam tracing represents a collection of rays as a generalized cone with a polygonal cross-section. Objects in the scene are assumed to be composed of polygonal facets, so that polygon-polygon

intersection suffices to compute beam-object intersections. Reflections preserve the nature of beams, since reflection is a linear transformation on polygonal surfaces. However, refraction is not linear, and is only approximated by a linear transformation. The ray classification algorithm proposed by Arvo and Kirk [AK87] is another technique that exploits ray coherence. The entire ray space is a 5-dimensional hypercube. A hierarchical data structure, which is the 5-dimensional analog of an octree, is built by recursively subdividing the ray space. Each hypercube in the hierarchy represents a collection of rays originating from a 3-dimensional rectangular volume, and directed through a 2D solid angle. Each leaf is associated with a set of objects that are candidates for intersection with the collection of rays represented by the leaf. During rendering, each ray is mapped to the corresponding 5-dimensional point, the hypercube containing this point is located in the tree, and only those objects in the candidate list of the hypercube are tested for intersection.

The design and cost analysis of data structures for ray-tracing has been of interest in the field of computational geometry. This includes, for example, the work of Mitchell, Mount and Suri on simple cover complexity [MMS97], the work of Aronov and Fortune on low weight triangulations [AF99], cost prediction for ray shooting by Aronov, Brönnimann, Chang and Chiang [ABCC02, ABCC03] and hierarchical uniform grids (HUG) space partitioning data structure introduced by Cazals, Drettakis and Puech [CDP95]. In a later study, Cazals and Puech compared uniform grids, recursive grids, and HUG for ray-tracing, and demonstrated statistically that recursive grids and HUG outperform uniform grids for non-uniform distribution of scene objects [CP97].

Interactive Ray-tracing and Accelerating Animations: Recent research has focused on interactive ray-tracing and accelerating animation sequences. This requires fast generation of ray-traced images from multiple viewpoints. Parker *et al.* accelerate rendering relying on multiprocessor hardware [PMS+99]. Their implementation is brute-force—it explicitly traces rays through each pixel.

Some systems accelerate animation sequences by exploiting frame-to-frame coherence. The main idea is to reuse pixels from the previous frame by reprojection and only recompute or possibly refine the potentially incorrect pixels [AH95, Bad88]. In the method described by Adelson and Hodges [AH95], the reference frame is completely ray-traced and along with each sampled pixel, the 3-dimensional intersection point, surface normal and diffuse color is stored. The method proceeds in three steps to generate a new frame. The first step is to *reproject* the intersection points from the previous frame to the new position. If more than one sample from the previous frame is projected to the same pixel of the new frame, the closest one to the viewpoint is chosen, and verified in the second step. The second step is called the *verification* phase. The projected points are checked for self-occlusions. Back faces are identified using the dot product of the viewing vector and the surface normal at the intersection. Since the method is restricted to convex objects, it is guaranteed that parts with forward-facing normals are not subject to self-occlusions. Then, the system checks for the points which were visible in the previous frame, but occluded in the new frame. These cases are determined by casting a ray from the viewpoint to the intersection point, and checking for potential occluding objects along the ray. For the points that became disoccluded in the new frame, standard ray-tracing is used. The last step, *enhancement*, adds the view-dependent shading phenomena. However, this is

achieved by casting arbitrary levels of reflection and refraction rays as in the standard ray-tracer. Thus, this method introduces savings in time only for diffuse scenes.

Chapman *et al.* [CCD91] described another method to accelerate generation of animation sequences by computing a *continuous intersection* of rays with a polygonal scene given the trajectory of the viewpoint.

Walter *et al.* cache the results while rendering a frame and reproject previously cached samples to approximate the current frame [WDP99]. Similarly, in Larson's Holodeck system, rays are computed, cached and reused for subsequent frames by utilizing a 4-dimensional data structure [Lar98].

Interpolant Ray Tracer: The ray-tracing acceleration technique most similar to our work is the Interpolant Ray Tracer system described by Bala, Dorsey and Teller [BDT99]. In their system, they distinguish between visibility and shading components of the ray tracer and accelerate them independently. Acceleration of shading is similar to our approach of accelerating intersection computations such that they make use of the fact that radiance is a smoothly varying function over the ray space most of the time, and a sparse set of samples can be interpolated to approximate radiance. Radiance samples are cached in a 4-dimensional quadtree-based data structure, called a *linetree*. However, we differ in that our data structure is designed to map rays to geometric attributes such as normals and reflection rays rather than mapping rays to radiance, and we are primarily interested in fast rendering of reflective and refractive objects from multiple viewpoints. Storing and interpolating geometric attributes rather than radiance lets the object be represented independent of the illumination and the geometry of the environment unlike their method.

Moreover, their quadrilinear interpolation requires that the ray trees of all sixteen samples used for interpolation be identical to constitute a valid interpolant. This strong requirement reduces the number of cases where interpolation could be substituted for ray-tracing, especially when there are many reflective and refractive objects in the scene. Instead, we apply heuristics that would allow us to use interpolations in more cases while trading off quality to some extent. In their system, the approximation error is conservatively bounded at the expense of allowing only convex objects. We do not provide theoretical bounds. But, in addition to simple convex objects, our system supports rendering bicubic patches. They also accelerate visibility independently in case of multiple frames. Interpolants from the previous frame are reprojected to the new viewpoint, and used to shade the pixels they cover in the new frame. We have also investigated the use of compatible simplicial decompositions for subdividing the ray space. This has provided us with a simpler data structure as well as continuous interpolations in contrast to theirs.

Hybrid Rendering: The idea of treating a reflective/refractive object as a local lens object, which maps incoming rays to outgoing rays is used by Hakura and Snyder [HS01]. In this respect their work is similar to ours, and was developed both independently and concurrently with ours. They apply this basic approach in a different setting than ours. Their method is based on partitioning local and distant geometry as in environment mapping. They combine ray-tracing of local geometry of reflective/refractive objects with hardware supported environment maps to approximate distant geometry. A set of layered environment maps are generated in pre-processing for each local object and over a number of viewpoints. At run time, they dynamically trace rays through vertices of the local

object and determine where the ray exits the object. This outgoing ray then is used to access the appropriate environment map. Unlike ours, their system traces rays accurately for each viewpoint; there is no ray parameterization or ray interpolation.

3.2 Image-Based Rendering

Image-Based Rendering (IBR) is a relatively new rendering paradigm, which supports fast rendering of scenes from multiple viewpoints. A good survey can be found in [MG99]. These systems store a database of pre-rendered or pre-acquired images of a 3-dimensional scene from a set of viewpoints, and use them to synthesize new views of the scene. The main advantages of IBR are:

- the rendering time is independent of the geometrical complexity of the scene,
- the reference images can be of real scenes, eliminating the need for modeling complex geometry and light effects.

Image-based rendering has some common elements with our work in the sense that both methods rely on sampling and reconstruction of scenes to accelerate rendering from multiple viewpoints.

Image Reprojection and View Interpolation: By storing a depth (disparity) value along with each pixel of the reference images, new images can be generated by reprojecting pixels from one or more reference images to the desired view [MB95]. This method is called *image warping* or *image reprojection*. Depth information might be stored explicitly or is encoded implicitly in the form of correspondences between pairs of points in

different projections. The method described by Mark *et al.* [MMB97] treats the reference image as a mesh. They use 3-dimensional warping to perturb the vertices of the mesh. The reconstruction in the new frame occurs by rendering the perturbed mesh triangles. The pixel colors are linearly interpolated across the reconstructed mesh triangle. They perform this warping from the two nearest reference images one at a time, and composite the results.

The view interpolation method proposed by Chen and Williams [CW93] is a similar approach to image reprojection, but instead of reprojecting one or more images to a new view, new views are interpolated between reference images associated with nearby viewpoints. This method, too, relies on pixel-to-pixel correspondences between each pair of reference images.

An important challenge in these methods is handling gaps—when reprojecting images, surfaces that were not visible in the reference images might become visible. Chen and Williams propose using multiple reference images to avoid gaps. Otherwise, they fill the gaps by interpolating nearby pixels. Another novel way of solving this problem is to use *Layered Depth Images (LDI)* [SGHS98]. An LDI stores with each pixel, multiple color and depth information corresponding to all the surfaces intersected by the ray rather than storing only the information about closest intersection.

Most of these image-based methods using image reprojection cannot handle non-diffuse phenomena such as specular highlights and reflection, since they assume that every point in the scene will have the same color when viewed from different directions.

Lumigraph and Light Field Rendering: Among the IBR methods, the most relevant to our work is the Lumigraph [GGSC96] and Light Field Rendering (LFR) [LH96] techniques. Both systems are based on dense sampling of the *plenoptic function* [AB91]. The plenoptic function is a 5-dimensional quantity describing the flow of light at every position (x, y, z) for every direction (θ, ϕ). By considering only the light leaving a bounded object (or scene), the domain of the plenoptic function can be reduced to 4-dimensional, since the radiance along a ray is constant. The Lumigraph and Light Field techniques capture and represent the plenoptic function in a bounded environment, and use this information to render new images of the environment from an arbitrary viewpoint. However, the viewpoint is restricted to lie outside the enclosed environment.

We will explain only the Lumigraph here, since Light Field Rendering is very similar. The 4-dimensional plenoptic function is discretized by means of a data structure called a Lumigraph. The scene is enclosed within a cube for simplicity. The surface of the cube holds all the radiance information of the scene. At any point in space, the radiance along any ray in any direction can be determined by tracing the ray to the surface of the cube, assuming the empty space outside the cube does not alter the radiance.

Rays are parameterized by the so-called *two-plane parameterization*. To conform to the 4-dimensional representation of the plenoptic function, each ray is represented by its intersection points with two parallel planes, hence a 4-dimensional quantity. The first plane is actually the cube face with axes labeled as s and t. The direction is parameterized by a second plane parallel to the first plane, with axes labeled as u and v. Thus, a point (s, t, u, v) in the Lumigraph corresponds to a ray intersecting the first plane at (s, t) and the second at (u, v). See Figure 3.2(a) [GGSC96].

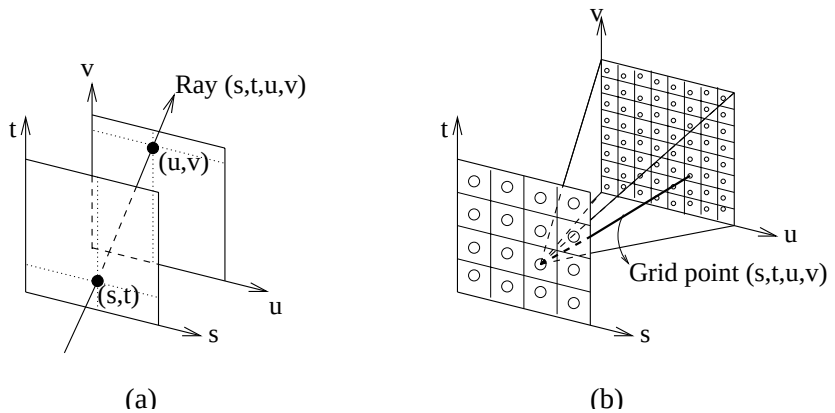


Figure 3.2: Two-plane parameterization

During preprocessing, the data structure is built by sampling the 4-dimensional Lumigraph function by uniformly subdividing in all four dimensions resulting in a regular grid structure on both planes. See Figure 3.2(b). The associated radiance value is stored associated with each grid point. One way of viewing a Lumigraph is as a 2D array of images with viewpoints on a regular grid of the st plane. The image associated with each (s, t) point is referred to as the *uv image*.

It is easy to see how to get samples into the Lumigraph from an arbitrary image and how to reconstruct a new image from the Lumigraph. For any image, when the viewpoint and the pixel location is fixed, a ray is associated with each pixel—originating from the viewpoint and passing through the pixel. Let (s, t, u, v) be the parameterization of this ray. Given an input image, the value to be stored at the location (s, t, u, v) of the Lumigraph is the color of the image at the pixel intersected by the ray parameterized as (s, t, u, v). On the other hand, given a Lumigraph, a pixel on a new image that is associated with the ray (s, t, u, v), can be constructed by using the value of the Lumigraph function at this parameter value.

The radiance along any ray from any viewpoint is interpolated from the radiance values from the nearest sixteen samples as follows. Each ray passes through a grid cell on the st plane, and a grid cell on the uv plane. Since each grid cell is bounded by four grid points, there are sixteen nearby radiance samples corresponding to the rays from each of the four (s, t) grid points to each of the four (u, v) points. The sixteen samples are quadrilinearly interpolated to give an approximate radiance for the query ray.

Gortler *et al.* [GGSC96] describe a texture-mapping-based rendering to perform the reconstruction of image with hardware acceleration. The uv images associated with the (s, t) points are used as textures and the process of blending textures to approximate quadrilinear interpolation is described in detail.

To handle complex effects such as reflection, refraction and specular highlights with reasonable quality, these methods should sample very densely. The Lumigraph and LFR are simpler and considered closest to pure image-based rendering, since they do not require additional information like depth or optical flow. However, since they rely on dense sampling, they require very large amounts of storage. This is partly because of oversampling in the regions where radiance is smooth due to the fixed sampling rate. Compression mechanisms are proposed in both [GGSC96] and [LH96], however, this introduces additional overhead of decompression when rendering from the Lumigraph.

Schirmacher *et al.* [SHS99] proposed adaptive acquisition of images for Lumigraphs, by optimizing the set of viewpoints with respect to the quality of image reconstruction using the images from these viewpoints. This is achieved by an *apriori* error estimate predicting the gain in reconstruction quality when adding a new viewpoint to the Lumigraph data structure. However, this estimate requires using geometric informa-

tion. When a new candidate viewpoint is to be rendered, they warp the nearest already acquired images to the candidate point, and estimate the resulting error of object visibility and color. By this error estimate, they decide how accurately the image for a viewpoint could be reconstructed by warping. If it is accurate enough, there is no need to add the new viewpoint to the database. Their algorithm constructs an adaptive mesh of already sampled viewpoints, predicts the gain of adding new viewpoint at each edge and chooses to split the edge with the greatest gain.

As opposed to adaptive Lumigraphs, Camahort *et al.* [CLF98] argue that uniformly sampled Lumigraphs are advantageous. Since the two plane parameterization proposed in [GGSC96] and [LH96] used different arrangements of pairs of planes, when the camera crosses the boundary of two plane-pairs, there could be noticeable artifacts. To remedy this, they propose two uniform sampling strategies referred to as the two-sphere (2SP) and sphere-plane (SPP) parametrizations. An additional advantage is that uniform sampling is essential for compression techniques like Fourier transforms.

For the 2SP parameterization, rays are represented by their two intersection points (s, t) and (u, v) with two overlapping spheres. The object is enclosed by a tight sphere and the sphere surface is subdivided into nearly equilateral triangles, called patches. One light field sample is stored for each ordered pair of patches. For the SPP parameterization, a ray is defined by a normal direction, (θ, ϕ) specifying a plane which passes through the center of the sphere, and by some point (u, v) on that plane. Both the normal and the point are sampled from a uniform distribution.

Sloan *et al.* [SCG97] considered a number of methods to improve the performance of lumigraph rendering trading off quality for time. The main motivation is to limit the

number of images used for the reconstruction of a new image due to limited texture memory and main memory. Their methods fall into two categories, those that use a smaller set of textures than a full rendering, and a method that uses the current reconstructed image as a new texture itself for subsequent nearby frames. They organize the viewpoints in an adaptive triangle mesh. New viewpoints (their associated uv images) are brought in, and unused ones are deleted using a benefit/cost model.

Heidrich *et al.* [HLCS99] proposed a light field method focusing on rendering refractive objects. Basically, they use the Lumigraph data structure with all methodologies developed in [GGSC96], but, the RGB color triplet associated with each grid point is replaced with four numbers representing the direction component of a refraction ray. This refraction ray is then used to access a static environment map, ignoring local effects further from the object, or to access another light field. Their method has similarities to ours in that they associate refraction rays with samples. However, since their system is built on a lumigraph/light field structure, it relies on uniform dense sampling of the rays for capturing clear object boundaries and handling discontinuities. This results in the main problem with the light field methods, large storage requirements. Our method, on the other hand, samples rays adaptively and applies a variety of heuristics to achieve high quality discontinuity rendering at lower sampling rates. Their method rely on static large structures built in a costly preprocessing phase, whereas our focus is on building dynamic structures with caching—avoiding the preprocessing step and sampling rays on demand. Moreover, we sample and interpolate normal vectors and intersection points in order to compute the diffuse and specular components of shading. Our methods apply to a general framework of ray-tracing.

Lischinski and Rappoport use LDIs to render both view-independent (geometry and diffuse shading) and view-independent (specular highlights, reflections) scene information from new viewpoints [LR98]. Their method is an integration of the Image-Based Rendering and Light Field methods. All view-independent scene information is represented using three orthogonal high resolution LDI's—called the *layered depth cube* (LDC). The view-dependent information is represented as a separate and larger collection of low resolution LDIs corresponding to various directions—called the *layered light field* (LLF). Each sample in the LLF contains, in addition to its depth, the total radiance leaving the scene sample in the direction of projection. Thus, each of the LDIs samples the light field along oriented lines parallel to the direction of projection. Lischinski and Rappoport describe a rendering algorithm to combine these two components. Rendering proceeds in two stages: In the first stage a *primary image* is constructed by applying 3-dimensional warping to view-independent LDIs similar to the method described in [SGHS98]. In the second stage, first the specular component is computed by simply evaluating the local shading model for each visible light source. Then, the reflections are computed by either of the two methods:

- *light field gather:* In this technique, the light field is reconstructed from the LLF. To determine the view-dependent radiance leaving the point of interest in a certain direction, first, the incoming radiance from each direction in the LLF is computed by interpolation of nearby samples. Then, the BRDF at that point is used to weigh each incoming radiance, and compute the outgoing radiance.

- *image-based ray-tracing:* This method traces rays through the LLF.

Light field gather is good for glossy objects with fuzzy reflections, but not for perfect mirror reflections. Ray-tracing is used for such cases. Lischinski and Rappoport's method is a nice combination of different techniques, but is computationally expensive.

Chapter 4

The Ray Interpolant Tree for Efficient Ray-tracing

4.1 Introduction

There is a growing interest in algorithms and data structures that combine elements of discrete algorithm design with continuous mathematics. This is particularly true in computer graphics. Consider for example the process of generating a photo-realistic image. The most popular method for doing this is *ray-tracing* [Gla89a]. Ray-tracing models the light emitted from light sources as traveling along rays in 3-space. The color of a pixel in the image is a reconstruction of the intensity of light traveling along various rays that are emitted from a light source, transmitted and reflected among the objects in the scene, and eventually entering the viewer's eye.

There are many different methods for mapping this approach into an algorithm. At an abstract level, all ray-tracers involve forming an image by combining various continuous quantities, or *attributes*, that have been generated from a discrete set of sampled rays. These continuous attributes include color, radiance, surface normals, and reflection and refraction vectors. These attributes vary continuously either as a function of the location

on the surface of an object or as a function of the location of the viewer and the locations of the various light sources in 3-space. The reconstruction process involves combining various discretely sampled attributes in the context of some illumination model.

Producing images by ray-tracing is a computationally intensive process. The degree of realism in the final image depends on a number of factors, including the density and number of samples that are used to compute a pixel's intensity and the fidelity of the illumination model to the physics of illumination. Scenes can involve hundreds of light sources and from thousands to millions of objects, often represented as smooth surfaces, including implicit surfaces [Blo97], subdivision surfaces [ZSS96], and Bézier surfaces and NURBS [FvDFH90]. Reflective and transparent objects cause rays to be reflected and refracted, further increasing the numbers of rays that need to be traced. In traditional ray-tracing solutions, each ray is traced through the scene as needed to compute the intensity of a pixel in the image [Gla89a]. To achieve smoothness and avoid problems with aliasing, many rays may be shot for each pixel. A high resolution rendering can easily involve shooting on the order of hundreds of millions of rays. Much of the computational effort involves determining the first object that is intersected by each ray and the location that the ray hits.

In this chapter, we propose an approach to help accelerate this process by reducing the number of intersection calculations. Our algorithm facilitates fast, approximate rendering of a scene from any viewpoint, and is also useful when the scene is rendered from multiple viewpoints, as arises in computing animations. Rather than tracing each input ray to compute the required attributes, we collect and store a relatively sparse set of *sampled rays* and associate a number of continuous geometric attributes with each sample in

a fast data structure. We can then use inexpensive *interpolation* methods to approximate the value of these sampled quantities for other input rays. Using an adaptive strategy, it is possible to avoid oversampling in smooth areas while providing sufficiently dense sampling in regions of high variation. We dynamically maintain a *cache* of the most recently generated samples, in order to reduce the space requirements of the data structure.

The information associated with a given ray is indexed according to the directed line that supports the ray, which in turn is modeled as a point in a 4-dimensional *line space*. The idea of associating radiance information with points in line space has a considerable history, dating back to work in the 1930's by Gershun on vector irradiance fields [Ger39] and Moon and Spencer's concept of photic fields [MS81], and more recently Light Fields introduced by Levoy and Hanrahan [LH96] and the Lumigraph introduced by Gortler, et al. [GGSC96].

Our notion is more general than the Light Fields and the Lumigraph because we consider interpolation of any continuous information, not just radiance. Most methods for storing light field information in computer graphics are based on discretizing the space into uniform grids. In contrast, we sample rays adaptively, concentrating more samples in regions where the variation in attribute values are higher. In addition, both Light Fields and the Lumigraph sample the entire space of rays in a pre-processing step, which result in high pre-processing times as well as very high space requirements. We, on the other hand, fill our data structure on-demand, that is, we generate samples only when they are needed by some interpolation.

The most closely related work to ours is the Interpolant Ray-tracer system introduced by Bala, Dorsey, and Teller [BDT99], which combines adaptive sampling of *radi-*

ance information and interpolation for rendering convex objects. Our method generalizes theirs by storing and interpolating not only radiance information but other sorts of continuous information, which may be relevant to the rendering process. In particular, we store and interpolate information such as normal vectors, intersection points, reflection and refraction rays. Unlike radiance interpolants [BDT99], our method allows the objects to be represented independent of the illumination and the geometry of the environment. In Section 4.6.5 we demonstrate the value of our approach. We also allow nonconvex objects. Unlike their method, however, we do not provide guarantees on the worst-case approximation error.

4.1.1 Design Issues

The approach of computing a sparse set of sample rays and interpolating the results of ray shooting is most useful for rendering smooth objects that are reflective or transparent, for rendering animations when the viewpoint varies smoothly, and for generating high-resolution images and/or antialiased images generated by supersampling [Gla89a] in which multiple rays are shot for each pixel of the image.

Although we have motivated our approach from the perspective of ray-tracing, there are a number of applications having to do with lines in 3-space that can benefit from this general approach. To illustrate this, in addition to ray-tracing, we have studied another application involving volume visualization with applications in medical imaging for radiation therapy.

There are a number of issues that arise in engineering a practical data structure for interpolation in line space. These include the following.

How and where to sample rays? Regions of space where continuous information varies more rapidly need to be sampled with higher density than regions that vary smoothly.

Whether to interpolate? In the neighborhood of a discontinuity, the number of rays that may need to be sampled to produce reasonable results may be unacceptably high. Because the human eye is very sensitive to discontinuities near edges and silhouettes, it is often wise to avoid interpolating across discontinuities. This raises the question of how to detect discontinuities. When they are detected, is it still possible to interpolate or should we avoid interpolation and use standard ray-tracing instead?

How many samples to maintain? Even for reasonably smooth scenes, the number of sampled rays that would need to be stored for an accurate reconstruction runs well into millions. For this reason, we *cache* the results of only the most relevant rays. What are the space-time tradeoffs involved with this approach?

In the sequel, we investigate these and other questions in the context of a number of experiments based on the applications mentioned above.

4.2 Mapping Rays to Geometric Attributes and Ray Coherence

We can distinguish two major components in a ray-tracer. A *geometric component*, which is responsible for calculating the closest visible object point along a specific ray, and other geometric attributes such as the surface normal at that point, and a *shading component*, which computes the color of that point. Our approach primarily aims to accelerate the geometric component.

The key idea of our method is that each object can be modeled abstractly as a function f that maps input rays to a set of geometric attributes that are used in color and shading computation. These attributes depend on the object's surface reflectance properties. For objects whose surfaces are neither reflective nor transparent, denoted *simple surfaces*, the function returns the point of intersection and the surface normal at this point. For objects whose surfaces are either reflective or transparent, the function additionally returns the *exit ray*, that is, the reflected or refracted ray, respectively, that leaves the object's surface after a number of reflections or refractions, respectively. The exit ray is represented by its origin, the *exit point*, and directional *exit vector*. In general, objects that are both reflective and refractive could be handled by associating multiple exit rays with an input ray, but our implementation currently does not support this. These quantities are depicted in Figure 4.1 and the function is described schematically below.

For simple surfaces: $f : \text{Ray} \rightarrow \{\underline{\text{Normal}}, \underline{\text{IntersectionPoint}}\}$

Otherwise: $f : \text{Ray} \rightarrow \{\text{Normal}, \text{IntersectionPoint}, \underline{\text{ExitPoint}}, \underline{\text{ExitVector}}\}$

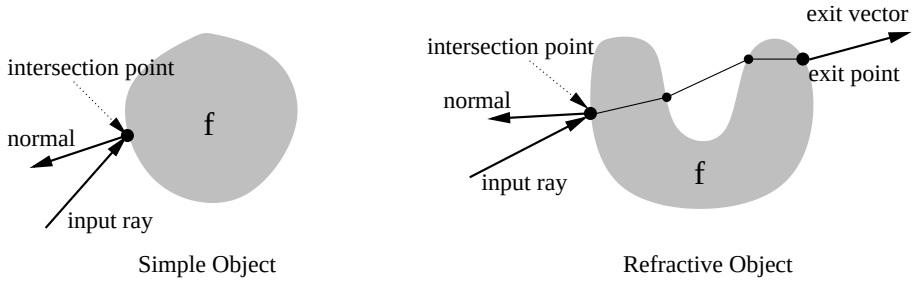


Figure 4.1: Geometric attributes.

We refer to the combination of the underlined attributes as the *output ray*. The output ray serves as the *key* of the entire set of attributes, and has a special function in the construction and use of the data structure. Basically, the *distance* between two attribute

sets is defined to be the *distance* between their output rays. This distance is used in adaptive subdivision of the data structure as will be explained later.

For many real world objects that have large smooth surfaces, f is expected to vary smoothly. In the context of ray-tracing, this is referred to as *ray coherence*. Nearby rays follow similar paths, hit nearby points having similar normal vectors, and hence are subject to similar reflections and/or refractions.

In the neighborhood of discontinuities, however, nearby input rays may follow quite different paths. We use additional heuristics to permit interpolation when the parts of an interpolant lie on different sides of a discontinuity. While avoiding interpolation across the discontinuity boundary, we still interpolate on either side. In cases where we cannot find sufficient evidence to interpolate, we perform ray-tracing instead.

In a traditional ray-tracer, each object is associated with a procedure that computes intersections between rays and this object. For objects whose ray-object intersection computations are expensive (such as Bézier surfaces) and boundaries are sufficiently smooth, we replace this intersection procedure with a data structure, which will be introduced in Section 4.3. This data structure approximates the function f through interpolation.

4.3 The Ray Interpolant Tree

In this section we introduce the main data structure used in our algorithm, the *RI-tree* or *ray interpolant tree*. A RI-tree is associated with a single *object* of the scene, where an object is loosely defined to be a collection of logically related surfaces. The object is enclosed by an axis-aligned bounding box. The data structure stores the geometric

attributes associated with some set of sampled rays, which may originate from any point in space and intersect the object's bounding box.

4.3.1 Parameterizing Rays as Points

In ray-tracing implementations, a common way to represent a ray is by its origin and unit-length directional vector. Since two spherical angles are sufficient to define a unique direction vector, geometrically a ray has only five degrees of freedom. Thus, a ray in 3-space can be represented as a point in a 5-dimensional space. For the most part, it is possible to achieve a reduction in the dimension of the space by representing a ray by a directed line.

Consequently, we model each ray by the directed line that contains the ray. Directed lines can be represented as a point lying on a 4-dimensional manifold in 5-dimensional projective space using Plücker coordinates [Som34], but we will adopt a simpler and popular representation, called the *two-plane parameterization* [BDT99, GGSC96, LH96]. A directed line is first classified into one of 6 different classes (corresponding to 6 *plane pairs*) according to the line's *dominant direction*. The dominant direction is defined to be the axis corresponding to the largest coordinate of the line's directional vector and its sign. (Ties may be broken arbitrarily.) These classes are denoted $+X, -X, +Y, -Y, +Z, -Z$. The directed line is then represented by its two intercepts (s, t) and (u, v) with the *front plane* and *back plane*, respectively, that are orthogonal to the dominant direction and coinciding with the object's bounding box. To define the planes, the corresponding bounding box faces are extended on both sides by the distance between the two planes, so that a ray R with dominant direction d intersects both planes of the plane-pair corresponding

to the dominant direction d. For example, as shown in Figure 4.2, ray R with dominant direction $+X$ first intersects the front plane of the $+X$ plane pair at (s, t), and then the back plane at (u, v), and hence is parameterized as a 4-tuple (s, t, u, v). Thus, each ray is represented by a 4-dimensional point. Note that, the $+X$ and $-X$ dominant directions involve the same plane pair but differ in the distinction between the front and back planes. It is easy to see that if the bounding box has width w (along x), height h (along y) and depth d (along z), then a ray that intersects the bounding box and whose dominant direction is $\pm X$ intersects the front and back planes through two parallel rectangles of height $h + 2w$ and depth $d + 2w$. (See Figure 4.2 [BDT99].)

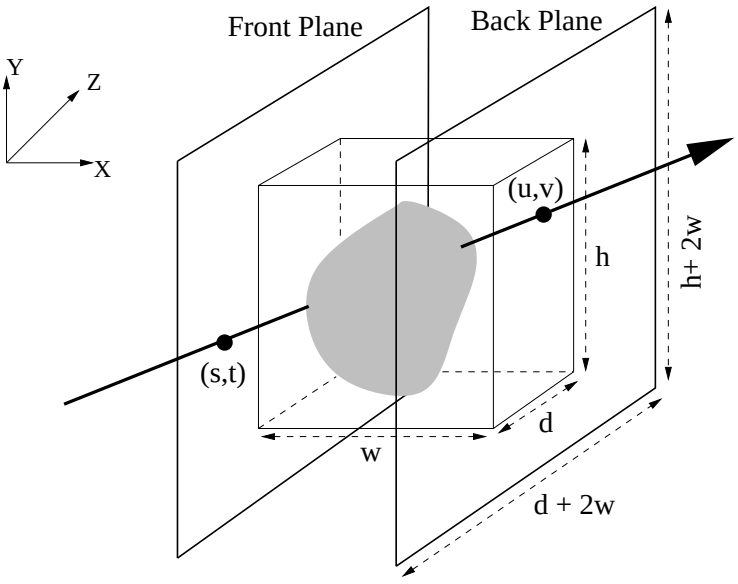


Figure 4.2: The two-plane parameterization of directed lines. The +X plane pair is shown.

4.3.2 The Structure of the RI-tree

The RI-tree is a binary tree based on a recursive subdivision of the 4-dimensional space of directed lines. It consists of six separate 4-dimensional kd-trees [Ben75, Sam90b] one for

each of the six dominant directions. The root of each kd-tree is a 4-dimensional hypercube in line space containing all rays that are associated with the corresponding plane pair. The 16 corner points of the hypercube represent the 16 rays from each of the four corners of the front plane to the each of the four corners of the back plane. Each node in this data structure is associated with a 4-dimensional hyperrectangle, called a *cell*. The 16 corner points of a leaf cell constitute the ray samples, which form the basis of our interpolation. When the leaf cell is constructed, these 16 rays are traced and the associated geometric attributes are stored in the leaf.

4.3.3 Adaptive Subdivision and Cache Structure

The RI-tree grows and shrinks dynamically based on demand. Initially, only the root cell is built by sampling its 16 corner rays. A leaf cell is is subdivided by placing a cut-plane at the midpoint orthogonal to the coordinate axis with the longest length. In terms of the plane pair, this corresponds to dividing the corresponding front or back plane through the midpoint of the longer side. We partition the existing 16 corner samples between the two children, and sample eight new corner rays that are shared between the two child cells. These new rays are illustrated in Figure 4.3 in the case that the s-axis is split. For this case, front planes corresponding to the child cells are the split halves of the front plane of the parent, but back plane corresponding to the child cells is the same as the parent.

Note that most of the corner points in the subdivision, that is, most of the sample rays, are shared by more than one cell. Thus, we have to be careful not to sample the same ray more than once in order to keep the cost of sampling as low as possible. For rays shared between a parent and its children, or rays shared by two siblings, this can

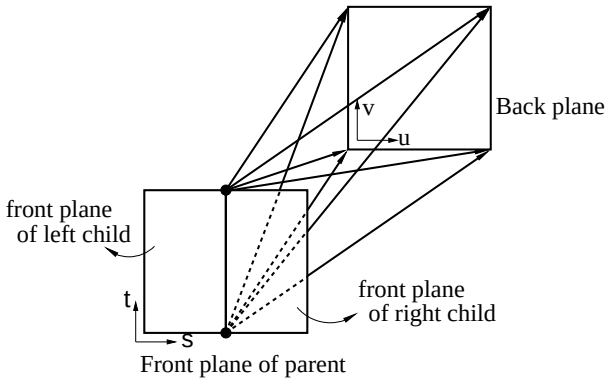


Figure 4.3: Subdivision along s-axis.

be easily be done by careful implementation. However, for those rays that are shared by neighboring cells which are arbitrarily far away in the tree, additional care has to be taken. For this purpose, we use a hash-table of rays, which is indexed by the 4-dimensional representation of the ray. One hash-table per plane-pair is used. The average search time to determine whether a ray is already sampled or not is constant, as supported by empirical evidence.

Rays need to be sampled more densely in some regions than others, for example, in regions where geometric attributes have greater variation. For this reason, the subdivision is carried out adaptively based on the distance between output attributes. The distance between two sets of output attributes are defined as the distance between their associated output rays. We define the *distance* between two rays to be the L_2 distance between their 4-dimensional representations. We determine whether a cell should be subdivided based on the error of approximation corresponding to the midpoint of the cell. We first compute the correct output ray associated with the midpoint of the cell, and then we compute an approximate output ray by interpolation of the 16 corner rays for the same point. If the distance between these two output rays is smaller than a given user-defined *distance*

threshold, we decide that the output attributes for any input ray that fall in that cell can be approximated with an acceptably low error by interpolation of the sixteen corner rays, and we stop subdividing. Otherwise, the cell is subdivided into two equal-sized children. We also impose an upper limit on the tree depth to prevent the tree from growing excessively at discontinuity regions. A cell can be subdivided only when the depth of the cell in the tree is less than a user-defined *depth constraint*. When subdivision is not required due to the error evaluation described above or not allowed due to the depth constraint, the leaf is marked as *final*.

If we were to expand all nodes in the tree until they are final, the resulting data structure could be very large, depending on the distance threshold and the depth constraint. For this reason, we only expand a node to a final leaf if this leaf node is needed for some interpolation. Once a final leaf node is used, it is marked with a time stamp. If the size of the data structure exceeds a user-defined *cache size*, then the the tree is pruned to a constant fraction of this size by removing all but the most recently used nodes. In this way, the RI-tree behaves much like an LRU-cache.

Comment on the depth constraint: An absolute bound on the maximum tree depth is a rather unnatural parameter. It is, however, possible to infer this value based on some more geometrically natural parameters. For example, consider instead a *angular similarity constraint*, θ, which bounds the maximum angle between any two input rays that fall within the same leaf cell. In other words, if the angle between any two input rays that lie within the same leaf cell of the subdivision is at most θ, then these samples are sufficiently close to one another that further subdivision is not required.

Let us illustrate how to compute the tree depth constraint from θ. We assume for simplicity that the object has been enclosed within a bounding cube. Since angles are not affected by uniform scaling, we may assume that this is a unit cube. It follows from our parameterization, that for each of the dominant directions, it suffices to consider rays that intersect a pair of parallel squares lying on the front and back planes for this direction, whose side lengths are 3 units. (Consider Figure 4.2 in the case $w = h = d = 1$.) Now, consider any leaf cell of the associated tree for this direction. Such a cell corresponds to the set of rays passing through two rectangular faces, one on each of these two parallel squares. Since the sides of each rectangle are alternatingly split along the side of maximum length, the worst case arises when both sides of these faces are of equal length, say r. (See Figure 4.4.) Among the 16 corner rays sampled for any face pair, it can be shown that the maximum angle occurs between the cross diagonals of two faces that are aligned orthogonally opposite one another, that is, so that the line connecting the centers of these two faces is orthogonal to both faces. This follows from Lemma 4.3.1 and Lemma 4.3.2 presented below. In this case, the diagonals are of equal length and intersect at their midpoints, from which we have

$$\tan\frac{\theta}{2} \;=\; \frac{r\sqrt{2}/2}{1/2} \;=\; r\sqrt{2},$$

And so, $r = (\tan(\theta/2))/\sqrt{2}$. Since four splits are required to halve the side length of a cell, and the initial (root) face is of side length 3, it follows that the maximum depth constraint, as a function of the angular similarity constraint, is

$$\mathit{depth}(\theta) \;=\; 4\lg\frac{3}{r} \;=\; 4\lg\frac{3\sqrt{2}}{\tan\frac{\theta}{2}} \;=\; 4\lg\left(3\sqrt{2}\cot\frac{\theta}{2}\right),$$

where $\lg$ denotes logarithm base 2.

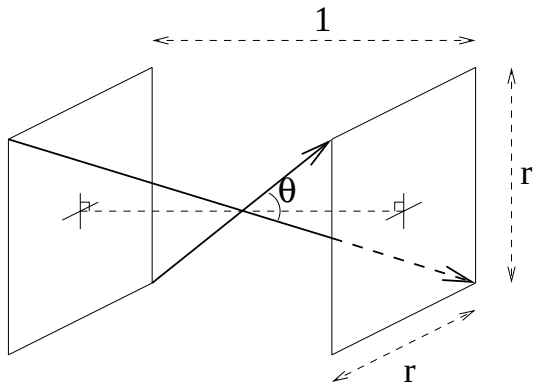


Figure 4.4: Maximum angle is achieved by the cross diagonals of orthogonally opposing faces.

We set a fixed depth constraints in our experiments, but this formulation in terms of an angular similarity constraint would be a more appropriate parameter for software design purposes.

Lemma 4.3.1 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ denote two parallel squares of side-length r that are separated by an orthogonal distance of 1 unit. Let $\mathcal{V}(\mathcal{S}_1, \mathcal{S}_2) = \{v_2 - v_1 | v_1 \in \mathcal{S}_1, v_2 \in \mathcal{S}_2\}$, denoting the set of direction vectors corresponding to all possible directed lines from $\mathcal{S}_1$ to $\mathcal{S}_2$. Let $\Theta(\mathcal{R}_1, \mathcal{R}_2)$ denote the angle between two direction vectors $\mathcal{R}_1$ and $\mathcal{R}_2$. $\forall \mathcal{R}_1, \mathcal{R}_2 \in \mathcal{V}(\mathcal{S}_1, \mathcal{S}_2)$, $\Theta(\mathcal{R}_1, \mathcal{R}_2)$ is maximized when $\mathcal{R}_1$ and $\mathcal{R}_2$ correspond to the cross diagonals between $\mathcal{S}_1$ and $\mathcal{S}_2$.*

Proof: Let **u** denote a unit-length vector orthogonal to $\mathcal{S}_1$ and $\mathcal{S}_2$, and is directed from $\mathcal{S}_1$ towards $\mathcal{S}_2$. Let $\mathcal{O}_1$ denote the center of $\mathcal{S}_1$, and let $\mathcal{O}_2 = \mathcal{O}_1 + \mathbf{u}$. In addition, let $\mathcal{P}_1$ and $\mathcal{P}_2$ denote two-dimensional vectors defined as

$$\mathcal{P}_1 = v_1 - \mathcal{O}_1, \qquad \mathcal{P}_2 = v_2 - \mathcal{O}_2.$$

This means that $\mathcal{P}_1$ and $\mathcal{P}_2$ are vector representations of points on $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, with respect to the origins $\mathcal{O}_1$ and $\mathcal{O}_2$ respectively. Then,

$$\mathcal{V}(\mathcal{S}_1, \mathcal{S}_2) = \{\mathbf{u} + \mathcal{P}_2 - \mathcal{P}_1 | \mathcal{P}_1 \in \mathcal{S}_1, \mathcal{P}_2 \in \mathcal{S}_2\}$$

Note that $\mathcal{V}(\mathcal{S}_1, \mathcal{S}_2)$ is the set of direction vectors corresponding to all possible directed lines from $\mathcal{S}_1$ to $\mathcal{S}_2$. Now, consider the set of directed lines that pass through $\mathcal{O}_1$ in all possible directions represented by $\mathcal{V}(\mathcal{S}_1, \mathcal{S}_2)$. Let this set be denoted $\mathcal{V}(\mathcal{O}_1)$. Then the set of intercepts of the lines in $\mathcal{V}(\mathcal{O}_1)$ with the plane of $\mathcal{S}_2$ is

$$\begin{aligned} &= \{\mathcal{O}_1 + \mathbf{u} + \mathcal{P}_2 - \mathcal{P}_1 | \mathcal{P}_1 \in \mathcal{S}_1, \mathcal{P}_2 \in \mathcal{S}_2\} \\ &= \mathcal{O}_2 + (\mathcal{S}_1 \ominus \mathcal{S}_2) \end{aligned}$$

This corresponds to the Minkowski difference of two planar squares of side-length r, that is a square of side-length $2r$, and is coplanar with $\mathcal{S}_2$. Let this square be denoted as $\mathcal{S}_m$. Thus, $\mathcal{V}(\mathcal{O}_1)$ is the set of lines which are directed from $\mathcal{O}_1$ to any point on $\mathcal{S}_m$.

Since the set of direction vectors represented by $\mathcal{V}(\mathcal{S}_1, \mathcal{S}_2)$ is equal to the set of direction vectors represented by $\mathcal{V}(\mathcal{O}_1)$, maximum angle between any two vectors in $\mathcal{V}(\mathcal{S}_1, \mathcal{S}_2)$ is equal to the maximum angle between any two directed lines in $\mathcal{V}(\mathcal{O}_1)$. The angle between any two directed lines in $\mathcal{V}(\mathcal{O}_1)$ is maximized when their corresponding intercepts with $\mathcal{S}_m$ are furthest from each other. This corresponds to the case when the two intercepts are at opposite corners of a diagonal of $\mathcal{S}_m$. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ denote those directed lines as shown in Figure 4.5. The direction vectors of $\mathcal{R}_1$ and $\mathcal{R}_2$ correspond to those in $\mathcal{V}(\mathcal{S}_1, \mathcal{S}_2)$ that are associated with the cross diagonals between $\mathcal{S}_1$ and $\mathcal{S}_2$. □

Lemma 4.3.2 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be equal length parallel segments of length r lying on parallel planes that are separated by an orthogonal distance of 1 unit. Let $\Theta(\mathcal{S}_1, \mathcal{S}_2)$ denote*

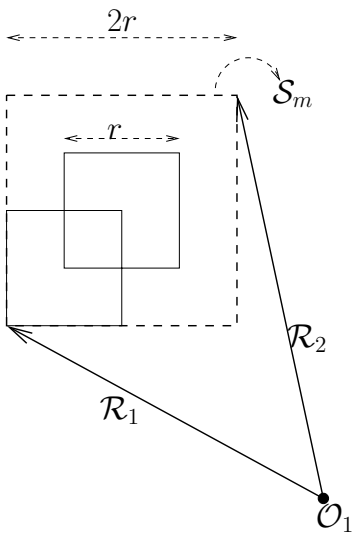


Figure 4.5: Minkowski difference of two planar squares of side-length r.

the angle between the cross diagonals that are directed from $\mathcal{S}_1$ to $\mathcal{S}_2$. Then

$$\Theta(\mathcal{S}_1, \mathcal{S}_2) \leq 2\arctan(r).$$

and the maximum angle is achieved when the two segments are aligned orthogonally opposite one another.

Proof: Consider the parallelogram generated by connecting the endpoints of the two segments as depicted in Figure 4.6(a). The height of such a parallelogram with base r is always greater than or equal to 1. Thus, The height of the shaded triangle is greater than or equal to 1/2. Now, consider an isosceles triangle $\mathcal{T}$ of height 1/2 and base r, and its circumcircle as shown Figure 4.6(b). Let Θ_1 denote the angle opposite the base. $\Theta_1 = 2\arctan(r)$. Any other triangle of height $\geq 1/2$ and the same base r as $\mathcal{T}$ has its apex vertex outside the circumcircle of $\mathcal{T}$. Thus, its angle opposite the base is smaller than Θ_1. For example, in Figure 4.6(b), $\Theta_1 > \Theta_2 > \Theta_3$. The shaded triangle of the parallegram could coincide with the isosceles triangle T only when the two segments are

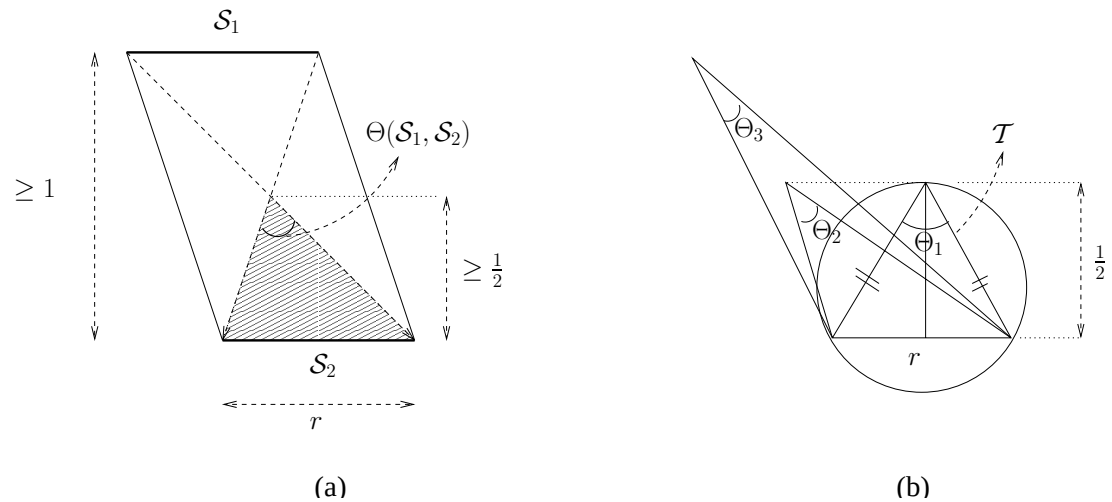


Figure 4.6: Maximum angle between the cross diagonals of two equal length parallel segments is achieved when the two segments are aligned orthogonally opposite one another.

aligned orthogonally opposite one another, in which case $\Theta(\mathcal{S}_1, \mathcal{S}_2) = \Theta_1 = 2\arctan(r)$. If the segments are not aligned, the height of the shaded triangle is greater than 1/2, thus $\Theta(\mathcal{S}_1, \mathcal{S}_2) < \Theta_1 = 2\arctan(r)$. □

4.4 Rendering and Interpolation Queries

Recall that our goal is to use interpolation between sampled output rays whenever things are sufficiently smooth. RI-tree can be used to perform a number of functions in rendering, including determining the first object that a ray hits, computing the reflection or refraction (exit) ray for nonsimple objects, and answering visibility queries, which are used for example to determine whether a point is visible to a light source or in a shadow.

Let us consider the interpolation of a given input ray R. We first map R to the associated point in the 4-dimensional directed line space and, depending on the dominant direction of this line, we find the leaf cell of the appropriate kd-tree through a standard

descent. Since the nodes of the tree are constructed only as needed, it is possible that R will reside in a leaf that is not marked as *final*. This means that this particular leaf has not completed its recursive subdivision. In this case, the leaf is subdivided recursively, along the path R would follow, until the termination condition is satisfied, and the final leaf containing R is now marked as *final*. (Other leaves generated by this process are not so marked.)

Given the final leaf cell containing R, the output attributes for R can now be interpolated. Interpolation proceeds in two steps. First we group the rays in groups of four, which we call the *directional groups*. Rays in the same group originate from the same corner point on the front plane, and pass through each of the four corners of the back plane. (For example, Figure 4.7 shows the rays that originate from the north-east corner of the front plane.) Within each directional group, bilinear interpolation with respect to the (u, v) coordinates is performed to compute intermediate output attributes. The outputs of these interpolations are then bilinearly interpolated with respect to the (s, t) coordinates to get the output attributes for R. Thus, this is essentially a quadrilinear interpolation.

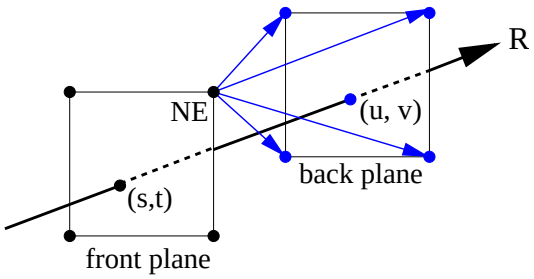


Figure 4.7: Sampled rays within a directional group.

4.5 Handling Discontinuities and Regions of High Curvature

Through the use of interpolation, we can greatly reduce the number of ray samples that would otherwise be needed to render a smooth surface. However, if the ray-output function f contains discontinuities, as may occur at the edges and the outer silhouettes of the object, then we will observe bleeding of colors across these edges. This could be remedied by building a deeper tree, which might involve sampling of rays up to pixel resolution in the discontinuity regions. This could result in unacceptably high memory requirements. Instead our approach will be to detect and classify discontinuity regions. In some cases we apply a more sophisticated interpolation. Otherwise we do not interpolate and instead simply revert to ray-tracing.

4.5.1 Grouping Samples in Equivalence Classes

Our objects are specified as a collection of smooth surfaces, referred to as *patches*. A patch could be a simple polygonal surface, or a more complex one such as a Bézier or NURBS surface. Each patch is assigned a *patch-identifier*. Associated with each sample ray, we store the patch-identifier of the first patch it hits. Since each ray sample knows which surface element it hits, it is possible to disallow any interpolation between different surfaces. It is often the case, however, that large smooth surfaces are composed of many smaller patches, which are joined together along edges so that first and second partial derivatives vary continuously across the edge. In such cases interpolation is allowed. Thus, the patches that share a common edge may or may not be joined with sufficiently high continuity to permit interpolation across the boundary. For example, in

Figure 4.8(a), we can interpolate between patches A and B, but not between patches C and D. We assume that the surfaces of the scene have been provided with this information, by partitioning patches into surface equivalence classes. Two adjacent patches in the same equivalence class are assumed to be connected continuously. Each *patch-identifier* is associated with a *class-identifier* denoting its equivalence class.

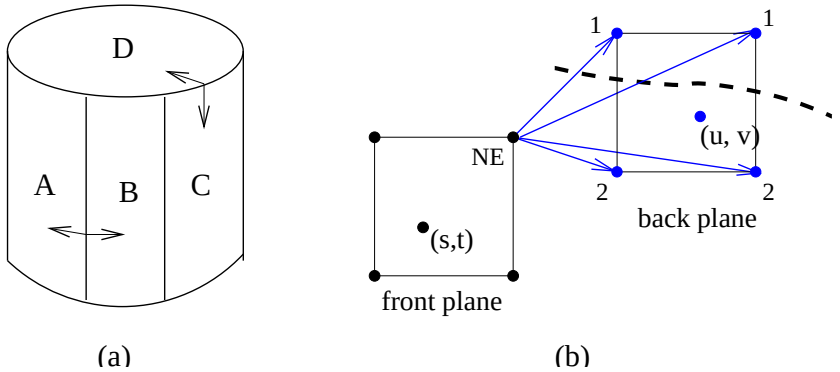


Figure 4.8: (a) Interpolation between A and B is allowed. Interpolation between C and D is not allowed. (b) Rays are grouped in two equivalence classes, implying a single discontinuity boundary.

If the patch-identifiers associated with the 16 corner ray samples of a final leaf are in the same equivalence class, we conclude that there is no discontinuities crossing the region surrounded by the 16 ray hits, and we apply the interpolation process described above. Requiring that all 16 patches arise from the same equivalence class can significantly limit the number of instances in which interpolation can be applied. After all, linear interpolation in 4-space can be performed with as few as 5 sample points. We assume that at lower levels of the tree, discontinuities crossing a cell will be of a simple nature and can be treated as a line segment. So, we find a model of the discontinuity and while avoiding interpolation across the discontinuity boundary, we still interpolate on

either side. If the patch-identifiers for the 16 corner samples of the leaf arise from more than two equivalence classes, we assume that multiple discontinuity boundaries cross the the region, and we revert to ray tracing. On the other hand, if exactly two equivalence classes are present, we decide that there is a single discontinuity boundary. Consider, for example Figure 4.8(b). To simplify the demonstration, we illustrate only one directional group. The projection of the discontinuity boundary on the (u, v) plane is shown for the NE directional group. Each corner on the (u, v) plane is labeled with the class-identifier of the patch hit by the ray passing through that corner. From the class-identifiers of the corner rays, we decide that there is a single boundary crossing the region surrounded by the four ray hits in this directional group.

Let us mention a few problematic cases that could arise. Since the knowledge of the *number of different equivalence classes overlapped* is based on the information obtained from the vertices, we might be mistaken. Consider the cases depicted in Figure 4.9(a) and (b). For example, in Figure 4.9(a), just by looking at the four corners, we would decide that the cell is overlapped by two equivalence classes and overlook the third one in between. Similarly for part (b). We do not detect these cases, and assume that they arise very rarely. We assume that if the patches are grouped in two equivalence classes, only the *good* cases depicted in Figure 4.9(c) and (d) could arise.

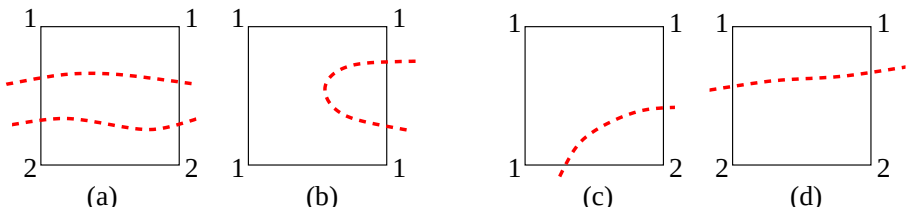


Figure 4.9: (a)-(b) Bad cases (c)-(d) Good cases

In the case where exactly two equivalence classes are present, we perform an intersection test to determine which patch the query ray hits. Let p_r denote this patch. This intersection test is not as expensive as a general tracing of the ray, since typically only a few patches are involved, and only the first level intersections of a ray-tracing procedure is computed (that is, no reflection rays or light rays need be traced). Among the 16 corner ray samples, only the ones that hit a patch in the same equivalence class as p_r are *usable* as interpolants. These are the ray samples hitting the same side of a discontinuity boundary as the query ray.

Since at least three interpolants are required to in each directional group interpolation and in the final interpolation of intermediate results, some cells cannot be used for interpolation due to unusable candidate rays. If we determine that there is a sufficient number of *usable* ray samples, we then interpolate the ray. Otherwise, we use ray-tracing. The algorithm given in Figure 4.10 summarizes the interpolation method. In the algorithm $f^*(R)$ denotes the final interpolated output attributes for query ray R, that is $f^*(R)$ is an approximation of $f(R)$.

For the three-interpolant cases, if the point for which we attempt to interpolate lies outside the triangular region formed by the points corresponding to the usable candidate rays, this is not an interpolation anymore—it becomes an extrapolation. Since extrapolated values are less reliable, the user is granted the option of tuning the extrapolation. If the point to be extrapolated is farther from the triangular region—in terms of its barycentric coordinates—than a given threshold, extrapolation is disabled and the cell cannot be used for interpolation.

```
determine the patch hit by the query ray R;
NumberOfUsableGroups = 0;
for each of the four directional groups do
    if (number of usable rays ≥ 3) then
        NumberOfUsableGroups++;
        compute intermediate output attributes
            by interpolating usable rays;
if (NumberOfUsableGroups ≥ 3) then
    compute f*(R) using the intermediate
        output attributes from successful directional groups;
    return f*(R);
else
    return failure; // Trace the ray
```

Figure 4.10: Interpolation algorithm

4.5.2 Angular Thresholds

Even if interpolation is allowed by the above criterion, it is still possible that interpolation may be inappropriate because the surface has high curvature, resulting in very different output rays for nearby input rays. High variations in the output ray (i.e. normal or the exit ray), signal a discontinuous region. As a measure to determine the distance between two output rays, we use the angular distance between their directional vectors. If any pairwise distance between the output rays corresponding to the usable interpolants is greater than a given *angular threshold*, then interpolation is not performed.

4.6 Experimental Results

The RI-tree is based on a number of parameters, which directly influence the algorithm's accuracy and the size and depth of the tree, and indirectly influences the running time. We have implemented the data structure and have run a number of experiments to test its performance as a function of a number of these parameters. We have performed our comparisons in the context of two applications.

Ray-tracing: This has been described in the previous sections. We are given a scene consisting of objects that are either simple, reflective or transparent and a number of light sources. The output is a rendering of the scene from one or more viewpoints.

Volume Visualization: This application is motivated from the medical application of modeling the amount of radiation absorbed in human tissue [dKL02]. We wish to visualize the absorption of radiation through a set of nonintersecting objects in 3-space. In the medical application these objects may be models of human organs, bones, and tumors. For visualization purposes, we treat these shapes as if they are transparent (but are not refractive). If we imagine illuminating such a scene by x-rays, then the intensity of a pixel in the image is inversely proportional to the length of its intersection with the various objects of the scene. For each object stored as an RI-tree, the geometric attribute associated with each ray is this intersection length.

4.6.1 Test Inputs

We have generated a number of input scenes including different types of objects. As mentioned earlier, for each object in a scene we may choose to represent it in the traditional

method or to use our data structure. Our choice of input sets has been influenced by the fact that the RI-tree is most beneficial for high-resolution renderings of smooth objects, especially those that are reflective or transparent. We know of no appropriate benchmark data sets satisfying these requirements, and so we have generated our own data sets.

Bézier Surface: This surface is used to demonstrate the results of interpolation algorithm for smooth reflective objects. It is a reflective surface consisting of 100 Bézier patches, joined with C^2 continuity at the edges. The surface is placed within a large sphere, which has been given a pseudo-random procedural texture [EMP+98]. Experiments run with the Bézier surface have been averaged over renderings of the surface from 3 different viewpoints. Figure 4.12(a) shows the Bézier surface from one viewpoint. We rendered images of size 600×600 without antialiasing. (That is, only one ray is shot per pixel.)

Random volumes: We ran another set of experiments on randomly generated refractive, nonintersecting, convex Bézier objects. In order to generate nonintersecting objects, a given region is recursively subdivided into a given number of nonintersecting cells by randomly generated axis-aligned hyperplanes, and a convex object is generated within each such cell. Each object is generated by first generating a random convex planar polyline that defines the silhouette of right half of the object. The vertices of the polyline constitute the control points for a random number (n) of Bézier curves, ranging from 5 to 16. Then a surface of revolution is generated, giving rise to $4n$ Bézier surface patches. The volumes are used both for the ray-tracing and the volume visualization experiments. For ray-tracing we

rendered anti-aliased images of size 300×300 (with 9 rays shot per pixel). For volume visualization we rendered 600×600 images without antialiasing. Results are averaged over three different random scenes containing 8, 6, and 5 volumes respectively. Figure 4.13 shows a scene of refractive volumes.

Tomatoes: This is a realistic scene used to demonstrate the performance and quality of our algorithm for real scenes. The scene consists of a number of tomatoes, modeled as spheres, placed within a reflective bowl, modeled using Bézier surfaces. This is covered by a reflective and transparent but non-refractive plastic wrap (the same Bézier surface described above). There is a Bézier surface tomato next to the bowl, and they are both placed on a reflective table within a large sphere. The wrap reflects the procedurally textured sphere. The scene is shown in Figure 4.18.

4.6.2 Metrics

We measured the *speedup* and *actual error* committed as a function of four different parameters. Speedup is defined both in terms of number of floating point operations, or *FLOPs*, and CPU-time. FLOP speedup is the ratio of the number of FLOPs performed by traditional ray-tracing to the number of FLOPs used by our algorithm to render the same scene. Similarly, CPU speedup is the ratio of CPU-times. Note that FLOPs and CPU-times for our algorithm include both the sampling and interpolation time. FLOP counts are machine independent, but they tend to underestimate the time spent in data structure access. However, our experience has shown that this access time is not a dominant component of the overall running time.

The actual error committed in a ray-tracing application is measured as the average L_2 distance between the RGB values of corresponding pixels in a ray-traced image and the interpolated image. RGB value is a 3-dimensional vector with values normalized to the range $[0, 1]$. Thus the maximum possible error is $\sqrt{3}$. The error in a volume visualization application is measured as the average distance between the actual length attribute and the corresponding interpolated length attribute.

4.6.3 Varying the Parameters

Varying Distance Threshold: Recall that the distance threshold, described in Section 4.3.3, is used to determine whether an approximate output ray and the corresponding actual output ray are close enough (in terms of L_2 distance) to terminate a subdivision process. We varied the distance threshold from 0.01 to 0.25 while the other parameters are fixed. The results for the Bézier surface scenes are shown in Figure 4.11. As expected, the actual error decreases as the threshold is lowered, due to denser sampling. But, the overhead of more sample computations reduces the speedup. However, even for low thresholds where the image quality is high, the CPU-speedup is greater than 2 and the FLOP-speedup is greater than 3. These speedups can be quite significant for ray-tracing, where a single frame can take a long time to render.

Figure 4.12 (b) and (c) demonstrate how the variation in error reflects the changes in the quality of the rendered image. Notice the blockiness in part (c) when the data structure is not subdivided as densely as in part (b).

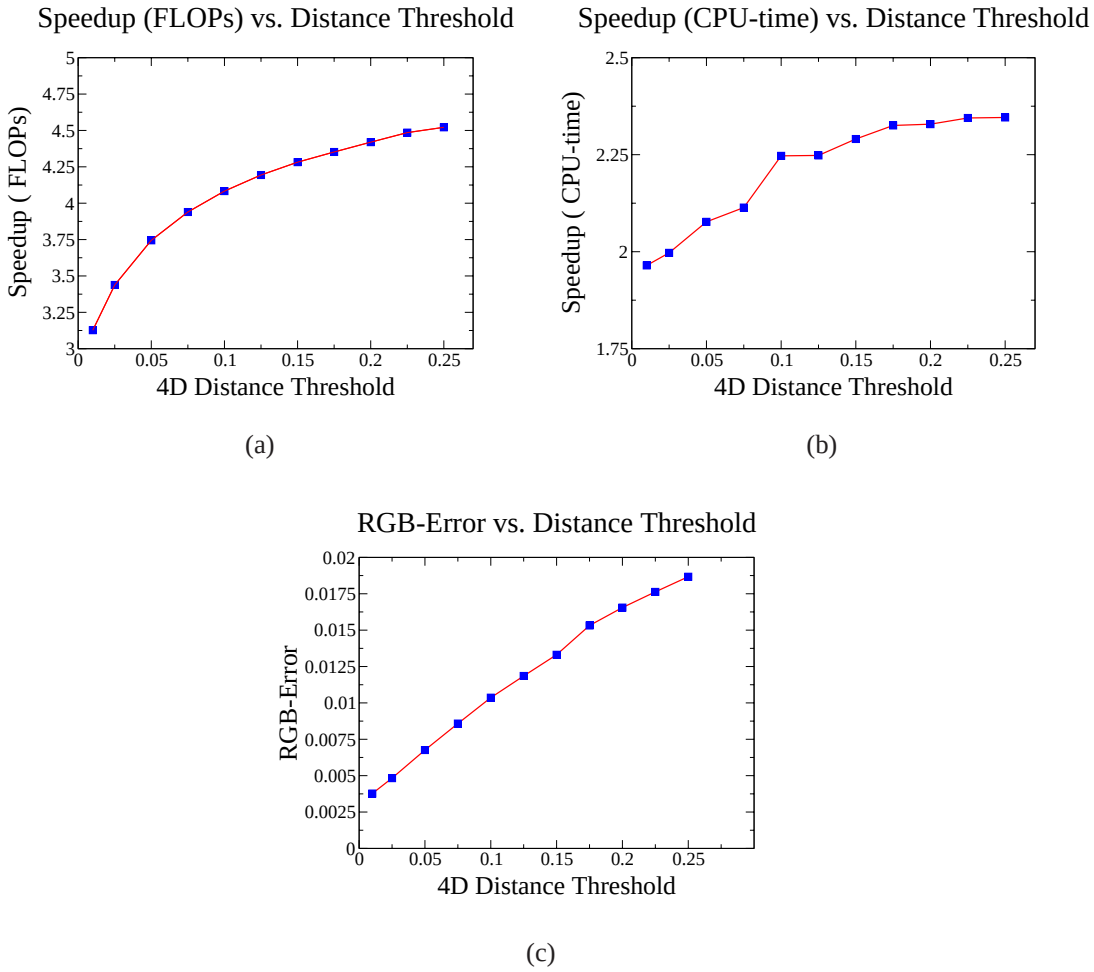
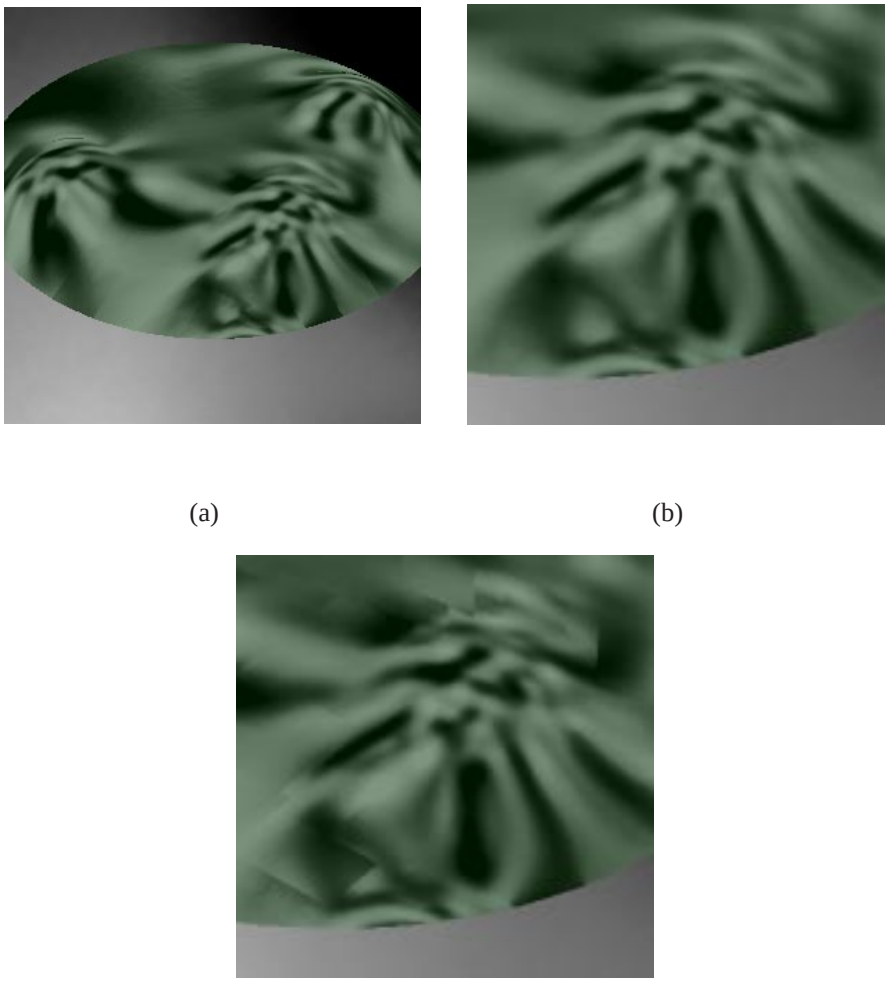


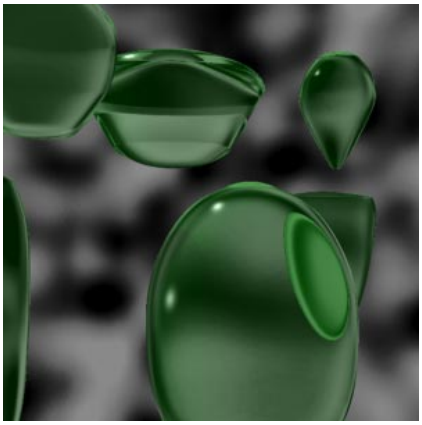
Figure 4.11: Varying the distance threshold. (Angular threshold = $30°$, maximum tree depth = 28, 600×600 image, non-antialiased). Note that the y-axis does not always start at 0.



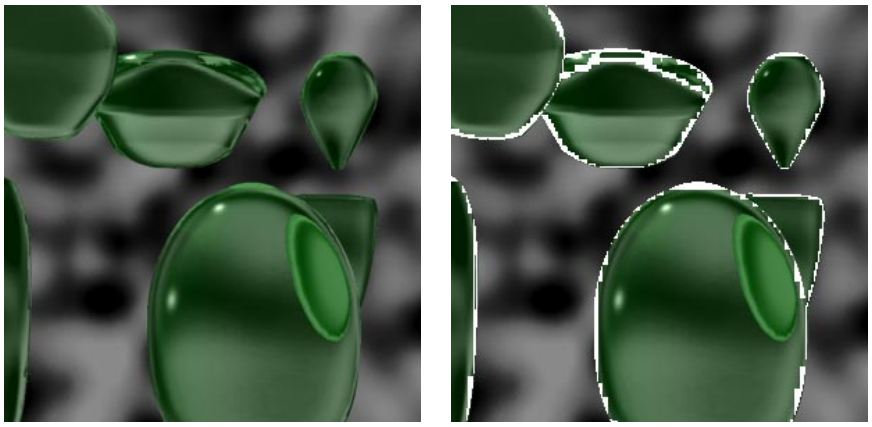
(a) (b)

(c)

Figure 4.12: (a) Ray-traced image, (b) Lower right part of interpolated image (distance threshold=0.01), error = 0.00377, (c) Lower right part of interpolated image (distance threshold=0.15), error = 0.01331.



(a)

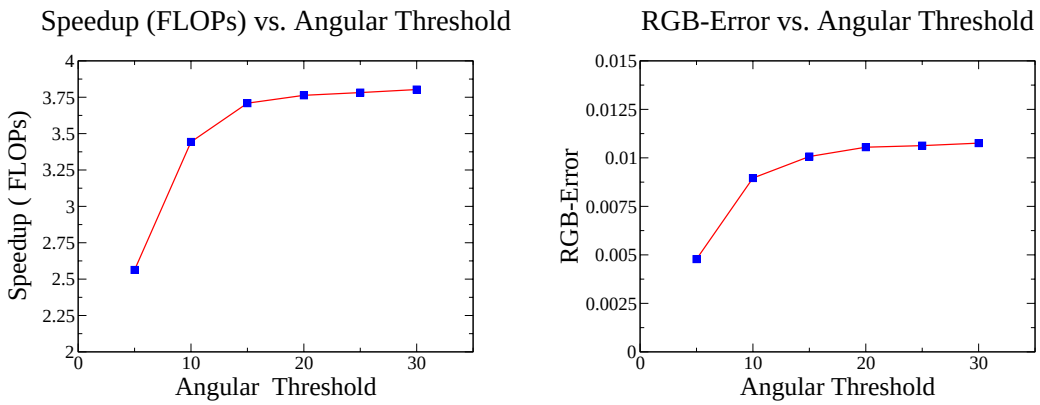


(b)

Figure 4.13: (a) Ray-traced image, (b) Interpolated image (distance threshold=0.05) and the corresponding color-coded image where white regions indicate pixels that were ray-traced.

Varying Angular Threshold: The angular threshold, described in Section 4.5, is applied to each query to determine whether the surface curvature variation is too high to apply interpolation. We investigated the speedup and error as a function of the angular threshold over the renderings of three different random volume scenes. The angular threshold is varied from 5° to 30°. The results are shown in Figure 4.14.

For lower thresholds, fewer rays could be interpolated due to distant interpolants, and those rays are traced instead. In this case, the actual error committed is smaller but at the expense of lower speedups. However, the speedups are still acceptable even for low thresholds.

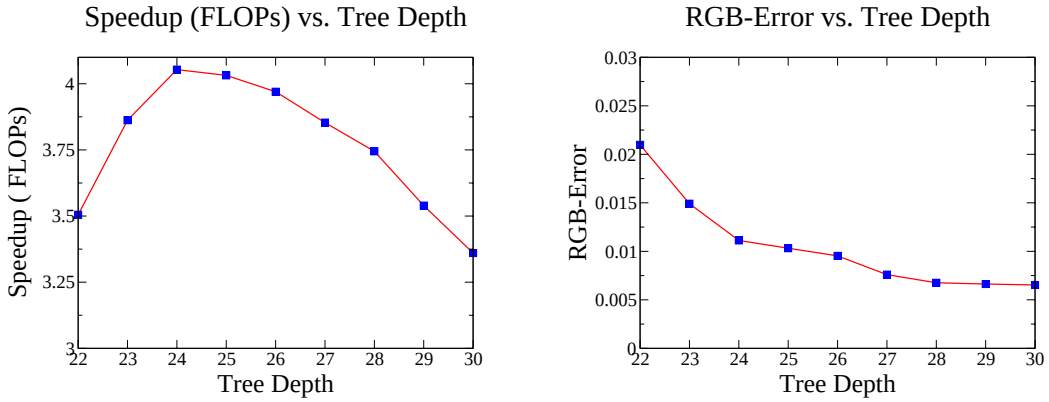


(a) (b)

Figure 4.14: Varying angular threshold (distance threshold=0.25, maximum depth=28, 300×300, antialiased).

Varying Maximum Tree Depth: Recall that the maximum tree depth, described in Section 4.3.3, is imposed to avoid excessive tree depth near discontinuity boundaries. We considered maximum depths ranging from 22 to 30—corresponding to angular similarity ranges of 11° down to 2.7° (Because this is a kd-tree in 4-space, four levels of descent are

generally required to halve the the diameter of a cell). The results for the Bézier surface scenes are shown in Figure 4.15. The angular threshold is fixed at $30°$, and the distance threshold is fixed at 0.05.



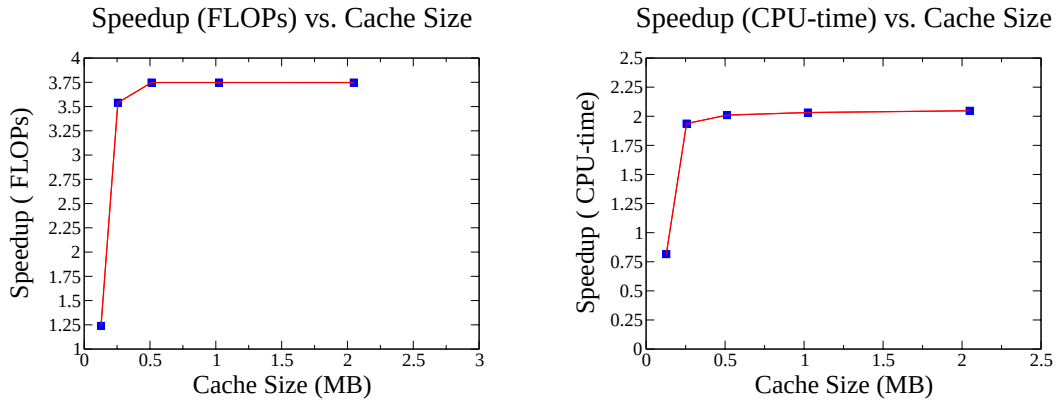
(a) (b)

Figure 4.15: Varying tree depth (distance threshold=0.05, angular threshold=30, 600×600, non-antialiased).

As the tree is allowed to grow up to a higher depth, rays are sampled with increasing density in the regions where the geometric attributes have greater variation, and thus, error committed by the interpolation algorithm decreases with higher depths. The speedup graph shows a more interesting behavior. Up to a certain depth, the speedup increases with depth. Speedups are poor for very low-depth trees, since many of the interpolants cannot pass the angular threshold test, and so many rays need to be traced rather than interpolated. However, the speedup decreases with very large depth values, since the overhead caused by denser sampling starts to dominate. It seems that a wise choice of depth would be a value that results in both a lower error, and reasonable speedup. For example for the given graph, depth around 28 would be a good choice for this image. (This corresponds to angular similarity constraint of roughly $3.8°$.) However, peak performance

depends on a number of parameters that are particular to the ray tracing application, such as the expected cost of a single ray shoot. In addition, Tables 4.1–4.3 shows the required memory when depth is varied. When the tree is unnecessarily deep, not only does the speedup decrease, but space requirements increase as well.

Varying Cache Size: As mentioned earlier, the RI-tree functions as an LRU cache. If an upper limit for the available memory—the cache size—is specified, the least recently used paths are pruned based on time stamps set whenever a path is accessed. Excessively small cache sizes can result in frequent regeneration of the same cells. For the Bézier surface scene, we have varied the cache size from 0.128 to 2.048 megabytes (MB). The resulting speedup graph is shown in Figure 4.16.



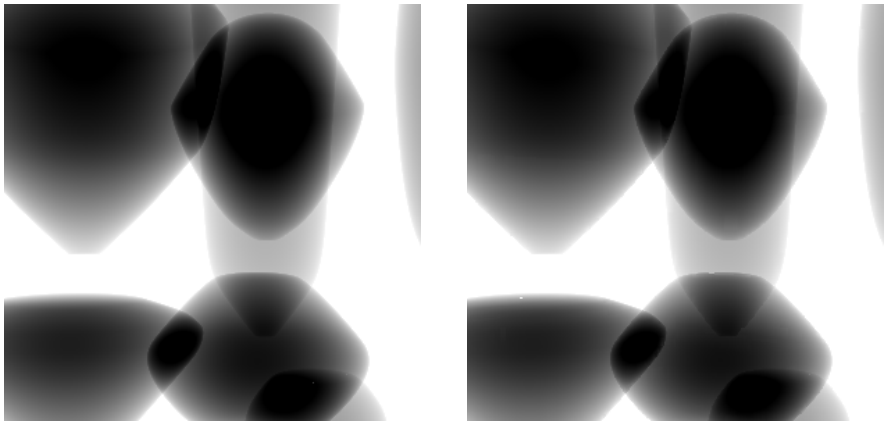
(a) (b)

Figure 4.16: Varying cache size (distance threshold = 0.05, angular threshold = 30, maximum tree depth = 28, 600×600 image, non-antialiased).

Notice that we used small cache sizes to demonstrate the sudden increase in speedup as the cache size approaches a reasonable value. Normally, we set the cache size to 100MB, which is high enough to handle bigger scenes with many data structures. There

are other parameters involved in garbage collection, such as the percentage of the cache being pruned. In these experiments, each garbage collection prunes 70% of the cache.

Volume Visualization Experiments: We have tested the algorithm for the volume visualization application using the same random volumes we used for refractive objects. Images are 600×600 and not antialiased. Results of our sample runs are shown in Table 4.1-4.3. The FLOP speedup varies from 2.817 to 3.549, and CPU speedup varies from 2.388 to 2.814. For higher resolutions, or anti-aliased images the speedups could be higher. The error could be as low as 0.008 for low distance thresholds, and is still at a reasonable value for higher thresholds. Figure 4.17 shows the actual image, and the interpolated image visualizing one of the random volume scenes. All objects have 0.5 opacity, and all have solid gray colors.



(a) (b)

Figure 4.17: (a) Ray-traced image, (b) Interpolated image (distance threshold=0.25).

Test Input	Dist Thresh	Speedup (FLOP)	Speedup(CPU-time)	Error	Memory (MB)
Bézier Surface	0.010	3.12704	1.96466	0.00377	2.925
	0.025	3.43796	1.99712	0.00483	2.371
	0.050	3.74473	2.07705	0.00676	1.931
	0.075	3.93950	2.11372	0.00858	1.699
	0.100	4.08325	2.24707	0.0103	1. 549
	0.125	4.19358	2.24816	0.01185	1.442
	0.150	4.28194	2.29041	0.01331	1.361
	0.175	4.35214	2.32532	0.01532	1.301
	0.200	4.41940	2.32863	0.01655	1.253
	0.225	4.48503	2.34465	0.01763	1.212
	0.250	4.52146	2.34591	0.01867	1.185
Random Volumes	0.010	3.12173	2.63532	0.00627	19.252
(ray-tracing)	0.025	3.26194	2.64317	0.00645	17.518
	0.050	3.40527	2.71941	0.00679	15.799
	0.075	3.49906	2.76194	0.00722	14.765
	0.100	3.56870	2.79244	0.00780	14.088
	0.125	3.62689	2.84409	0.00853	13.603
	0.150	3.67422	2.88046	0.00890	13.183
	0.175	3.70776	2.89190	0.00945	12.875
	0.200	3.74041	2.92770	0.00989	12.583
	0.225	3.77341	2.94416	0.01048	12.331
	0.250	3.80292	2.91917	0.01076	12.094
Random Volumes	0.050	2.95084	2.42804	0.00850	11.773
(volume	0.150	3.31043	2.67274	0.01179	9.503
visualization)	0.250	3.54958	2.81416	0.01488	8.344

Table 4.1: Varying the distance threshold: Speedup and actual error on Bézier Surface and Random Volumes (ray-tracing and volume visualization).

Test Input	Tree Depth	Speedup (FLOP)	Speedup(CPU-time)	Error	Memory (MB)
Bézier Surface	22	3.50486	2.05642	0.02098	0.565
	23	3.86223	2.14654	0.01491	0.706
	24	4.05344	2.21946	0.01112	0.881
	25	4.03178	2.17521	0.01032	1.084
	26	3.97010	2.15906	0.00953	1.318
	27	3.85335	2.05680	0.00760	1.603
	28	3.74473	2.07705	0.00676	1.931
	29	3.53944	2.04811	0.00663	2.265
	30	3.36016	1.97434	0.00653	2.629
Random Volumes	22	3.10450	2.56453	0.01859	3.729
(ray-tracing)	23	3.41967	2.63197	0.01708	4.431
	24	3.70909	2.74675	0.01526	5.441
	25	3.85445	2.90357	0.01449	6.560
	26	3.85108	2.93271	0.01305	7.989
	27	3.84435	2.87188	0.01187	9.660
	28	3.80292	2.91917	0.01076	12.094
	29	3.56893	2.79997	0.01026	14.361
	30	3.34045	2.73197	0.00987	17.413

Table 4.2: Varying the tree depth: Speedup and actual error on Bézier Surface and Random Volumes (ray-tracing and volume visualization).

Input Scene	Ang Thresh	Speedup (FLOP)	Speedup(CPU-time)	Error
Bézier Surface	5	2.68103	1.68226	0.00424
	10	3.51840	2.01129	0.00591
	15	3.68553	2.11734	0.00663
	20	3.72731	2.12195	0.00673
	25	3.74471	2.11754	0.00676
	30	3.74473	2.07705	0.00676
Random Volumes	5	2.56317	2.15410	0.00478
(ray-tracing)	10	3.44274	2.67800	0.00896
	15	3.70928	2.83973	0.01007
	20	3.76320	2.88208	0.01055
	25	3.78206	2.89311	0.01063
	30	3.80292	2.91917	0.01076
Random Volumes	10	2.81703	2.38833	0.01047
(volume	15	3.21517	2.62693	0.01340
visualization)	20	3.40653	2.73348	0.01411
	30	3.54958	2.81416	0.01488

Table 4.3: Varying the angular threshold: Speedup and actual error on Bézier Surface and Random Volumes (ray-tracing and volume visualization).

4.6.4 Results for the Tomatoes Scene

Finally, we have tested our algorithm on the tomatoes scene generating an image of size 1200×900, non-antialiased. Table 4.4 shows sample results for the tomato scene and Figure 4.18 shows the corresponding images. Figure 4.18(a) shows the ray-traced image. Part (b) shows the interpolated image, and a corresponding color-coded image in which the white regions denote the pixels that were traced rather than interpolated. Part (c) shows the interpolated image generated with lower thresholds and the corresponding color-coded image. Notice that the artifacts in part (b) are corrected in part (c).

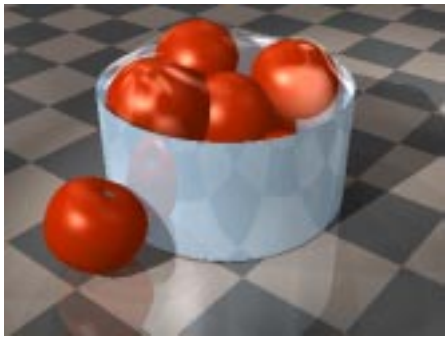
Dist.Thr.	Ang.Thr.	TreeDepth	Speedup(FLOP)	Speedup(time)	Error	Memory
0.25	30	28	2.65	1.89	0.00482	34 MB
0.05	10	28	2.40	1.75	0.00190	47 MB

Table 4.4: Sample results for tomatoes scene (1200×900 non-antialiased).

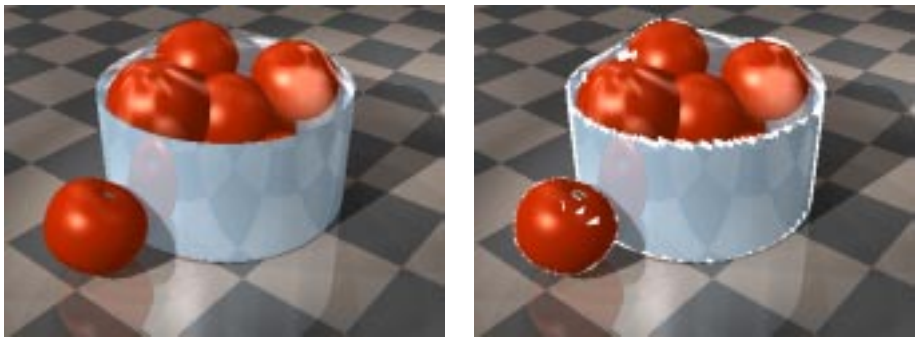
Note that the closest objects along the eye rays are correctly determined by interpolation, as are the reflection rays from the wrap and the bowl, and the shadows. The sky (procedural texture of the enclosing sphere) is reflected on the wrap. As expected, for lower threshold values we can get a very high quality image and still achieve speedups of 2 or higher. If quality is not the main objective, we can get approximate images at higher speedups.

4.6.5 Radiance versus Ray Interpolation

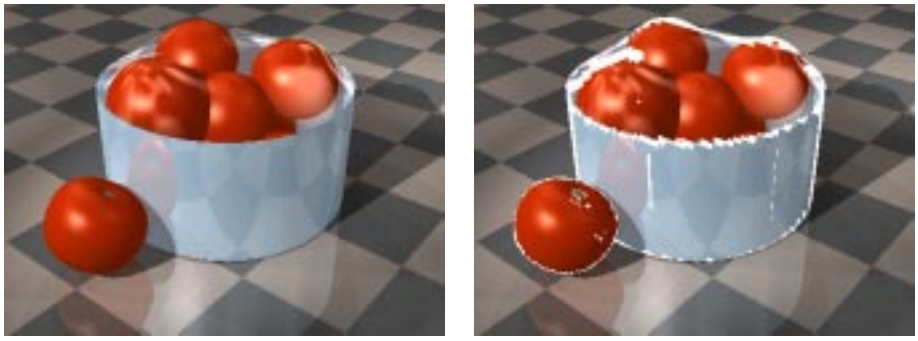
As we have mentioned before, in a ray-tracing application, we prefer interpolating geometric attributes such as normal vectors and exit rays, rather than interpolating radiance.



(a)



(b)



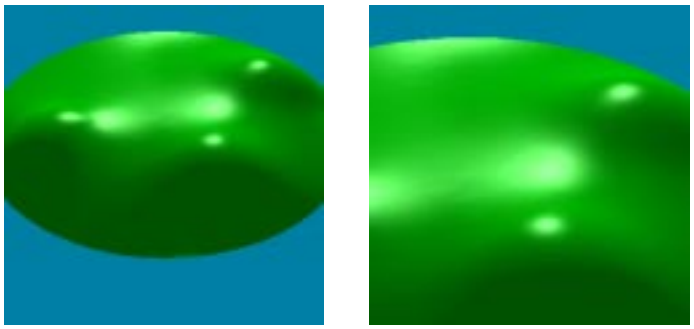
(c)

Figure 4.18: (a) Ray-traced image, (b) Interpolated image (dist. thr.=0.25, ang. thr.=30) and corresponding color-coded image, white areas show the ray-traced regions, (c) Interpolated image (dist. thr.=0.05, ang. thr.=10).

The main advantage of our approach is that it allows representation of an object independent from the other objects in the environment and/or illumination. For example, in methods based on radiance interpolation, if the illumination changes (a light source is removed or its intensity increased), many samples have to be recomputed even if the viewpoint remains the same, whereas our methods do not require altering the RI-tree of any object. Or, consider the case of reflective and refractive objects,for example, if an object A is reflected on object B, and if A moves slightly, the RI-tree of B will also be affected if radiance interpolants are used.

Another reason for sampling and interpolating normals and intersection points instead of radiance is related to our focus on reflective and refractive objects. Recall that for such objects, we also associate an exit ray with each sample. In order to build a unified framework, it makes more sense to store normal vectors with exit rays, since the variation in exit ray is more closely related to variation in normal vectors than radiance.

The comparison of radiance versus normal interpolation is somewhat analogous to the difference in the interpolation methods used in Gouraud and Phong shading. Consider the Bézier surface scene, where the surface is simple (neither reflective nor refractive), but the specularity is high. Figure 4.19 shows the ray-traced imageof a part of the surface illuminated by 2 light sources. If we choose to interpolate radiance, similar to Gouraud shading, we may have to collect samples much more densely, in the regions near specular highlights in order to achieve greater fidelity, since in those regions radiance varies rapidly. However, when we choose to interpolate normals and then compute shading with respect to the interpolated normal, dense sampling is only required in areas where the surface is not smooth enough, regardless of the radiance.



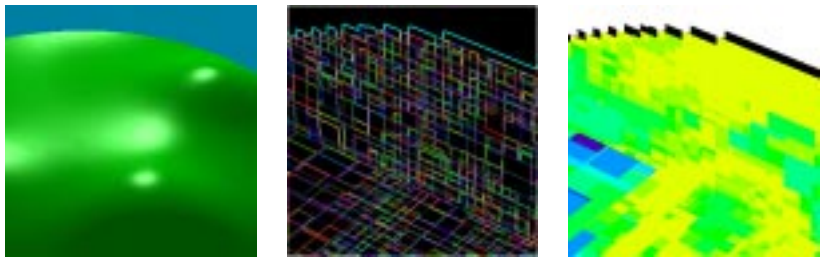
(a) (b)

Figure 4.19: (a) Ray-traced simple Bézier surface (b) Upper right part zoomed.

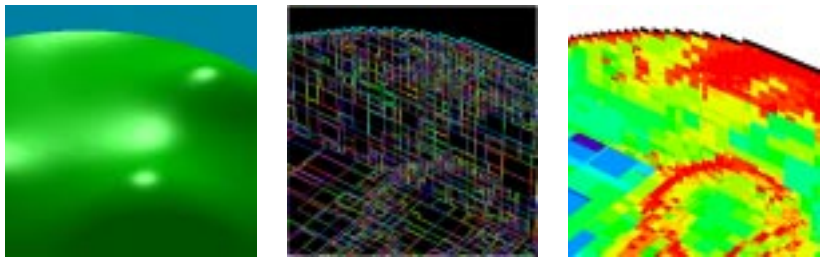
Consider Figure 4.20. In part (a), the leftmost image shows the surface rendered by using RI-tree, where normals are interpolated. The maximum depth allowed is 28, and this is enough to obtain a high quality approximation to the ray-traced image in Figure 4.19. The middle image is a visualization of the RI-tree cells. The rightmost image color codes the depth of the leaf cell used to interpolate the associated pixel. The depth-color scale is given in part (d). Part (b) gives the corresponding images when the depth is allowed to grow up to 32, some cells are refined more around regions of high curvature.

Part (c) shows the images when the image is rendered by radiance interpolation such that the tree is refined according to the variance in radiance. To generate an image of comparable quality to (a) and (b), the maximum depth should be set to at least 32. (For values lower than 32, the highlights are noticeably distorted.) Number of nodes are 7.4K, 13K, and 25K for parts (a), (b) and (c) respectively. Note that Part (a) is a high quality approximation to the ray-traced version, and to achieve the same quality by radiance interpolation, 3 times more nodes—and, 3 times more samples—should be generated.

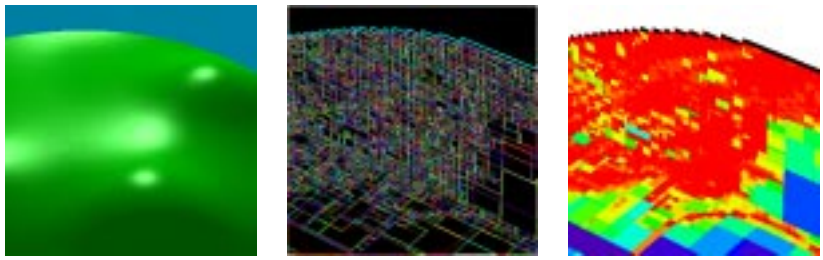
Depending on the scene and the relative costs of intersection computations and



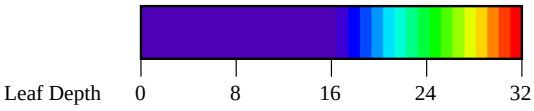
(a)



(b)



(c)



(d)

Figure 4.20: (a) Normal interpolation, max. depth = 28, no. of nodes = 7.4K. (b) Normal interpolation, max. depth =32, no. of nodes = 13K. (c) Radiance interpolation, max. depth = 32, no. of nodes = 25K. (d) Depth color scale.

shading, either radiance interpolation or ray interpolation may be better than the other in terms of performance. As described above in the case of a specular object, radiance interpolants are usually more expensive in terms of sampling cost—they generate more samples, and each sample is more costly since they also compute shading. On the other hand, when rendering, ray interpolation methods interpolate at least two vectors (normal and intersection point), and compute shading for each ray, whereas radiance interpolation methods interpolate just radiance but compute intersections with objects for each ray.

4.6.6 Animations

In an animation sequence, since many samples will be reused by subsequent frames, we expect that the performance gain after the first frame would be higher. We have tested the algorithm on the following two animation sequences. In both cases, the degree of speedup for the first frame was considerably lower than subsequent frames, since the data structure is built from scratch for the first frame, and subsequent frames can reuse some or all of the existing structure.

Light animation: In the first sequence, we use our tomatoes scene, illuminated by only one spotlight. During the animation, the viewpoint is fixed, but the spotlight is swinging, thus the illumination in the scene is different in each frame. Since, the viewpoint is fixed as well as the location of the light, no more nodes/samples are generated after the first frame, and so there is no sampling cost. The performance for the first three frames are given in Table 4.5. The speed-ups are roughly constant after the first frame.

Dist. Thresh.	Ang. Thresh.	Tree Depth	Speedup (FLOP)			Speedup (CPU-time)		
			Fr#1	Fr#2	Fr#3	Fr#1	Fr#2	Fr#3
0.25	30	28	2.35	2.98	2.98	1.83	2.23	2.21
0.05	10	28	2.02	2.67	2.67	1.67	2.04	2.09

Table 4.5: Sample results for light animation (1200×900 non-antialiased).

Viewpoint animation: In the second sequence, we use the original tomatoes scene illuminated by 9 light sources. In each frame, the viewpoint is rotated around the cup by 1°. And so, even though many samples are reused, some new samples are also generated in each frame. The performance for the first three frames are given in Table 4.6. The speed-ups are roughly constant after the first frame.

Dist. Thresh.	Ang. Thresh.	Tree Depth	Speedup (FLOP)			Speedup (CPU-time)		
			Fr#1	Fr#2	Fr#3	Fr#1	Fr#2	Fr#3
0.25	30	28	2.65	2.97	2.98	1.89	2.04	2.02
0.05	10	28	2.40	2.83	2.83	1.75	1.99	1.98

Table 4.6: Sample results for viewpoint animation (1200×900 non-antialiased).

4.7 Conclusions

In this chapter, we introduced the RI-tree data structure and illustrated its use in the context of efficient ray-tracing. By our approach of sampling and interpolating geometric attributes rather than radiance, we decouple the local geometry of the object from the rest of the scene geometry and illumination. Hence, we do not need to alter the RI-tree of a

specific object, if any other object moves, or lighting conditions in the scene changes. (Except when the location of a light source changes, since it may cause additional sampling in RI-trees of some objects. Recall that determining visibility of a light source involves checking intersections with possibly occluding objects). We only need to sample additional rays if the viewpoint or the viewing direction changes, since these may cause new parts of an object become visible.

The RI-tree is most useful for rendering smooth objects that are reflective or transparent, for rendering animations when the viewpoint varies smoothly or when the illumination varies from frame to frame, and for generating high resolution images and/or antialiased images generated by supersampling in which multiple rays are shot for each pixel of the image.

We demonstrated the performance-quality tradeoff by experimenting with the few parameters that control the quality of approximation. For our test scenes consisting mostly of reflective or refractive objects, we presented experimental results that our algorithm speeds up ray-tracing at least by a factor of two in terms of CPU-time, and by at least by a factor of three in terms of FLOPs. The speedups are higher if an input ray goes through multiple levels of reflections/refractions before escaping the object, since our algorithm performs a fixed set of interpolations independent of the number of levels of reflections/refractions. We also presented performance results for animations where the viewpoint changes and for animations where the lighting changes. Speedups are higher after the the first frame, since some samples generated for a frame are being reused for subsequent frames.

The performance gain is achieved at the potential expense of quality. However,

our system detects and deals with the object boundaries and other strong discontinuities where the artifacts are more likely to be noticed. A number of heuristics are introduced to allow more interpolation around discontinuity boundaries.

One of the disadvantages of the RI-tree is the need for these heuristics for detecting and handling discontinuities. This is the result in part of the fact that each node of the 4-dimensional kd-tree has 16 vertices. Unless the nodes are subdivided to a very fine level, it is quite likely that at least one of these vertices will lie on the wrong side of discontinuity boundary. In addition, the kd-tree subdivision is not a cell complex, which implies that there may be cracks, which also result in problems with continuity. In Chapter 6 we introduce an approach based on a hierarchical decomposition into 4-dimensional simplices. We will see that this method eliminates the need for these heuristics.

Chapter 5

Simplex Decomposition Tree: A Pointerless Representation

5.1 Introduction

In the previous chapter, we have suggested that an efficient approach to answering multidimensional interpolation queries is through data structures based on hierarchical subdivision of space and we have used a kd-tree based subdivision. However, a significant problem with both kd-trees and quadtrees is that the resulting subdivision is not generally a cell complex. Intuitively, a *cell complex* is a subdivision in which pairs of neighboring cells meet along a common face. A cell complex whose faces are simplices is called a *simplicial complex*. (See [Mun75] for definitions.) A subdivision which is a cell complex is also referred to as *compatible*. (Some authors also prefer *conforming* or *consistent*.) We will use the term *compatible* or *cell/simplicial complex* interchangeably in the rest of this thesis. When the subdivision is not compatible, *cracks* occur along faces of the subdivision (see Figure 5.1(a)), which in turn present problems when using the mesh for interpolation.

It is possible to further subdivide a kd-tree/quadtree subdivision to produce a sim-

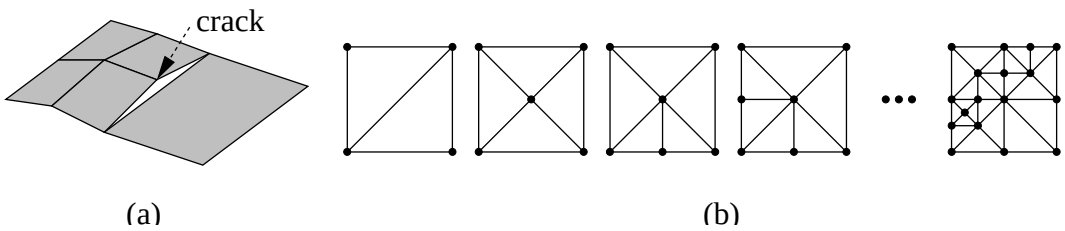


Figure 5.1: (a) A crack (b) A hierarchical simplicial mesh in the plane.

plicial complex [SS92, Paj02], by first *restricting* the quadtree with additional sampling in such a way that two leaf cells in the tree differ at most by one level, and then triangulating the quadtree according to a predefined set of patterns depending on the number of neighboring cells that have been subdivided. However, this approach does not scale well with dimension due to the exponential increase in the number of vertices and explosion of cases that need to be considered. In addition, operations such as point location, and determination of barycentric coordinates whose efficiency are crucial in our application becomes more complicated and requires more computational effort. Moreover, this approach usually creates more vertices compared to maintaining the subdivision as a triangulation (we describe this below), which is more costly from our perspective, since each vertex is sampled on-demand, and the sampling cost is included in the total cost of rendering a frame.

Instead, we use an attractive and simpler alternative: *hierarchical regular simplicial mesh*. This is a d-dimensional generalization of the concept of hierarchical regular triangulation in the plane [EKT01] or in 3-space [LDS01]. Each element of such a mesh is a d-simplex, that is, the convex hull of $d + 1$ affinely independent points [Ede87]. The vertices of the mesh correspond to the vertices of a d-dimensional grid. The mesh is generated by a process of repeated bisection applied to a hypercube initially subdivided into

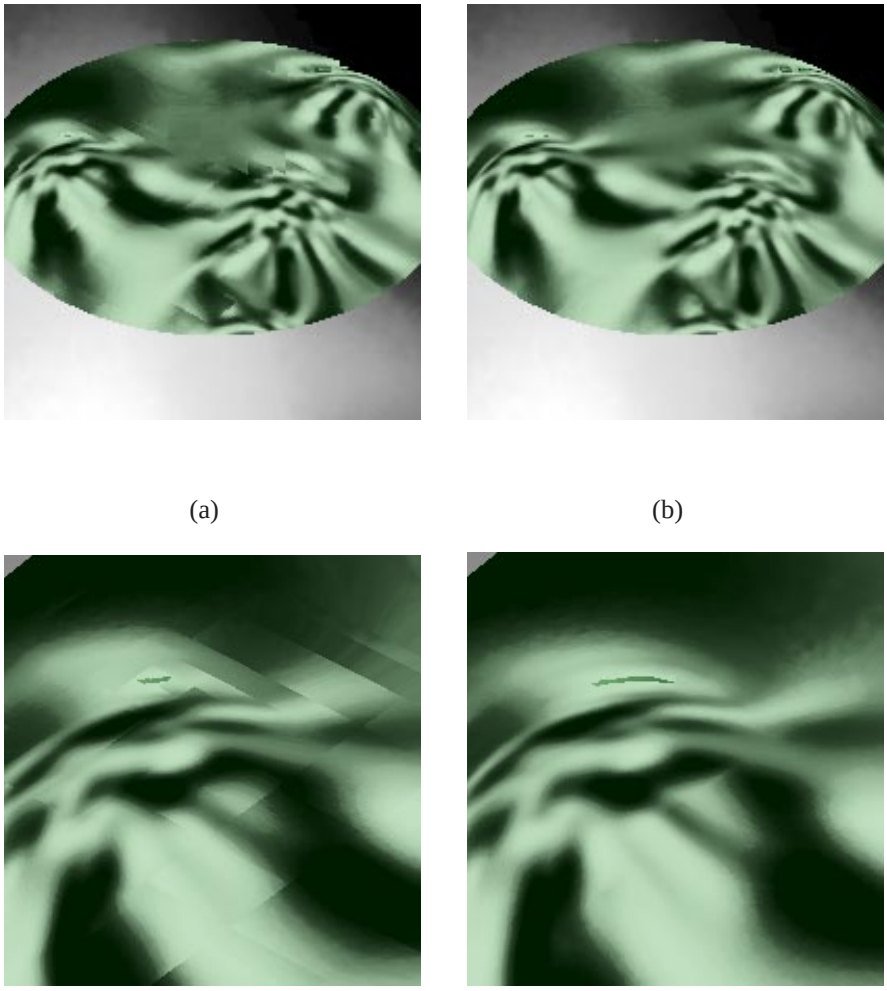
$d!$ congruent simplices. We employ a bisection process that was proposed by Maubach [Mau95]. Whenever a simplex is bisected, some of its neighboring simplices may need to be bisected as well, in order to guarantee that the entire subdivision remains compatible. (See Figure 5.1(b) for an example.)

To illustrate the advantage of interpolation using simplicial complexes, consider the images generated from our ray-tracing application in Figure 5.2. Images (a) and (c) show the result of an interpolation based on kd-trees [AM03], which is not a cell complex, and images (b) and (d) show the results of using the hierarchical simplicial decomposition described in this chapter.

For interpolation purposes, compatibly refined simplicial meshes are preferable over kd-tree/quadtree based subdivisions, not only because they guarantee C^0 continuous interpolants, but also because that they are much simpler in the sense that the interpolations are performed with a minimal number of samples, for example, 5 samples for the 4-dimensional case, hence this is much cheaper than the quadrilinear interpolation using 16 samples. Simplicial decompositions are also advantageous since they create much fewer vertices (i.e. much fewer samples) to reach a certain level of refinement than quadtree/kd-tree based subdivision.

For a simplicial subdivision to be a feasible alternative for our purposes, the following issues are important:

- The scheme for subdivision should be computationally efficient. That is why we have chosen the bisection scheme which decides which edge to be bisected without any computation, but simply based on the order of vertices.



(a) (b)

(c) (d)

Figure 5.2: 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).

- It should support efficient identification of the leaf cell containing a query point. Note that, for kd-tree based subdivisions, the cutting planes are orthogonal to the coordinate axes, hence it takes only one coordinate comparison to determine which child of a cell contains the query point. In a simplicial subdivision, this is not the case. Thus, it is more expensive to determine which child to descend to, if it is done in the straightforward way of determining the location of a point with respect to a given plane. We have shown an efficient way of doing this for the bisection-based regular simplicial decompositions.

- It should support efficient computation of *barycentric coordinates*. If barycentric coordinates are computed *after* locating the leaf cell containing the query point, this involves the inversion of a $(d+1) \times (d+1)$ matrix, hence, computationally quite expensive. Instead, we have shown how this can be done incrementally in a much simpler way with one comparison per level.

- It should support efficient neighbor finding. This is essential for providing compatibility, since a number of neighbor simplices have to be subdivided as well, whenever a simplex is subdivided. In addition, computing facet neighbors of a simplex efficiently is in general of great interest for many applications that require moving along adjacent simplices.

In this chapter, we present an efficient implementation of multidimensional hierarchical regular simplicial meshes in any dimension d. Rather than representing the hierarchy explicitly as a tree using child pointers, we access nodes through an index called a *location code*. Thus, we provide a pointerless representation. Location codes [Sam90a]

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. Also, the space savings realized by not having to store pointers (to the parent, two children, and $d + 1$ neighboring simplices) and simplex vertices is quite significant for large multidimensional meshes.

We store the hierarchical mesh in a data structure called a *simplex decomposition tree*. Our hierarchical decomposition is based on the same bisection method given by Maubach [Mau95].(Note that Maubach's representation is not pointerless.) We present a location code, called the *LPT code*, which can be used to access nodes of this tree. We show how to perform tree traversals, point locations, and answer interpolation queries efficiently through the use of these codes. We also show how to compute neighboring simplices using this code, which is an important step in guaranteeing that the subdivision is a cell complex.

5.1.1 Hierarchical Regular Subdivisions and Pointerless Representations

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 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. Regular hierarchical decompositions can be selectively refined and coarsened efficiently, which is useful for interactive

visualization. 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 known 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 does the 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) [CHL99]. 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 [AV88, Arg02] and cache-sensitive and cache-oblivious data structures and algorithms [CHL99, Dem02].

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 [Gar82, Sam90a]. 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 in the area of pointerless representations for the same class of regular simplicial meshes and neighbor computation has principally been in 2- and 3-dimensions. Evans, Kirkpatrick and Townsend [EKT01] presented a location code for the 2- dimensional case and provided an efficient neighbor finding method based on bit manipulation. Hebert [Heb94] presented a location code for hierarchical tetrahedral meshes and a set of rules to compute neighbors efficiently in 3-space. Lee, De Floriani and Samet [LDS01] developed an alternative location code for this same tetrahedral mesh, and presented algorithms for efficient neighbor computation. In both approaches, the neighbor finding methods are quite specific to 3-space, and are not readily generalizable to higher dimensions. We present neighbor finding methods in arbitrary dimensions with a very compact representation and using very few special cases.

We introduce a new location code that provides a unique encoding of the simplices generated by Maubach's [Mau95] bisection algorithm. This labeling scheme 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 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 recursively during construction of the tree [Mau96], and stores pointers to neighbors for each simplex. We, on the other hand, are

interested in efficiently computing any neighbor of any simplex 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 in $O(1)$ time, if the operations are encoded in lookup tables. Neighbor computation is a valuable operation not only during construction of the hierarchy, but in general, for any application that requires moving between adjacent simplices of a decomposition.

5.2 Preliminaries

Throughout, we consider real d-dimensional space, $\Re^d$. 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 $\Re^d$ 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 ith unit vector. 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 ith vertex is $\sum_{j=1}^{i} \mathbf{e}_j - \sum_{j=i+1}^{d} \mathbf{e}_j$.

For example, in $\Re^3$ we have

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

Recall from basic geometry that two geometric objects are *congruent* if are equiv-

alent 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.

5.2.1 Permutations and Reflections

Let Sym(d) denote the *symmetric group* of all $d!$ permutations over $\{1, 2, \ldots, d\}$. We denote a permutation $\Pi \in$ 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 ith row is the unit vector $\mathbf{e}_{\pi_i}$. For example, for $\Pi = [2\ 3\ 1]$,

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

It is well known that the collection of simplices $\{S_\emptyset\Psi : \Psi \in \text{Sym}(d)\}$ fully subdivides the reference hypercube, and further that this subdivision is compatible (is a simplicial complex) [AG79]. These $d!$ simplices form the starting point of our hierarchical simplicial mesh. 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 $\Re^d$, 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 ith unit vector to $r_i \mathbf{e}_{\pi_i}$.

For example, in $\Re^3$, 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, *perm*(Π), extracts the permutation part of Π, the second, *refl*(Π), extracts the (unpermuted) reflection part as a vector in $\{\pm 1\}^d$, and the third, *orth*(Π), returns the permutation of *refl*(Π) under Π. More formally,

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

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

Lemma 5.2.1 *Let* Π *be a signed permutation. Then*

$$\begin{aligned} \Pi &= \textit{diag}(\textit{refl}(\Pi))\textit{perm}(\Pi) \\ &= \textit{perm}(\Pi)\textit{diag}(\textit{orth}(\Pi)). \end{aligned}$$

5.2.2 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 a process of repeated subdivision, called *bisection* [Mau95]. (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 result-

ing 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 [Mau95] 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:

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

A portion of the tree is illustrated in Figure 5.3. Note that in both cases the first ℓ vertices are unchanged. The new ℓth vertex is the midpoint of the edge between the ℓth 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

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

Our bisection scheme is geometrically equivalent to the one defined by Maubach [Mau95], but we order the vertices differently from Maubach. Although the differences are theoretically insignificant, our ordering results in somewhat simpler and more regular formulas for computing descendents and neighbors.

5.2.3 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 nodes of the decomposition tree into a

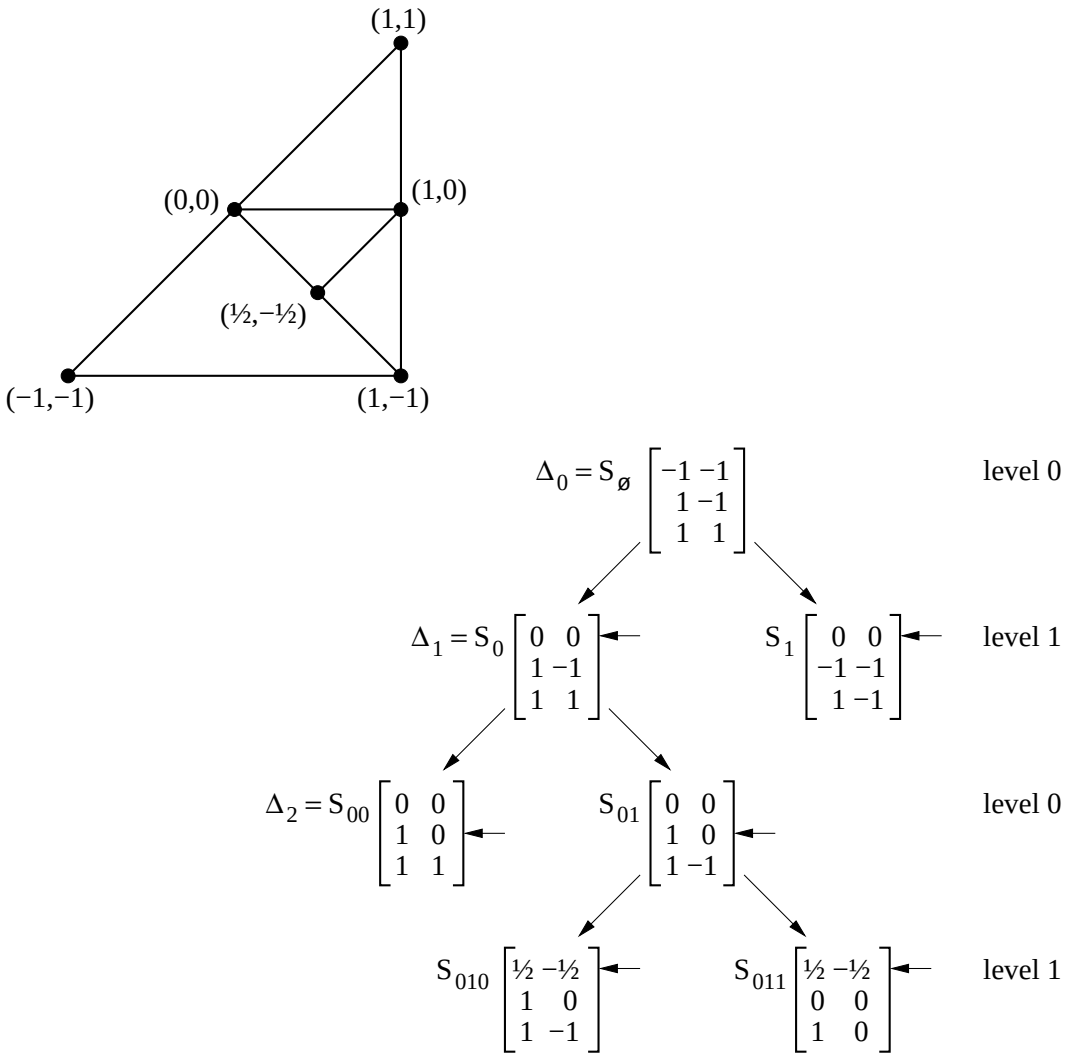


Figure 5.3: The simplex decomposition tree. The corresponding bisected simplex is shown on the top-left. The newly created vertex is indicated by an arrow in each case. The reference simplices Δ_i are indicated as well.

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 Figure 5.3.) 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 $\Re^3$ the 3 reference simplices together with Δ_3 are

$$\underset{(S_\emptyset)}{\Delta_0} = \begin{bmatrix} -1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \underset{(S_0)}{\Delta_1} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\underset{(S_{00})}{\Delta_2} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \underset{(S_{000})}{\Delta_3} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

5.3 The LPT code

So far we have defined an infinite decomposition tree and a procedure for generating the simplices of this tree from the top down. In order to provide pointerless implementation of the hierarchical mesh, we define a *location code*, which uniquely identifies encodes each simplex of the hierarchy. The most direct location code is 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 [LDS01]. Instead we modify an approach presented by Hebert [Heb94] 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 it encodes for each simplex 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), where $\ell = |p| \bmod d$ is the simplex's level, Π is a signed permutation relating $S_{\Psi,p}$ to its reference simplex, and 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 and orthant list $\Phi = \Phi_{\Psi,p}$ are defined below as functions of Ψ and p. Correctness will be established in Theorem 5.3.1 below.

Permutation Part: The signed permutation $\Pi_{\Psi,p}$ is defined recursively as follows for a base permutation Ψ and binary path p:

$$\Pi_{\Psi,\emptyset} \;=\; \Psi \qquad\qquad \Pi_{\Psi,p0} \;=\; \Pi_{\Psi,p} \qquad\qquad \Pi_{\Psi,p1} \;=\; \Pi_{\Psi,p} \circ \Sigma_\ell, \tag{5.1}$$

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 Figure 5.4. 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 Figure 5.4). 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 a vector defines the *orthant* containing the descendent hypercube relative to the central vertex of its ancestor. Consider, for example, the shaded simplex in Figure 5.4. 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.

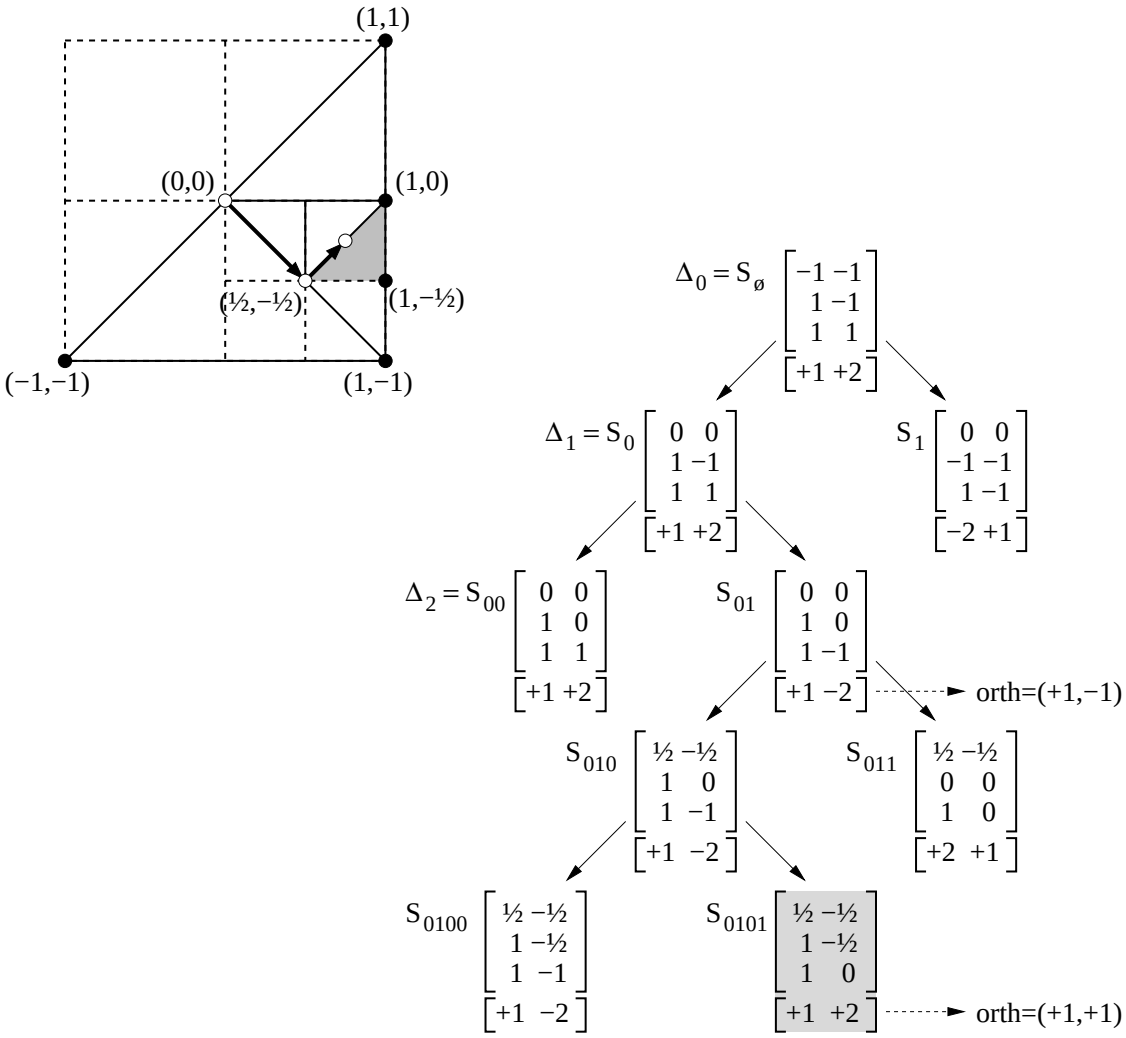


Figure 5.4: The signed permutations $\Pi_{\Psi,p}$ associated with each simplex are shown below each simplex matrix, and the entries of the orthant list are shown for the shaded simplex S_{0101}. The LPT code for this simplex is $(0, [+1\ +2], \langle(+1,-1),(+1,+1)\rangle)$.

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 Figure 5.5.) 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 ith element is *orth*$(\Gamma_{\Psi,p}[i])$, that is

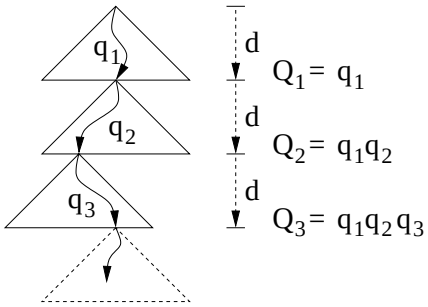


Figure 5.5: Orthant List

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

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. (5.1) and append *orth*(Π') to the current list. Observe that given the level ℓ and the orthant list Φ for any simplex, we can derive the length of the associated tree path p as $\ell + d \cdot \textit{length}(\Phi)$.

The computation of the LPT code is summarized in the procedure *LPTcode* shown in Figure 5.6. The code for the simplex $S_{\Psi,p}$ is computed by the call *LPTcode*$(p, (0, \Psi, \emptyset))$. 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 [Heb94] proved the analogous

LPTcode$(p, (\ell, \Pi, \Phi))$

if $(p = \emptyset)$ **return** (ℓ, Π, Φ)

Express p as xq, for $x \in \{0, 1\}$

$\ell \leftarrow (\ell + 1) \bmod d$

if $(x = 1)$ $\Pi \leftarrow \Pi \circ \Sigma_\ell$

if $(\ell = 0)$ $\Phi \leftarrow \Phi +$ *orth*(Π)

return *LPTcode*$(q, (\ell, \Pi, \Phi))$

Figure 5.6: Procedure *LPTcode*

result for his 3-dimensional bisection system. 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 **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 **v**, and hence adding this to any simplex matrix is equivalent to a translation by **v**.

Theorem 5.3.1 (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:*

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

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

Before proving this theorem, we will prove the following technical lemmas.

Lemma 5.3.1 *Given the reference simplex Δ_ℓ, $0 \leq \ell < d$,*

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

where Σ_ℓ is as defined in Section 5.3.

Proof: Here is an informal justification of the lemma. Since Σ_ℓ is an orthogonal matrix, the following holds.

$$\Sigma_\ell^{-1} = \Sigma_\ell^T = [e_1^T \dots e_\ell^T \quad -e_d^T \quad e_{\ell+1}^T \dots e_{d-1}^T]$$

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 Figure 5.7. 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. □

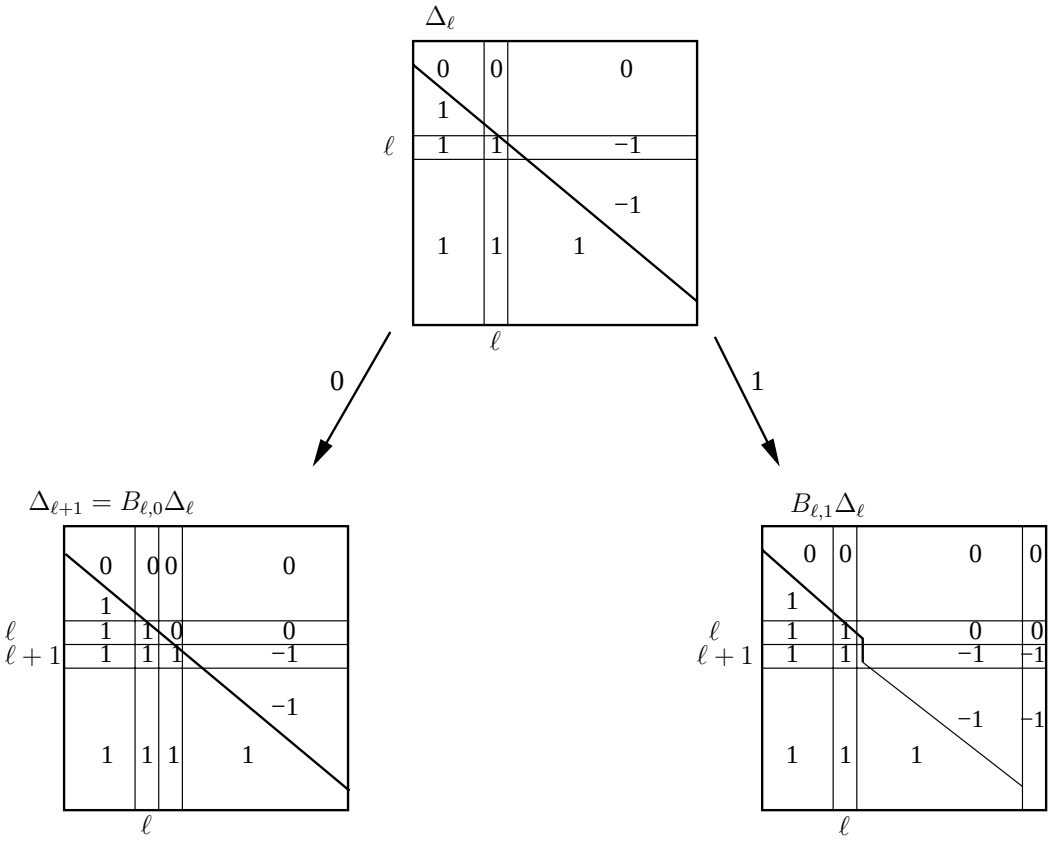


Figure 5.7: The two children of a reference simplex.

Lemma 5.3.2 *Given the reference simplex Δ_ℓ, $0 \leq \ell < d$, and a signed permutation $\Pi_{\Psi,p}$,*

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

Proof: By definition, $\Pi_{\Psi,p1} = \Sigma_\ell\Pi_{\Psi,p}$, that is $\Pi_{\Psi,p} = \Sigma_\ell^{-1}\Pi_{\Psi,p1}$.

$$B_{\ell,1}\Delta_\ell\Pi_{\Psi,p} = \Delta_{\ell+1}\Sigma_\ell\Pi_{\Psi,p} = \Delta_{\ell+1}\Sigma_\ell\Sigma_\ell^{-1}\Pi_{\Psi,p1} = \Delta_{\ell+1}\Pi_{\Psi,p1}$$

□

Proof: (Theorem 5.3.1) We will prove Theorem 5.3.1 by induction. Recall that $\Pi = \Pi_{\Psi,p}$, $\Phi = \Phi_{\Psi,p}$, $l = |p| \bmod d$ and $L = \lfloor |p|/d \rfloor$.

Induction Basis: The hypothesis holds for all root simplices, $S_{\Psi,\emptyset}$. Since $L = \lfloor |p|/d \rfloor = 0$, and $\ell = 0$ at root level,

$$\begin{aligned} S_{\Psi,\emptyset} &= \tfrac{1}{2^0}\Delta_0\Pi_{\Psi,\emptyset} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{0} \tfrac{1}{2^i}\Phi_{\Psi,\emptyset}[i] \\ &= \Delta_0\Pi_{\Psi,\emptyset} \quad \text{(holds by definition).} \end{aligned}$$

Induction Step: Assume that the inductive 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 $S_{\Psi,p}$. In addition to the above lemmas, we will make use of the following equalities:

$$\Delta_{\ell+1} \;=\; B_{\ell,0}\Delta_\ell.$$

Let $T = \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p}[i]$. Note that,

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

as well. Also, note that all rows of T are equal to each other. For such a matrix T, the following equalities hold,

$$B_{\ell,0}T \;=\; T, \quad B_{\ell,1}T \;=\; T.$$

In the induction, there are two cases to be distinguished depending on ℓ:

1. $0 \leq \ell < d-1$

   (a) First, consider $S_{\Psi,p0}$. By definition, $\Pi_{\Psi,p0} = \Pi_{\Psi,p}$.

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

   This completes the induction for $S_{\Psi,p0}$, since $\lfloor |p0|/d \rfloor = L$ for $0 \leq \ell < d-1$.

   (b) Now, consider $S_{\Psi,p1}$.

$$\begin{aligned} S_{\Psi,p1} &= B_{\ell,1}S_{\Psi,p} = B_{\ell,1}(\tfrac{1}{2^L}\Delta_\ell\Pi_{\Psi,p} + T) && \text{(by ind. hyp.)} \\ &= \tfrac{1}{2^L}\Delta_{\ell+1}\Pi_{\Psi,p1} + T. && \text{(by Lemma 5.3.2)} \end{aligned}$$

   This completes the induction for $S_{\Psi,p1}$, since $\lfloor |p1|/d \rfloor = L$ for $0 \leq \ell < d-1$.

2. $\ell = d-1$, the children of $S_{\Psi,p}$ will be at *level* 0.

   (a) First, consider $S_{\Psi,p0}$. By definition, $\Pi_{\Psi,p0} = \Pi_{\Psi,p}$.

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

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

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

By Lemma 5.2.1,

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

And so, we see that, the third term above is 0, yielding

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

This completes the induction for $S_{\Psi,p0}$, since $\lfloor |p0|/d \rfloor = L+1$ for $\ell = d-1$.

(b) Next, consider $S_{\Psi,p1}$.

$$\begin{aligned} S_{\Psi,p1} &= B_{\ell,1}S_{\Psi,p} \\ &= B_{\ell,1}(\tfrac{1}{2^L}\Delta_{d-1}\Pi_{\Psi,p} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p}[i]) \qquad \text{(by ind. hyp.)} \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p1} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p1}[i] \qquad \text{(by Lemma 5.3.2)} \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p1} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p1}[i] - \frac{1}{2^{L+1}}\mathbf{1}_{d+1}^T\Phi_{\Psi,p1}[L+1]. \end{aligned}$$

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

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

This completes the induction for $S_{\Psi,p1}$, since $\lfloor |p1|/d \rfloor = L+1$ for $\ell = d-1$.

□

Implementation Issues: We can now describe a pointerless implementation of a simplex decomposition tree. For each simplex $S_{\Psi,p}$ in the tree, we create an node that is indexed by an appropriate encoding of the associated LPT code. Theorem 5.3.1 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. In addition to the index, 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 denotes the maximum depth of any node in the tree. Each of the $d!$ permutations of Sym(d) can be encoded as a integer with $\log_2 d!$ bits [Knu73]. 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 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 5.5.) 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 $d \lfloor D/d \rfloor \leq D$. The total number of bits needed to represent the code for a simplex at depth D is $D + \log_2(d!) + O(d)$. This is close to optimal in the worst case, since there are $2^D d!$ simplices at depth D in a full tree. If we assume that the machine's word size is $\Omega((D/d) + \log_2 d!)$, then 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. (5.1) and the neighbor permutations of Section 5.5 below) can be precomputed and stored in tables. This allows very fast evaluation of permutation operations by simple table look-up.

5.4 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, and computing neighbors in the simplicial complex.

5.4.1 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 5.3. 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.

Lemma 5.4.1 *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 $\boldsymbol{o} = (o_i)_{i=1}^d$ 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$(o_{|\pi_{\ell^*}|})$.*

Proof: In order to prove Lemma 5.4.1, we prove the following more general lemma, which characterizes the child relations for a simplex's ancestors, up to the next 0th level.

Lemma 5.4.2 *Let $S_{\Psi,p}$ be a nonroot simplex, and let $S_{\Psi,t}$ be its nearest proper ancestor of level 0. Let $\boldsymbol{o} = orth(\Pi_{\Psi,t})$ 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 $\boldsymbol{o} = (o_i)_{i=1}^d$. Then*

$$b_i = \begin{cases} 0 & \textit{if} \quad \textit{sign}(\pi_i) = \textit{sign}(o_{|\pi_i|}) \\ 1 & \textit{if} \quad \textit{otherwise.} \end{cases},$$

Proof: We do not know what $\Pi_{\Psi,t}$ is, but since we know **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 down from $S_{\Psi,tb_1\ldots b_{i-1}}$ to $S_{\Psi,tb_1\ldots b_i}$. $\Pi_{\Psi,tb_1\ldots b_{i-1}}$ 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.

$$\overbrace{[+1 - 4 - 3 + 2]}^{S_{\psi,t}} \xrightarrow{1} [-2 + 1 - 4 - 3] \xrightarrow{0}$$

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

Consider the above example where $\ell = 3$. Note that $(o_i)_1^d = (+1, +1, -1, -1)$. o_2 had a positive sign, following the 1-path, it was negated, and became the first element in $\Pi_{\Psi,p}$, because after it was shifted to the first location, it was fixed. Similarly, following the 0-path, 1 remained positive and got fixed at the second location, and following 1-path 3 was negated and placed at the third location. And so, the path from $S_{\Psi,t}$ to $S_{\Psi,p}$ is 101.

□

Now Lemma 5.4.1 follows as an immediate corollary since $S_{\Psi,p}$ is a 0-child, if and only if $b_{\ell^*} = 0$. □

Lemma 5.4.1 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 **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. (5.1) and Theorem 5.3.1 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 **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. The computation of the parent is summarized in the procedure *parent* in Figure 5.8. The parent of the simplex $S_{\Psi,p}$ with LPT code (ℓ, Π, Φ) is computed by the call *parent*$(p, (\ell, \Pi, \Phi))$.

parent$(p, (\ell, \Pi, \Phi))$

if $(p = \emptyset)$ **return** $\emptyset$

Express p as $tb_1b_2 \ldots b_{\ell^*}$.

$\ell' \leftarrow (\ell - 1) \bmod d$

if $(\ell = 0)$ $\Phi' \leftarrow \Phi - \langle \Phi[L] \rangle$

else $\Phi' \leftarrow \Phi$

if $(b_{\ell^*} = 0)$ $\Pi' \leftarrow \Pi$

else $\Pi' \leftarrow \Pi \circ \Sigma_{\ell'}^{-1}$

return (ℓ', Π', Φ')

Figure 5.8: The procedure *parent*.

5.4.2 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.

We begin by locating 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. Let us assume that we have a function *sortDescending* that computes this permutation.

Letting $\mathbf{v}_i$ denote the ith vertex of the root simplex $S_{\Psi,\emptyset}$ that contains $\mathbf{q}$, the *barycentric coordinates* of $\mathbf{q}$ with respect to this simplex is the unique $d+1$ vector $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$, $0 \leq \alpha_i \leq 1$, such that $\sum_i \alpha_i = 1$ and $\mathbf{q} = \sum_i \alpha_i \mathbf{v}_i$. Because of the special structure of

Δ_0, it is easy to verify that the procedure *findRoot* shown in Figure 5.9 computes these coordinates.

After this initialization, we recursively descend the hierarchy until finding a leaf simplex. We use the barycentric coordinates of **q** relative to the current simplex to determine in which child it resides. Then we generate the barycentric coordinates of **q** with respect to this child. This is done with the aid of the following lemma, which is proved in the appendix. The descent algorithm is given in Figure 5.9 and its correctness follows from Lemma 5.4.3. To simplify the presentation, we have omitted the orthant list processing, but it is essentially the same as in the code block just prior to Theorem 5.3.1.

Lemma 5.4.3 *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 **q** lies within this simplex with the barycentric coordinates $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$.*

- *If $\alpha_\ell \leq \alpha_d$, then **q** lies in the 0-child. Let $\boldsymbol{\alpha}'$ be the $(d+1)$-vector that is identical $\boldsymbol{\alpha}$ except that $\alpha'_\ell = 2\alpha_\ell$ and $\alpha'_d = \alpha_d - \alpha_\ell$. Then the barycentric coordinate vector of **q** relative to this child is $\boldsymbol{\alpha}'$.*
- *Otherwise, **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 **q** relative to this child is $\boldsymbol{\alpha}'\Sigma'_\ell$.*

Proof: Let $S_{\Psi,p}$ be the simplex of the hierarchy at level ℓ that contains **q**. Let $\Pi_{\Psi,p} = [\pi_i]_{i=1}^d$ be the associated permutation vector. Let $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$ denote **q**'s barycentric

findRoot($\mathbf{q}$)

$\Psi \leftarrow$ *sortDescending*$((q)_{i=1}^{d})$

$\alpha_0 \leftarrow (1 - q_{\psi_1})/2$

$\alpha_d \leftarrow (1 + q_{\psi_d})/2$

for $(0 < i < d)$ $\alpha_i \leftarrow (q_{\psi_i} - q_{\psi_{i+1}})/2$

return$(\Psi, \boldsymbol{\alpha})$

search$(\mathbf{q}, (\ell, \Pi), \boldsymbol{\alpha})$

if $((\ell, \Pi)$ 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*$(\mathbf{q}, ((\ell + 1) \bmod d, \Pi \circ \Sigma_\ell), \boldsymbol{\alpha}'\Sigma'_\ell)$

Figure 5.9: 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 5.4.3 and the permutation Σ_ℓ was given in Section 5.3, Eq. 5.1.

coordinates with respect to $S_{\Psi,p}$. Let $\mathbf{v}_i$ denote the ith 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,

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

And so, if $(\alpha_d - \alpha_\ell) \geq 0$, it follows that $\mathbf{q}$ resides in $S_{\Psi,p0}$, and otherwise it resides in $S_{\Psi,p1}$. From the above equations, we can also see the barycentric coordinates when $\mathbf{q}$ resides in $S_{\Psi,p0}$ or in $S_{\Psi,p1}$.

□

Given the query point $\mathbf{q}$, the point location procedure first calls *findRoot* to find the appropriate root simplex Ψ of the decomposition tree and the barycentric coordinates $\boldsymbol{\alpha}$. Then it invokes the recursive procedure *search*$(0, \Psi, \boldsymbol{\alpha})$ to locate $\mathbf{q}$ within the appropriate

root simplex. Once the point has been located, we can answer the interpolation query for this point. We access the stored vector field values at each of the simplex vertices, and then weight these values according to the barycentric coordinates of **q**. The result is a piecewise linear, continuous interpolant.

This simple sequential search makes as many memory accesses as the depth of the final leaf simplex that contains **q**. A more efficient procedure in terms of 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 locate the exact depth of the leaf simplex that contains **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.

5.5 Neighbors in the Simplicial Complex

As we mentioned earlier, when simplices of the decomposition tree are 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.

In this section we provide rules for computing facet neighbors based solely on their LPT codes. In all but one of the cases, the neighbor can be computed in $O(d)$ time, independent of the depth of the simplex. In the case where the computation may require time proportional to the depth in the tree, we show that this computation can be sped up by a factor of d times the machine's word size, and so is nearly constant time for practical purposes.

Consider a simplex S in the complex defined by the decomposition tree. For $0 \leq i \leq d$, let $\mathbf{v}_i$ denote its ith 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 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 proof of their correctness is based on a straightforward but lengthy induction argument.

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 show how to compute the LPT codes of those neighbors, as well.

5.5.1 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 5.4.1. These permutations are illustrated in Figure 5.10, 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.

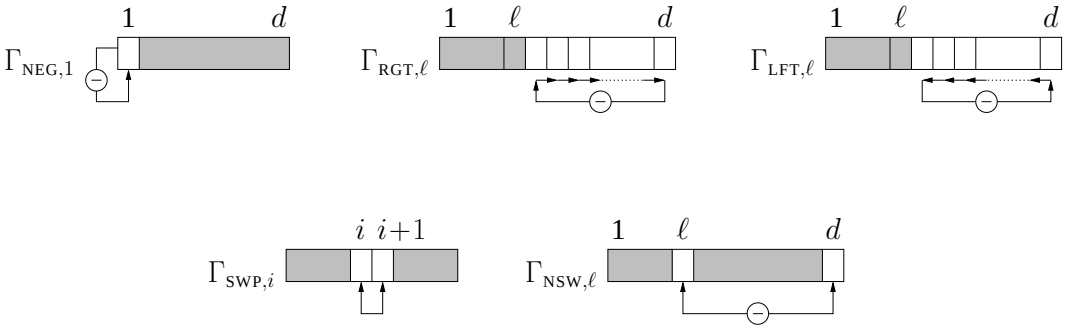


Figure 5.10: Neighbor permutations. (The circle with a minus sign indicates that the element is negated.)

The neighbor rules are given in Theorem 5.5.1. 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.

Theorem 5.5.1 *Let S denote a simplex at level ℓ, and let Π denote the permutation code for S.*

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

The proof is presented at the end of this chapter.

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 5.2.1. 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.

5.5.2 Neighbor Orthant List

In order to compute the orthant list component of the neighbor, from the LPT code of S, we distinguish 3 cases:

1. If $\ell \neq 0$ or $1 \leq i < d$, $N^{(i)}(S)$, is in the same final orthant as S, and so $\Phi^{(i)} = \Phi$.

2. 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 $\mathit{orth}(\Pi^{(d)})$ in $\Phi^{(d)}$. Thus the orthant list can be updated in $O(d)$ time in this case.

3. 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.

To compute $\Phi^{(0)}$, we use a method similar to the one for computing neighbor quadrants in quadtrees [Sam90a]. In our representation, the path from the root to the orthant is the list of orthants in Φ. Consider the 2-dimensional example in Figure 5.11(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.)

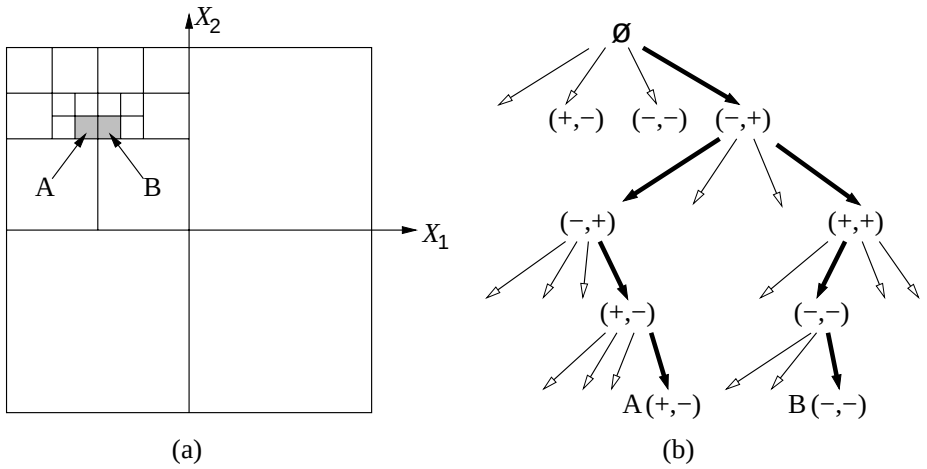


Figure 5.11: Orthant B is a neighbor of orthant A in $+X_1$ direction. (a) The quadtree-like subdivision of space (b) The corresponding tree representation.

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

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

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. Figure 5.11(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 neighbor orthant B of equal size located in the given direction with respect to A.

Similar to the algorithm described for quadtrees by Samet [Sam90a], the algorithm to find 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}. And 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.

Now, if we consider the original problem of computing $\Phi^{(0)}$ corresponding to the 0^{th} neighbor of simplex S, we can use the algorithm explained above, if we know 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*(π_1).

Implementation Issues: This operation can be implemented very rapidly through a simple trick with bit manipulations. The neighbor computation [Sam90a] essentially involves an operation, which is applied to a bit string that consists of the ith 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 [LDS01].) Under the assumption that the machine's word size is $\Omega(D/d)$, where D is the maximum depth of any simplex, it follows that the orthant list for the neighbor can be computed in $O(1)$ time.

5.6 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 [Heb94] and Maubach [Mau95] 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 5.5 provide a means to locate same-depth neighboring simplices that share a common $(d-1)$-face with S, that is, the *facet neighbors* of S. Let $N_e(S)$ denote the facet neighbors of S that contain the edge e, or equivalently, the facet neighbors lying opposite all the $d-1$ vertices of S other than the endpoints of e. In order to access all the simplices of $E_e(S)$ we compute facet neighbors recursively. The algorithm was given by Maubach [Mau95], and is shown as the recursive function *compatBisect* in the codeblock shown in Figure 5.12. The procedure *simpleBisect* performs the basic bisection step described in Section 5.2.2.

```
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)
```

Figure 5.12: Procedure *compatBisect*

Maubach proved that in a compatible subdivision, the facet neighbors of S needed in this refinement, either appear at the same depth as S or one level closer to the root

[Mau95]. 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.

5.7 Neighbors at different depths

Neighbor rules of Theorem 5.5.1 provide the LPT code for the same depth neighbors. However, 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 neighbors that do not. Maubach already proved that a neighbor sharing the edge-to-be-bisected is either at depth $|p|$ or at depth $|p| - 1$, and that 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 above rules, and if the same depth neighbor does not exist in the tree, we compute its parent's LPT code as described in Section 5.4.

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 neighbor of S_p across vertex $\mathbf{v}_{\ell}$,

if $(N^{(d)}(S_p)$ is a bisected simplex$)$

$N^{\ell}(S_{p1})$ is the neighbor of S_p across vertex $\mathbf{v}_d$.

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 **u**. It can be shown that when $N^{\ell}(S_p)$ (resp., $N^{(d)}(S_p)$) is bisected, **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 (**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 **u**, hence these children at depth $|p|+2$ cannot be neighbors of S_p.

5.8 Conclusions

In this chapter, we have presented a representation of hierarchical regular simplicial meshes based on Maubach's [Mau95] 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 nodes, 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 nodes. (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.

5.9 Proof of Theorem 5.5.1

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.

S_0 and S_1 denote the 0- and 1-children of S, respectively.

$S_0^{(i)}$ and $S_1^{(i)}$ denote the 0- and 1-children of $S^{(i)}$, respectively.

Π_0 and Π_1 denote the signed permutation code associated with S_0 and S_1, respectively.

$\Pi_0^{(i)}$ and $\Pi_1^{(i)}$ denote the signed permutation code associated with $S_0^{(i)}$ and $S_1^{(i)}$, respectively.

$(S_0)^{(i)}$ denote the i^{th} neighbor of S_0. $(\Pi_0)^{(i)}$ denotes the code for $(S_0)^{(i)}$.

$(S_1)^{(i)}$ denote the i^{th} neighbor of S_1. $(\Pi_1)^{(i)}$ denotes the code for $(S_1)^{(i)}$.

m and **m**′ are used to denote the new vertex generated by bisection.

u is used for the vertex that differs in the neighbor simplex.

Inductive 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 $\ell^* = \ell^- + 1$.

The rules of the theorem can be stated more explicitly as:

if (S is a 0-child)

$$S^{(0)} = [\mathbf{u}\ \mathbf{v}_1 \dots \mathbf{v}_d]^T \qquad \Pi^{(0)} = \Pi \circ \Gamma_{\text{NEG},1}$$

$$S^{(i)} = [\mathbf{v}_0 \dots \mathbf{v}_{i-1}\ \mathbf{u}\ \mathbf{v}_{i+1} \dots \mathbf{v}_d]^T, \quad (0 < i < d) \qquad \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i}$$

$$S^{(d)} = [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-}\ \mathbf{u}\ \mathbf{v}_{\ell^*} \dots \mathbf{v}_{d-1}]^T \qquad \Pi^{(d)} = \Pi \circ \Gamma_{\text{RGT},\ell^-}$$

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 of Induction: We will show that the neighbor rules hold for the $d!$ root simplices. Note that the *level* of a root simplex is 0, and the rules are the same whether the simplex is a 0-child, or a 1-child. Let S denote any root simplex, with $LPT\ code$ $\Pi = [\pi_1 \dots \pi_i \quad \pi_{i+1} \dots \pi_d]$.

- For all root simplices, $S^{(0)} = \emptyset$, and $S^{(d)} = \emptyset$, that is, the 0^{th} and the d^{th} neighbors do not exist, since they are outside the *reference hypercube*.

- Other neighbors, $S^{(i)}$, $0 < i < d$, should be obtainable by swaps. Recall that the *base simplex* $S_\emptyset$ can be represented as,

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

and, any root simplex S can be written as,

$$S = [\mathbf{y}'_1 \dots \mathbf{y}'_d], \quad \mathbf{y}'_j = \mathbf{y}_i = \begin{bmatrix} y_{i,0} \\ \vdots \\ y_{i,i-1} \\ y_{i,i} \\ \vdots \\ y_{i,d} \end{bmatrix} = \begin{bmatrix} -1 \\ \vdots \\ -1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad \text{iff} \quad \pi_i = j.$$

If $\pi_i = j$, and $\pi_{i+1} = k$, then $\mathbf{y}'_j = \mathbf{y}_i$ and $\mathbf{y}'_k = \mathbf{y}_{i+1}$.

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

Note that swapping columns $\mathbf{y}'_j$ and $\mathbf{y}'_k$ of S, will give us another valid root simplex, S' that differs from S only in the i^{th} row, that is the i^{th} vertex. So, S' is basically 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 $(i+1)^{th}$ entries in Π are swapped to obtain Π'. And so,

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

Induction Step: Let S be a simplex at level ℓ^- such that the inductive hypothesis holds. We will show that the inductive 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.

1. $i = 0$

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

$$S{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T, \qquad S_0{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^*-1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \ldots \mathbf{v}_d]^T$$

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

   Otherwise if $\quad \ell^- = 0$ then

$$S{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \ldots \mathbf{v}_d]^T, \qquad S_0{=}[\mathbf{m} \quad \mathbf{v}_1 \ldots \mathbf{v}_d]^T$$

$$S^{(0)}{=}[\,\mathbf{u} \quad \mathbf{v}_1 \ldots \mathbf{v}_d]^T, \quad S_0^{(0)}{=}[\mathbf{m}' \quad \mathbf{v}_1 \ldots \mathbf{v}_d]^T.$$

   In either case, $(S_0)^{(0)} = S_0^{(0)}$.

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

2. $i = d$

   By definition of the bisection rules, the d^{th} neighbor of S_0 is its sibling, that is S_1, and so,

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

3. $0 < i < \ell^-$

$$S{=}[\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{v}_i \quad \mathbf{v}_{i+1} \ldots \mathbf{v}_{\ell^*-1} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T,$$

$$S^{(i)}{=}[\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{u} \quad \mathbf{v}_{i+1} \ldots \mathbf{v}_{\ell^*-1} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T,$$

$$S_0{=}[\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{v}_i \quad \mathbf{v}_{i+1} \ldots \mathbf{v}_{\ell^*-1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \ldots \mathbf{v}_d]^T,$$

$$S_0^{(i)}{=}[\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{u} \quad \mathbf{v}_{i+1} \ldots \mathbf{v}_{\ell^*-1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \ldots \mathbf{v}_d]^T.$$

In this case $(S_0)^{(i)} = S_0^{(i)}$, and so

$$(\Pi_0)^{(i)} = \Pi_0^{(i)} = \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i} = \Pi_0 \circ \Gamma_{\text{SWP},i}.$$

4. $i = \ell^-$, $\ell^- \neq 0$

   (a) If S is a 0-child then

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

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

   (b) If S is a 1-child then

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

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

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

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

5. $\ell^- < i < d$

$$S=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_d]^T$$

$$S^{(i)}=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-1} \quad \mathbf{y} \dots \mathbf{v}_d]^T$$

$$S_0=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_d]^T$$

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

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

This completes the case of the 0-child.

Next, consider the 1-child of S, that is S_1. Let ℓ denote the level of S_1. Note that $\ell^- = (\ell - 1) \bmod d$. Let $\ell^* = \ell^- + 1$. Letting i denote the neighbor number, again, there are multiple cases.

1. $i = 0$

   (a) $\ell^- = 0$

$$S=[\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T, \qquad S_1=[\mathbf{m} \quad \mathbf{v}_0 \dots \mathbf{v}_{d-1}]^T$$

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

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

$$\Pi = [\pi_1 \dots \pi_d], \qquad \Pi_1 = [-\pi_d \quad \pi_1 \dots \pi_{d-1}],$$

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

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

(b) $\ell^- \neq 0$

$$S{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T$$

$$S^{(0)}{=}[\,\mathbf{y} \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T$$

$$S_1{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T$$

$$S_1^{(0)}{=}[\,\mathbf{y} \quad \mathbf{v}_1 \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T.$$

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

$$\Pi = [\pi_1 \ldots \pi_d], \qquad \Pi_1 = [\pi_1 \ldots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \ldots \pi_{d-1}],$$

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

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

2. $0 < i < \ell^-$

$$S{=}[\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{v}_i \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T$$

$$S^{(i)}{=}[\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{y} \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_d]^T$$

$$S_1{=}[\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{v}_i \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T$$

$$S_1^{(i)}{=}[\mathbf{v}_0 \ldots \mathbf{v}_{i-1} \quad \mathbf{y} \ldots \mathbf{v}_{\ell^- - 1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \ldots \mathbf{v}_{d-1}]^T.$$

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

$$\Pi = [\pi_1 \ldots \pi_d],$$

$$\Pi_1 = [\pi_1 \ldots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \ldots \pi_{d-1}],$$

$$\Pi^{(i)} = [\pi_1 \ldots \pi_{i-1} \quad \pi_{i+1} \quad \pi i \quad \pi_{i+2} \ldots \pi_d],$$

$$\Pi_1^{(i)} = [\pi_1 \ldots \pi_{i-1} \quad \pi_{i+1} \quad \pi i \ldots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \ldots \pi_{d-1}].$$

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

3. $i = \ell^-$

(a) If S is a 0-child then

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

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

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

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

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

4. $i = \ell^*$

By definition, the $(\ell^*)^{th}$ neighbor of S_1 is its sibling, that is S_0, and

$$(\Pi_1)^{(\ell^*)} = \Pi_0 = \Pi_1 \circ \Gamma_{\text{LFT},\ell^-}.$$

5. $\ell^* < i \leq d, \quad \ell \neq 0$

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

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

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

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

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

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

$$\Pi = [\pi_1 \dots \pi_{\ell^-} \dots \pi_{d-1} \quad \pi_d],$$

$$\Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}], \quad \text{since} \quad d-1 > \ell^-$$

$$\Pi^{(d-1)} = [\pi_1 \dots \pi_{d-2} \quad \pi_d \quad \pi_{d-1}],$$

$$\Pi_1^{(d-1)} = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_{d-1} \quad \pi_{\ell^*} \dots \pi_{d-2} \quad \pi_d].$$

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

This completes the induction step and so complete the proof of Theorem 5.5.1.

Chapter 6

Using Hierarchical Simplicial Meshes to Render Atmospheric Effects

6.1 Introduction

A fundamental element of computer graphics is producing realistic visualizations of various natural phenomena. An important and challenging problem in this area is that of rendering atmospheric effects such as smoke and dust, which arise as a result of the absorption and scattering of light while passing through a participating medium. In this chapter, we illustrate the practical value of the *SD-tree* for rendering atmospheric effects.

An accurate simulation of the interaction of light with a participating medium is quite complex, since it involves the use of radiative transport theory [Kru90, Cha60]. However, for the purpose of rendering atmospheric effects, simpler models have been proposed, and have been shown to be quite satisfactory. A good survey of different optical models can be found in [Max95]. There has been considerable success in recent years in producing realistic physical models for smoke and related natural phenomena[FM97b, JC98, Max86, NMN87, PPS99, Sta99]. Our interest here is not on how to model such phenomena, but rather on how to render them efficiently.

A number of hardware-based approaches for rendering smoke and other atmospheric phenomena have been proposed in the literature. In the context of rendering, Stam [Sta99] and Fedkiw et al. [FSJ01] propose the use of 3-dimensional texture maps to store the density of the atmospheric medium in each voxel of the texture map, and then render this texture map from front to back. Dobashi, et al. [DYN02] propose a similar approach based on computing a collection of preprocessed sample planes. For approaches involving other types of atmospheric phenomena, see also [BR98, LHJ99, WE98].

However, these methods suffer from a limited ability to model multiple scattering and other effects needed to render media with high albedo. Also, the use of grid-based representations, while amenable to hardware implementation, cannot readily adapt to variations in the density and color of the media. An alternative approach for generating realistic images, which can handle media with high albedo, is based on photon maps [JC98, FSJ01]. This process is more computationally intensive, but achieves a high degree of realism by solving the full volume rendering equation for the medium. This is done in two passes. The first pass builds a photon map for the volume containing the medium, by shooting photons into the medium and storing these as they interact with the medium, and the second pass that integrates the effects of the photon map by forward ray marching. A significant component in the actual rendering time is the numerical integration performed by marching along the length of each ray in order to determine the overall opacity and color of the media.

We propose utilizing the SD-tree for accelerating the ray-marching process. The basic idea is similar to the one applied for accelerating ray-tracing as described in Chapter 4, namely, we can think of the participating medium as a function f, which maps rays

to a pair consisting of the color and opacity. These are the net color and opacity obtained by marching the ray through the medium, until its collision with a solid object. Since we model rays as points in 4-dimensional space, the function f is a function over $\Re^4$:

$$f : \text{ray} \rightarrow (\text{color}, \text{opacity}).$$

In regions where f varies smoothly, we expect that ray coherence can be exploited, that is, nearby rays will pass through regions of similar color and density, and so the accumulated color and opacity will be close to each other.

Hence, we replace wherever possible the computationally intensive numerical integration along each ray with a combination of sampling and interpolation. Rays are sampled adaptively, and the result of the numerical integration (color and opacity) for each of these rays is computed accurately and stored in a 4-dimensional *simplex decomposition tree* that serves as a spatial index. In order to achieve high accuracy, regions with higher variations in color and density sampled with more densely. In order to generate the final rendering, rather than integrating along each ray, we instead interpolate its values from neighboring sampled rays.

Because a simplicial complex is used, we can guarantee a C^0 continuous approximation of f. In addition, interpolations are performed with a minimal number of samples, 5 samples for the 4-dimensional case, and hence, this is much cheaper than the quadrilinear interpolation using 16 samples.

The SD-tree involves a subdivision of 4-dimensional space, and it is well known that the complexities of subdivisions tend to increase exponentially as a function of the dimension. Consequently, it is important to save space wherever possible. We discuss a

number of issues involved in the use of the data structure for the purposes of rendering, and how to minimize the size of the resulting data structure.

The data structure does not rely on any particular model or representation of the medium or a particular method of modeling light transport along the ray. It merely assumes that it is possible to determine the color and density of the medium at any point, and that we have access to a function for integrating this information along each ray to determine its contribution in terms of opacity and color.

6.2 Construction of the SD-tree

The smoke volume is defined by an axis-aligned bounding box, and the data structure stores the attributes associated with some set of sample rays that intersect the volume. Recall from Section 4.3.1 that space of rays intersecting an axis aligned bounding box can be parameterized as points in 4-dimensional space by using 6 plane-pairs, each of which is associated with a 4-dimensional hypercube in line space containing all rays that pass through it. Recall from the SD-tree description that an hypercube is initially subdivided into $4! = 24$ coarse simplices which are then recursively bisected. Each coarse simplex is the root of a separate binary tree, which are conceptually joined under a common super-root corresponding to the hypercube. Hence, the data structure built for a single smoke volume consists of 6 such SD-trees one for each plane-pair. From this point on, we will use the term *SD-tree* to refer to the collection of these 6 trees built for a volume. A 4-dimensional simplex has 5 vertices, which is the minimum number of points required for linear interpolation in 4-dimensional space. The 5 vertices of a simplicial leaf cell in

SD-tree constitute the ray samples which form the basis of our interpolation.

Just like the RI-tree, the SD-tree grows and shrinks dynamically based on demand. Initially, only the 16 corners of each hypercube are sampled, and the initial 24 coarse simplices are constructed. A leaf simplex is subdivided by bisection along its longest edge, by sampling the midpoint of that edge. To determine whether to subdivide the leaf cell or not, we use the following *termination conditions*:

Degree of Variation: We use a heuristic that defines the degree of variation $\mathcal{V}(S)$ associated with a leaf simplex S as the maximum distance between the values of any two distinct vertices of S:

$$\mathcal{V}(S) = \max\{d(f(v_i), f(v_j)) \mid 0 \leq i < j \leq 4\},$$

where v_0,...,v_4 are the vertices of S. Here, $f(v_i)$ denotes the correct value of the function at v_i computed by ray marching. For the smoke volume application, the distance d is a weighted distance of color and opacity.

Pixel Resolution versus Depth Constraint: The SD-tree could be allowed to grow until pixel resolution (i.e. projected leaf simplex width is less than the pixel width), or, in order to avoid excessive growth at strong discontinuity regions, the user may specify a *depth constraint*, such that the tree is not allowed to grow beyond that depth. If the subdivision is stopped due to the depth constraint, though, that leaf is not used for interpolation.

Consequently, if $\mathcal{V}(S)$ exceeds a user-defined *distance threshold* and the depth of the cell in the tree is less than a user-defined *depth constraint*(or pixel resolution is not reached), the cell is subdivided. Otherwise, the leaf is said to be *final*.

6.3 Rendering by Interpolation

In order to interpolate the color and opacity for a given input ray **r**, we first locate the leaf simplex containing **r** within the tree corresponding to the appropriate hypercube depending on the dominant direction of **r**. Along with the search, we also incrementally compute barycentric coordinates of **r** with respect to the leaf simplex. This process is described in Section 5.4.2 in detail. Recall that, due to on-demand construction, the nodes on the path to the *final* leaf containing **r** may be constructed along with this process, if they have not been already constructed. The color and opacity for **r** can now be interpolated by barycentric interpolation of the values associated with the 5 vertices of this *final* leaf simplex.

However, other practical issues arise when building on-demand simplicial decompositions for efficient rendering purposes. In this section, we discuss these issues.

6.3.1 One-pass versus Two-pass Rendering

Notice that, even though the final tree constructed is compatible, this method does not totally avoid cracks in interpolation if the rendering and construction are done in the same single pass. Consider the two dimensional analogy in Figure 6.1. If q_1 arrives before q_2, there is no problem, since splitting of S_1 will force S_2 to split, and when q_2 arrives, the simplices will be compatible. However, assume that q_2 arrives before q_1 and that S_2 satisfies the termination condition, and marked as final. When q_2 arrives, it is answered by interpolation of the vertices of S_2. Then, when q_1 arrives, assume that the subdivision in the figure occurs splitting S_1 two more levels. Thus, q_1 is answered by interpolating the

vertices of a grandchild of S_1. However, since q_2 is already answered at this point, there would be a crack in the interpolation, even though S_2 is forced to split by the split of S_1.

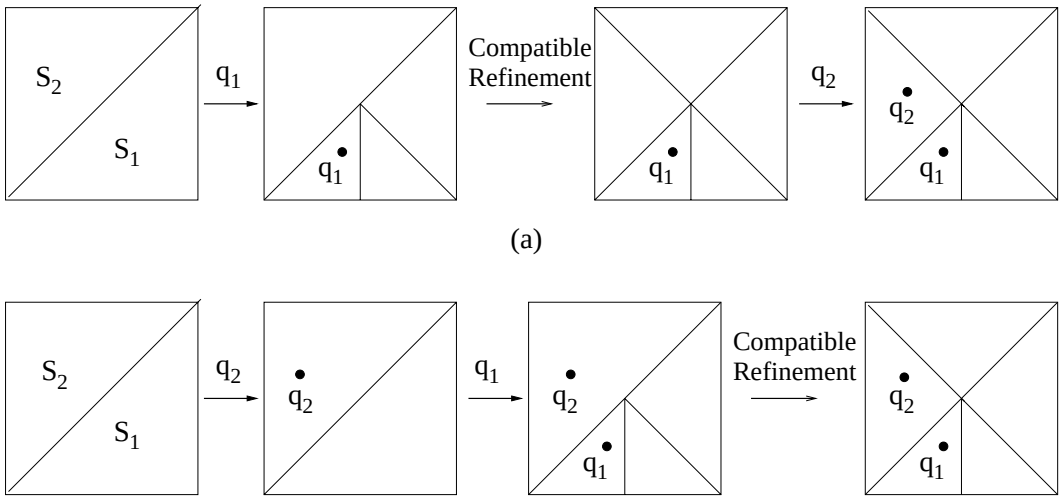


Figure 6.1: Compatible refinement (a) q_1 arrives first (b) q_2 arrives first

To avoid this, we have to render in two passes. In the first pass, the tree is constructed given all the query points, but without doing the interpolations. In the second pass, the queries are answered by performing the interpolations. Obviously, the two-pass rendering will be slightly more expensive, since the point location procedure will be done twice. Alternatively, for a smoke volume application, an auxiliary data structure can be used to keep a pointer to the leaf simplex located in the first pass associated with each pixel, and so, point location can start from this leaf simplex instead of the root in the second pass.

6.3.2 On-demand Compatible Refinement

Note that, there is a conflict between on-demand construction and compatible refinement for our purposes. To preserve compatibility, some simplices in the hierarchy will be refined, even though they will not be used for any interpolation query eventually. Thus, a lot of work done for construction of those simplices will be useless, unnecessarily reducing the efficiency of the overall algorithm, and increasing the size of the data structure. To prevent this, while keeping the compatibility property, we perform *on-demand compatible refinement*, which works as follows. The bisection of a simplex S does not trigger the bisection of a neighboring simplex, before that neighbor is actually required by some interpolation. Consider Figure 6.2. S_1 and S_2 are neighbors of each other at the same level. Let the query point q_1 cause refinement of S_1 as shown. At the time q_1 caused this refinement, S_2 is not bisected to provide compatibility. Unless another query needs S_2, the tree will remain non-compatible in fact, but still compatible for our purposes. However, if later, a query q_2 is located in S_2, before any termination condition is checked, we first check whether any neighbor of S_2 is refined by bisecting an edge shared by S_2 (even if S_2 is already marked as a final leaf, this check is performed, and might cause splitting of the final leaf). In Figure 6.2, such a neighbor exists, that is S_1. Hence, S_2 will be bisected as well, and q_2 will continue its descent in the tree until no more splits are required, before being interpolated.

This method is much more efficient, since it generates a much smaller tree. But, for similar reasons with the original compatible refinement, it cannot avoid cracks totally (See Figure 6.3). In this case, two-pass rendering corrects a substantial percentage of the

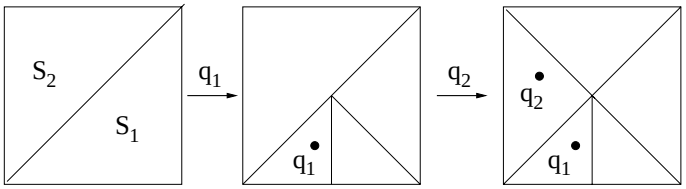


Figure 6.2: On-demand compatible refinement

cracks, but may not eliminate all the cracks. (Unlike the two-pass rendering explained in Section 6.3.1, the second pass as well, will induce subdivisions in the tree to correct the cracks.) Experimentally, we have seen that, among the final leaf nodes, less than 5% have cracks, and that the on-demand version performs comparably well with respect to the quality of the image generated. Moreover, a two pass approach similar to the one explained above, reduces the number of cracks substantially. In fact, after a number of passes, the tree will converge to a crack-free tree.

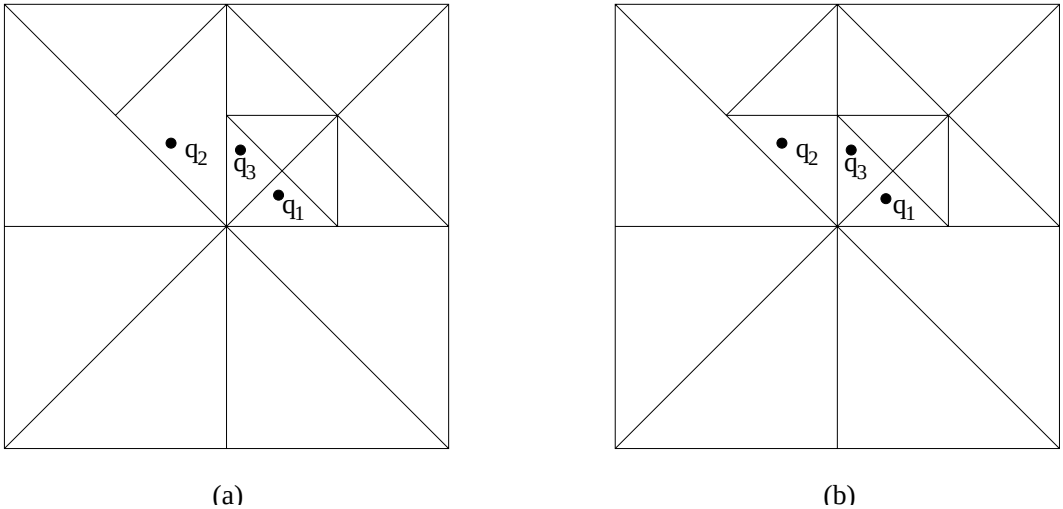


Figure 6.3: On-demand compatible refinement in multiple passes, queries arrive in the order of q_1, q_2 and q_3 in both passes. (a) First-pass (b) Second pass corrects the crack between the cells of q_2 and q_3.

6.4 Experimental Results

In order to establish the space and time efficiency, and accuracy of the two-pass and on-demand methods, we ran a number of experiments. For our experiments, we applied a simple light model, which accounts for extinction of light due to absorption by particles (opacity) and for the addition of light by reflection of external illumination. We have adapted the smoke volume shader code given in "PhotoRealistic RenderMan Application Note 20-Writing Fancy Volume Shaders"[PRM] to our own ray-tracer. The general idea is to ray march along the viewing ray choosing an appropriate step size, sampling illumination and accounting for atmospheric extinction based on smoke density at every portion of the ray. The smoke density at any point is determined by a noise function. This type of volume shaders that are used by renderers like PRMan or BMRT [GH96] are very expensive, since reasonably small step sizes have to be chosen to avoid banding artifacts.

In general, this type of volume shaders must bind to surfaces, that is, there should be an object in the background, so that, the ray marching continues until the background object is hit. We model the smoke density as a finite volume, defined by an axis-aligned bounding box. The viewing ray enters the volume and the integration continues until the ray exits the volume (or hits an object that is within the volume). For simplicity, we have assumed that the smoke volume is designed to extend up to the background objects, and does not include any objects inside.

We have modeled the interior of a warehouse, with a number of windows letting sunlight in. The smoke volume covers the interior, extending from the left wall to the right wall, from the floor to the ceiling and from the back wall to the viewpoint. The

viewpoint is slightly outside the volume. The step size we picked is 0.3 units (the shortest distance from the viewpoint to the back plane is 100 units). For smoke, we assume that all wavelengths are subject to same amount of scattering (color and opacity values have equal red, green, and blue components), thus, we store color and opacity as scalars. We have rendered images of size 800×600 anti-aliased (9 rays per pixel are shot.)

We investigated the speedup and actual error committed by the interpolation algorithm, as well as the number of ray samples required, and the percentage of cracks in the data structure for the on-demand compatible refinement algorithm. Speedup is the ratio of the CPU-time for the traditional ray marching approach to the CPU-time for our interpolation algorithm. The error committed by the interpolation algorithm is measured as the average distance between the actual color and opacity, and the corresponding quantity for the interpolated case. We also report the maximum error committed among all the rays shot. The color and opacity values are normalized to the range [0,1]. For our test scene, the actual color values are in the range [0, 0.2953], and the opacity values are in the range [0,0.5045]. Average color is 0.05419 and the average opacity is 0.1249. Figure 6.4(a), (b) and (c) demonstrate how the variation in error reflects the change in the quality of the rendered image. Notice the artifacts in (b) and (c) when the data structure is not subdivided as densely as in (a).

The percentage of cracks is given both in terms of the percentage of the final leaves (the leaves used for interpolation) that have cracks, and the percentage of the rays that are interpolated using the leaves with cracks.

The number of ray samples is the number of rays that are sampled during the construction of the data structure at simplex vertices. Sampling is the dominating cost. For

(a) (b) (c)

Figure 6.4: Given errors are with respect to color. (a) distance thr = 0.015, average error = 0.00233, max error = 0.02704. (b) distance thr = 0.035, average error = 0.00371, max error = 0.06754. (c) distance thr = 0.05, average error = 0.00545, max error = 0.13001.

example, for the compatible, two-pass rendering, the first pass during which the construction is done takes 90% of the total time, while the second pass takes only 10% of the total time to do point location and interpolation. For more expensive smoke rendering models, or for smaller step sizes, the cost of sampling will be even more dominant, since the time taken by interpolation and point location will remain almost constant. Hence, the speedup is bounded by the ratio of the total number of rays shot while rendering by ray marching to the number of sample rays generated while rendering by the interpolation algorithm. This suggests that, for higher resolution images, the speedups will be much higher.

Table 6.1 shows sample results for rendering the image by the compatible, two-pass method, and by the on-demand compatible algorithms. We used a distance threshold of 0.015 which was found to perform well experimentally. Recall that the distance threshold, described in Section 6.2, is used to determine whether to terminate a subdivision process. The on-demand compatible algorithm performs as well as the compatible, two-pass algo-

rithm in terms of quality, while sampling 69% fewer rays and creating 92% fewer nodes. Therefore, the on-demand compatible algorithm achieves a significant speedup of 18.24, which is 3 times the speedup achieved by the compatible, two-pass method. Even the speedup of 6.2 for the compatible, two-pass method is significant for expensive applications like this one. Corresponding images are given in Figure 6.5. Part (a) shows the correct image generated by marching all rays, and part (b) shows the interpolated image generated using the on-demand compatible algorithm. (Since the on-demand compatible algorithm generates almost the same image as the compatible two-pass algorithm, we show only the image generated by the on-demand version.)

Algorithm	Speedup	Error (color)		Error(opacity)		#Rays	Size
		average	max	average	max		
Ray-marching	1	0	0	0	0	4,320,000	-
Compat., two-pass	6.20	0.00230	0.02704	0.00396	0.04163	334,438	354MB
On-demand comp.	18.24	0.00233	0.02704	0.00401	0.04163	101,605	38MB

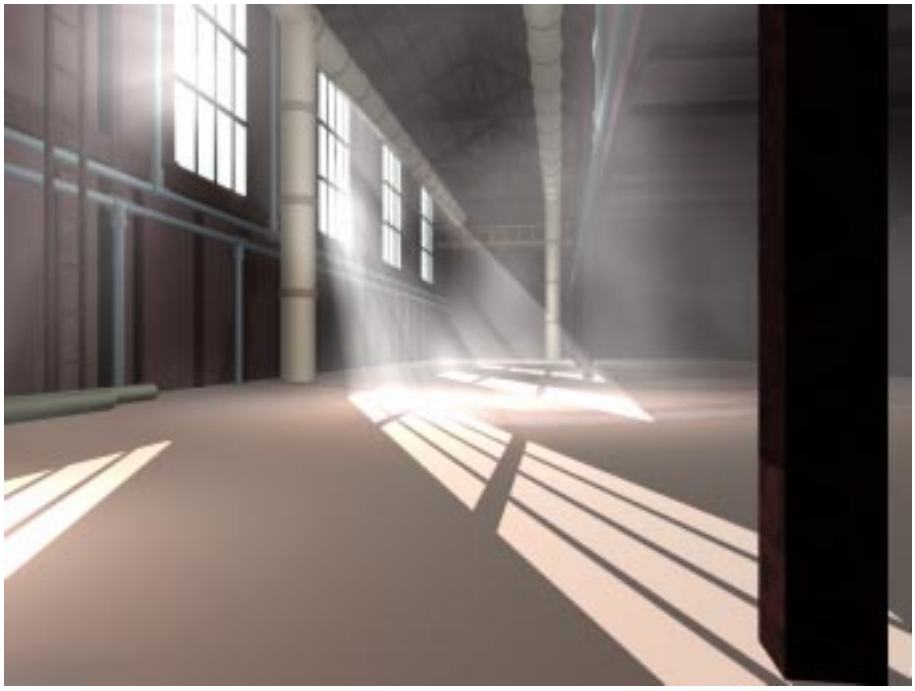
Table 6.1: Sample results for the warehouse scene (800×600 anti-aliased, distance threshold = 0.015).

If the on-demand compatible algorithm is used to render in multiple passes as explained in Section 6.3.2, the percentage of cracks is reduced substantially as shown in Table 6.2, but of course reducing the speedup.

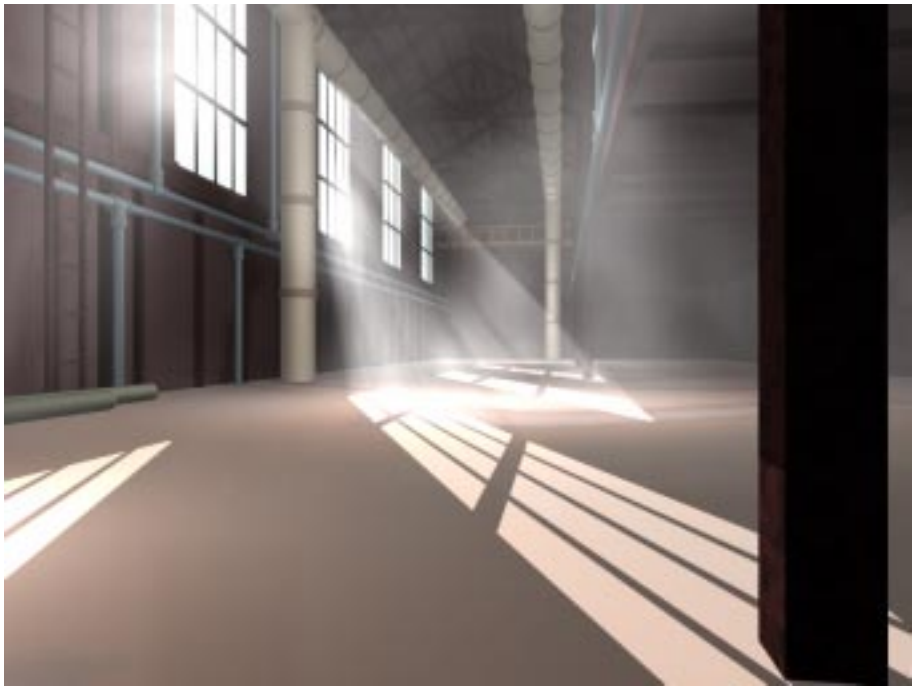
If desired, higher quality approximations can be rendered by lowering the distance threshold, at the potential expense of performance.

Algorithm	%Cracks		Speedup
	%leaf nodes	%rays	
On-demand compatible, one-pass	2.494	4.273	18.24
On-demand compatible, two-pass	0.204	0.489	13.43
On-demand compatible, three-pass	0.041	0.055	10.66

Table 6.2: The percentage of cracks for multiple passes of the on-demand compatible algorithm.



(a)



(b)

Figure 6.5: (a) Ray-marched image (b) Interpolated image using the on-demand compatible algorithm (800x600, anti-aliased, distance threshold = 0.015).

Chapter 7

Conclusions

We conclude this dissertation by summarizing our contributions and outlining a number of possible directions for future work.

7.1 Summary of Contributions

The Ray Interpolant Tree: We introduced the RI-tree data structure and showed that it can produce high-quality renderings significantly faster than ray-tracing by storing an adaptively sampled set of rays, and using inexpensive interpolation methods to approximate the attribute values for new input rays. Our approach of sampling and interpolating geometric attributes rather than radiance allows decoupling of an object from the rest of the scene geometry and illumination. The RI-tree is most useful for rendering smooth objects that are reflective or transparent, for rendering animations when the viewpoint varies smoothly or when the illumination varies from frame to frame, and for generating high resolution images.

The Simplex Decomposition Tree: Next, we introduced the SD-tree data structure which improves the functionality of the RI-tree by ensuring continuity of the interpolating surface. We adapted a subdivision scheme based on bisection due to Maubach [Mau95]. We presented efficient incremental methods for performing point location and computing the weights needed in interpolation. Compared to the RI-tree, the SD-tree is much simpler and more efficient for interpolation purposes, since linear interpolations are performed with minimal number of samples. We also observed that better quality images can be generated with fewer samples with the SD-tree, compared to the RI-tree. This is mostly due to the refinement method, which avoids cracks, and partly because the same level of refinement is achieved with fewer samples in a simplicial subdivision.

Pointerless Representation of d-dimensional Hierarchical Regular Simplicial Mesh: Another major contribution of this thesis is the development of a pointerless representation for hierarchical regular simplicial meshes. We introduced the LPT code, that uniquely encodes the simplices of the hierarchy and is used to access a node in the hierarchy in constant time. We addressed algorithmic issues in efficient implementation of the tree operations based on the LPT code and showed that all traversal operations can be performed in constant time. The space savings realized by not having to store pointers and simplex vertices is significant for large multidimensional meshes. We believe that this representation may find numerous applications in areas where d-dimensional hierarchical regular simplicial meshes are used.

Efficient Neighbor Computation for d-dimensional Simplicial Meshes: We introduced a compact set of neighbor rules to compute equal-depth neighbors of a simplex directly form 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 a significant gain both in terms of storage, and computational efficiency, since our approach is local and runs in constant time. We proved correctness of our neighbor finding rules. In addition to the same-depth neighbors, we presented rules to compute the neighbors that can possibly appear at other depths in compatible subdivisions.

Two-pass Rendering and On-demand Compatible Refinement: We have also demonstrated the use of a 4-dimensional SD-tree for accelerating rendering of smoke through ray marching. Within this context, first we observed that in order to avoid cracks entirely, the rendering has to be done in two passes. Experimentally we have seen that the overhead of the second pass is tolerable, since the total cost is dominated by the cost of sampling in the first pass.

Next we observed that full compatibility and on-demand construction conflict in the sense that some portions of the data structure built due to compatible refinement are never used for queries. Instead, we proposed *on-demand compatible refinement*, which aims to provide compatibility only for those simplices that are needed for interpolation. This approach generates a data structure of much smaller size—for our test scene %89 smaller— compared to the fully compatible version. More importantly, it achieved a speedup of 18.24, which is 3 times the speedup achieved by the fully-compatible method. Even though it cannot avoid cracks entirely, we have seen experimentally that a very small

fraction of the final leaf nodes have cracks, and that the on-demand version performs comparably well with respect to the quality of the image generated.

7.2 Future Work

Bounds on error of approximation: In our current methods, we use heuristics to determine the accuracy of the interpolation. Experimentally, these heuristics are shown to work well in most cases, but the interpolation errors are not bounded. So, for some cases arbitrary approximation errors could arise. It would be desirable to extend our methods to provide theoretical bounds on the error introduced by interpolation. We have the option of imposing conservative error bounds at the expense of lower performance. Bounding error is likely to be easier for the smoke rendering application, since there is usually a well-defined function for computing the color and opacity at a certain point.

Dealing with aliasing and improving animations: It is very desirable to investigate how we can improve the quality of animations by examining ways for smoother transition between frames. For example, how can we avoid flickering due to temporal aliasing and variation in interpolation which result from different viewpoints.

Interpolation in temporal domains: A direct and interesting extention of our current methods would be to consider how to apply our data structures for time varying physical phenomena, for example, for rendering simulations of smoke or clouds over time. This would require using 5-dimensional extentions of our data structures, and considering that complexities of subdivisions tend to increase exponentially with dimension, it is not read-

ily clear whether interpolation methods would still be beneficial. It would be interesting to see how the performance would be affected.

Application of neighbor rules in visualization: We believe, our neighbor rules would be very useful for efficient visualization of high-dimensional fields—especially dimensions greater or equal to four. (Many visualization algorithms require moving between adjacent simplices rapidly.) This arises as an important problem with the emergence of time-varying fields with various applications in medicine, computational fluid dynamics (CFD) and molecular dynamics.

Different subdivision schemes: Our current work on hierarchical regular simplicial meshes has led us to several related issues, including pointerless representations and cache-sensitive data structures, efficient neighbor finding and different subdivision techniques, independently of the interpolation problem. Along these lines, it would be interesting to study other subdivision methods and develop labeling and neighbor finding rules.

Bibliography

[AB91] E. Adelson and J. Bergen. The plenoptic function and the elements of early vision. *Computational Models of Visual Processing*, pages 1–20, 1991.

[ABCC02] B. Aronov, H. Bronnimann, A. Y. Chang, and Y. Chiang. Cost prediction for ray shooting. In *Proc. 18th ACM Symp. on Comput. Geom.(SoCG'02)*, pages 293–302, 2002.

[ABCC03] B. Aronov, H. Bronnimann, A. Y. Chang, and Y. Chiang. Cost-driven octree consruction schemes: An experimental study. In *Proc. 19th ACM Symp. Comput. Geom. (SoCG'03)*, pages 227–236, 2003.

[AF99] B. Aronov and S. Fortune. Approximating minimum weight triangulations in three dimensions. *Discrete Comput. Geom.*, 21(4):527–549, 1999.

[AG79] E. Allgower and K. Georg. Generation of triangulations by reflection. *Utilitas Mathematica*, 16:123–129, 1979.

[AH95] S. J. Adelson and L. F. Hodges. Generating exact ray-traced animation frames by reprojection. *IEEE Comp. Graph. and Appl.*, 15(3):43–52, May 1995.

[AK87] J. Arvo and D. Kirk. Fast ray tracing by ray classification. *Computer Graphics (Proc. of SIGGRAPH 87)*, 21(4):196–205, 1987.

[AK89] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In A.S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201–262. Academic Press, San Diego, 1989.

[Alf89] P. Alfeld. Scattered data interpolation in three or more variables. In T. Lyche and L. L. Schumaker, editors, *Mathematical Methods in Computer Aided Geometric Design*, pages 1–34. Academic Press, 1989.

[AM02] F. B. Atalay and D. M. Mount. Ray interpolants for fast ray-tracing relections and refractions. *J. of WSCG*, 10(3):1–8, 2002. Proc. Int. Conf. in Central Europe on Comp. Graph., Visual. and Comp. Vision.

[AM03] F. B. Atalay and D. M. Mount. Interpolation over light fields with applications in computer graphics. In *Proc. of the 5th Workshop on Algorithm Engineering and Experiments (ALENEX 2003)*, pages 56–68. SIAM, 2003.

[AM04a] F. B. Atalay and D. M. Mount. Pointerless implementation of hierarchical simplicial meshes and efficient neighbor finding in arbitrary dimensions. To appear in Proc. International Meshing Roundable (IMR 2004), 2004.

[AM04b] F. B. Atalay and D. M. Mount. Pointerless implementation of hierarchical simplicial meshes and efficient neighbor finding in arbitrary dimensions. Technical Report CS-TR-4586/UMIACS-TR-2004-29, University of Maryland, College Park, 2004.

[Ama84] J. Amanatides. Ray tracing with cones. *Computer Graphics (Proc. of SIGGRAPH 84)*, 18(3):129–135, 1984.

[AML01] D.N. Arnold, A. Mukherjee, and L.Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM J. Sci. Comput.*, 22(2):431–448, 2001.

[Arg02] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[AV88] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[Bad88] J. S. Badt. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4(3):123–132, September 1988.

[Bal99] K. Bala. *Radiance Interpolants for Interactive Scene Editing and Ray Tracing*. PhD thesis, Massachusetts Institute of Technology, 1999.

[Ban91] E. Bansch. Local mesh refinement in 2 and 3 dimensions. *Impact of Computing in Science and Engineering*, 3:181–191, 1991.

[Ban98] R.E. Bank. Pltmg: A software package for solving elliptic partial differential equations, user's guide 8.0. *Software, Environments and Tools*, 5, 1998.

[BDT99] K. Bala, J. Dorsey, and S. Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Trans. on Graph.*, 18(3), August 1999.

[BEG94] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *J. Comput. Syst. Sci.*, 48:384–409, 1994.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. of ACM*, 18(9):509–517, 1975.

[Bey95] J. Bey. Tetrahedral grid refinement. *Computing*, 55:355–378, 1995.

[Bey00] J. Bey. Simplicial grid refinement: On freudenthal's algorithm and the optimal number of congruence classes. *Numer. Math.*, 85(1), 2000.

[Blo97] J. Bloomenthal. *An Introduction to Implicit Surfaces*. Morgan-Kaufmann, San Francisco, 1997.

[BN76] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Commun. of ACM*, 19:542–546, 1976.

[BR98] U. Behrens and R. Ratering. Adding shadows to a texture-based volume renderer. In *1998 Volume Visualization Symposium*, pages 39–46, 1998.

[BSW83] R.E. Bank, A.H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing*, pages 3–17, 1983.

[Bur88] P. J. Burt. Moment images, polynomial fit filters, and the problem of surface interpolation. In *Computer Vision and Pattern Recognition*, pages 144–152, 1988.

[CBL99] C. Chang, G. Bishop, and A. Lastra. LDI tree: A hierarchical representation for image-based rendering. *Computer Graphics (Proc. of SIGGRAPH 99)*, pages 291–298, 1999.

[CCD91] J. Chapman, T. W. Calvert, and J. C. Dill. Spatio-temporal coherence in ray tracing. In *Proc. of Graphics Interface '91*, pages 101–108, June 1991.

[CDM+03] P. Cignoni, L. De Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 2003. (in print).

[CDP95] F. Cazals, G. Drettakis, and C. Puech. Filtering, clustering and hierarchy construction: A new solution for ray-tracing complex scenes. *Computer Graphics Forum*, 14(3):371–382, 1995.

[Cha60] S. Chandrasekhar. *Radiative Transfer*. Dover, New York, 1960.

[Che95] S. E. Chen. QuickTime VR — an image-based approach to virtual environment navigation. *Computer Graphics (Proc. of SIGGRAPH 95)*, 29:29–38, 1995.

[Chi99] T. M. Chilimbi. *Cache-Conscious Data Structures*. PhD thesis, Computer Sciences Dept., University of Wisconsin-Madison, 1999.

[CHL99] T. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Programming Languages Design and Implementation*, 1999.

[CLF98] E. Camahort, A. Lerios, and D. Fussell. Uniformly sampled light fields. *Rendering Techniques '98 (9th Eurographics Workshop on Rendering)*, pages 117–130, 1998.

[CP97] F. Cazals and C. Puech. Bucket-like space partitioning data structures with applications to ray tracing. In *Proc. 13th ACM Symp. Comput. Geom. (SoCG'97)*, pages 11–20, 1997.

[CPC84] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *Computer Graphics (Proc. of SIGRAPH 84)*, 18(3):137–145, July 1984.

[CRMT91] S. E. Chen, H. E. Rushmeier, G. Miller, and D. Turner. A progressive multi-pass method for global illumination. *Computer Graphics (Proc. of SIGGRAPH 91)*, 25(4):165–174, 1991.

[CW93] S. E. Chen and L. Williams. View interpolation for image synthesis. *Computer Graphics (Proc. of SIGGRAPH 93)*, 27:279–288, 1993.

[DB94] P. Diefenbach and N. Badler. Pipeline rendering: Interactive refractions, reflections and shadows. *Displays: Special Issue on Interactive Computer Graphics*, 15(3):173–180, 1994.

[dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[Dem02] E. D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, 2002.

[dKL02] J. B. Van de Kamer and J. J. W. Lagendijk. Computation of high-resolution SAR distributions in a head due to a radiating dipole antenna representing a hand-held mobile phone. *Physics in Medicine and Biology*, 47:1827–1835, 2002.

[DKP03] L. De Floriani, L. Kobbelt, and E. Puppo. A survey on data structures for level-of-detail models. 2003.

[DKW85] N. Dadoun, D. G. Kirkpatrick, and J. P. Walsh. The geometry of beam tracing. In *Proc. of 1st Annual ACM Symp. Comput. Geom.*, pages 55–61, 1985.

[DKY+00] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, and T. Nishita. A simple, efficient method for realistic animation of clouds. In *Proc. of SIGGRAPH 2000*, pages 19–28, 2000.

[DM02] L. De Floriani and P. Magillo. Multiresolution mesh representation: Models and data structures. *Principles of Multiresolution in Geometric Modeling*, 2002.

[DMMP00] L. De Floriani, P. Magillo, F. Morando, and E. Puppo. Dynamic view-dependent multiresolution on a client-server architecture. *Computer-Aided Design Journal (Special Issue on Multiresolution Geometric Models)*, 32(13):805–823, 2000.

[DWS+97] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein. Roaming terain: Real-time optimally adapting meshes. pages 81–88, 1997.

[DYN02] Y. Dobashi, T. Yamamoto, and T. Nishita. Interactive rendering of atmospheric scattering effects using graphics hardware. In *Graphics Hardware 2002*, pages 99–108, 2002.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.

[EKT01] W. Evans, D. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica.*, 30(2):264–286, 2001.

[EMP+98] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modelling*. Academic Press Professional, San Diego, 1998.

[FB74] R.A. Finkel and J.L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[Fek90] G. Fekete. Rendering and managing spherical data with spherical quadtrees. In *Proc IEEE Visualization 90*, pages 176–186, 1990.

[FKN80] H. Fuchs, M. Kedem, and B.F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (Proc. of SIGGRAPH 80*, 14(3):124–133, 1980.

[FLPR99] M. Frigo, C. B. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symp. on Found. of Comput. Sci.*, pages 285–297, 1999.

[FM96] N. Foster and D. Metaxas. Realistic animation of liquids. In *Graphics Interface '96*, pages 204–212, 1996.

[FM97a] N. Foster and D. Metaxas. Controlling fluid animation. In *Computer Graphics International 1997*, 1997.

[FM97b] N. Foster and D. Metaxas. Modeling the motion of a hot, turbulent gas. In *Proc. of SIGGRAPH 97*, pages 181–188, 1997.

[Fre42] H. Freudenthal. Simplizialzerlegungen von beschrankter flachheit. *Annals of Math.*, 43:580–582, 1942.

[FSJ01] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In *Proc. of SIGGRAPH 2001*, pages 15–22, 2001.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray tracing system. *IEEE Comp. Graph. and Appl.*, 6(4):16–26, April 1986.

[FvDFH90] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics Principles and Practice*. Addison-Wesley, Reading, Mass., 1990.

[Gar82] I. Gargantini. An effective way to represent quad-trees. *Commun. ACM*, 25(12):905–910, 1982.

[GD] J. P. Grossman and W. J. Dally. Point sample rendering. In *Rendering Techniques '98 (9th Eurographics Workshop on Rendering)*, pages 181–192.

[GDL+02] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proc. IEEE Visualization 2002*, 2002.

[Ger39] A. Gershun. The light field. *Journal of Mathematics and Physics*, XVIII:51–151, 1939. Moscow, 1936, Translated by P. Moon and G. Timoshenko.

[Ger03] T. Gerstner. Multiresolution visualization and compression of global topographic data. *GeoInformatica*, 7(1):7–32, 2003.

[GGC97] X. Gu, S. J. Gortler, and M. F. Cohen. Polyhedral geometry and the two-plane parameterization. In *Rendering Techniques '97 (8th Eurographics Workshop on Rendering)*, pages 1–12, 1997.

[GGSC96] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 43–54, August 1996.

[GH96] L. Gritz and J. Hahn. Bmrt: A global illumination implementation of the renderman standard. *J. Graphics Tools*, 1(3):29–47, 1996.

[GI97] T. Gutzmer and A. Iske. Detection of discontinuities in scattered data approximation. *Numerical Algorithms*, 16(2):155–170, 1997.

[Gla84] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Comp. Graph. and Appl.*, 4(10):15–22, October 1984.

[Gla89a] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press, San Diego, 1989.

[Gla89b] A. S. Glassner. An overview of ray tracing. In A.S. Glassner, editor, *An Introduction to Ray Tracing*, pages 1–32. Academic Press, San Diego, 1989.

[Gla95] A. S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, New York, 1995.

[GLE97] R. Grosso, C. Lurig, and T. Ertl. The multilevel finite element method for adaptive mesh optimization and visualization of volume data. In *Proc. Visualization'97*, 1997.

[GR99] T. Gerstner and M. Rumpf. Multiresolutional parallel isosurface exraction based on tetrahedral bisection. In *Proc. Symp. Volume Visualization*, 1999.

[GS87] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comp. Graph. and Appl.*, 7(5):14–20, May 1987.

[GS92] M.F. Goodchild and Y. Shiren. A hierarchical spatial data structure for global geographic information systems. *CVGIP: Graph. Models and Image Processing*, 54(1):31–44, 1992.

[GTGB84] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics (Proc. of SIGGRAPH 84)*, pages 213–222, 1984.

[Guo98] B. Guo. Progressive radiance evaluation using directional coherence maps. *Computer Graphics (Proc. of SIGGRAPH 98)*, 32:255–266, 1998.

[HDD+92] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics (Proc. of SIGGRAPH 92)*, 26(2):71–78, 1992.

[Heb94] D. J. Hebert. Symbolic local refinement of tetrahedral grids. *J. of Symbolic Comput.*, 17:457–472, 1994.

[HH84] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. *Computer Graphics (Proc. of SIGGRAPH 84)*, 18(3):119–127, July 1984.

[HLCS99] W. Heidrich, H. Lensch, M. Cohen, and H. Seidel. Light field techniques for reflections and refractions. In *10th Eurographics Rendering Workshop*, June 1999.

[Hop96] H. Hoppe. Progressive meshes. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 99–108, 1996.

[HS01] Z. Hakura and J. Snyder. Realistic reflections and refractions on graphics hardware with hybrid rendering and layered environment maps. In *Proc. 12th Eurographics Workshop on Rendering Techniques*, pages 289–300, 2001.

[ICG86] D. S. Immel, M. F. Cohen, and D. P. Greenberg. A radiosity method for non-diffuse environments. *Computer Graphics (Proc. of SIGGRAPH 86)*, 20(4):133–142, 1986.

[JC95] H. W. Jensen and N. J. Christensen. Efficiently rendering shadows using the photon map. In *Proc. of Compugraphics*, pages 285–291, 1995.

[JC98] H. W. Jensen and P. H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *Proc. of SIGGRAPH 98*, pages 311–320, 1998.

[Jen95] H. W. Jensen. Importance driven path tracing using the photon map. In *Rendering Techniques '95 (6th Eurographics Workshop on Rendering)*, pages 326–335, 1995.

[Jen96] H. W. Jensen. Global illumination using photon maps. In *Rendering Techniques '96 (7th Eurographics Workshop on Rendering)*, pages 21–30, 1996.

[Jen97] H. W. Jensen. Rendering caustics on non-Lambertian surfaces. *Computer Graphics Forum*, 16(1):57–64, 1997.

[Kaj86] J. T. Kajiya. The rendering equation. *Computer Graphics (Proc. of SIGGRAPH 86)*, 20(4):143–150, August 1986.

[Kap85] M. R. Kaplan. Space tracing a constant time ray tracer. *State of the Art in Image Synthesis (SIGGRAPH 85 Course Notes)*, 11, July 1985.

[Kap87] M. R. Kaplan. The use of spatial coherence in ray tracing. *Techniques for Computer Graphics*, pages 173–193, 1987.

[KH84] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. In *Computer Graphics (Proc. of SIGGRAPH 84)*, volume 18, pages 165–174, 1984.

[KK86] T. L. Kay and J. Kajiya. Ray tracing complex scenes. *Computer Graphics (Proc. of SIGGRAPH 86)*, 20(4):269–278, August 1986.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[Kru90] W. Krueger. The application of transport theory to visualization of 3d scalar fields. In *Proc. IEEE Visualization '90*, pages 273–280, 1990.

[Lar98] G. W. Larson. The holodeck: A parallel ray-caching rendering system. In *2nd Eurographics Workshop on Parallel Graphics and Visualisation*, September 1998.

[LDS01] M. Lee, L. De Floriani, and H. Samet. Constant-time neighbor finding in hierarchical tetrahedral meshes. In *Proc. Int. Conf. on Shape Modelling*, pages 286–295, 2001.

[LF94] S. Laveau and O. Faugeras. 3d scene representation as a collection of images. In *Twelfth International Conference on Pattern Recognition*, pages 689–691, 1994.

[LG95] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proc. SIGGRAPH Symposium on Interactive 3D Graphics*, pages 105–106, 1995.

[LH96] M. Levoy and P. Hanrahan. Light field rendering. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 31–42, August 1996.

[LHJ99] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proc. IEEE Visualization '99*, pages 355–362, 1999.

[LJ94a] A. Liu and B. Joe. On the shape of tetrahedra from bisection. *Math. Comp.*, 63:141–154, 1994.

[LJ94b] A. Liu and B. Joe. Relationship between tetrahedron shape measures. *BIT*, 34:268–287, 1994.

[LJ95] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM J. Sci. Comput.*, 16:1269–1291, 1995.

[LJ96] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision. *Math. of Comput.*, 65(215):1183–1200, 1996.

[LKR+96] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, and G.A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proc. of SIGGRAPH 96*, pages 109–118, 1996.

[LR98] D. Lischinski and A. Rappoport. Image-based rendering for non-diffuse synthetic scenes. *9th Eurographics Workshop on Rendering*, pages 301–314, 1998.

[LS00] M. Lee and H. Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Trans. on Computer Graphics*, 19:79–121, 2000.

[LW85] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report TR-85-022, Computer Science Department, University of North Carolina at Chapel Hill, 1985.

[LW93] E. P. Lafortune and Y. D. Willems. Bi-directional path tracing. In *Proc. of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, 1993.

[Mau95] J. M. Maubach. Local bisection refinement for N-simplicial grids generated by reflection. *SIAM J. Sci. Stat. Comput.*, 16:210–227, 1995.

[Mau96] J. M. Maubach. The efficient location of neighbors for locally refined n-simplicial grids. In *5th Int. Meshing Roundable*, 1996.

[Max86] N. Max. Atmospheric illumination and shadows. *Computer Graphics(Proc. of SIGGRAPH 86)*, 20(4):117–124, 1986.

[Max95] N. Max. Optical models for direct volume rendering. *IEEE Trans. on Visualization and Comp. Graph.*, 1(2):99–108, 1995.

[MB95] L. McMillan and G. Bishop. Plenoptic modeling. *Computer Graphics (Proc. of SIGGRAPH 95)*, pages 39–46, 1995.

[MF53] P. M. Morse and H. Feshbach. *Methods of Theoretical Physics, Part I*. McGraw-Hill, New York, 1953.

[MG99] L. McMillan and S. Gortler. Image-based rendering: A new interface between computer vision and computer graphics. *Computer Graphics*, 33(4):61–64, November 1999.

[Mit87] D. P. Mitchell. Generating antialiased images at low sampling densities. *Computer Graphics (Proc. of SIGGRAPH 87)*, 21(4):65–72, 1987.

[Mit88] W. F. Mitchell. *Unified multilevel adaptive finite element methods for elliptic problems.* PhD thesis, UIUCDCS-R-88-1436, Dept. of Computer Science, Univ. of Illinois, Urbana, IL, 1988.

[Mit91] W. F. Mitchell. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *Journal of Computational and Applied Mathematics*, 36:65–78, 1991.

[Mit92] W. F. Mitchell. Optimal multilevel iterative methods for adaptive grids. *SIAM J. Sci. Stat. Comput.*, 13:146–167, 1992.

[MMB97] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Symposium on Interactive 3D Graphics*, pages 7–16, 1997.

[MMS97] J.S.B. Mitchell, D. M. Mount, and S. Suri. Query-sensitive ray shooting. *International Journal of Computational Geometry*, 7(4):317–347, 1997.

[MS81] P. Moon and D. E. Spencer. *The Photic Field*. MIT Press, Cambridge, 1981.

[Mun75] J. R. Munkres. *Topology: A first course*. Prentice Hall, Englewood Cliffs, NJ, 1975.

[NMN87] T. Nishita, Y. Miyawaki, and E. Nakamae. A shading model for atmospheric scattering considering luminous intensity of light sources. *Computer Graphics (Proc. of SIGGRAPH 87)*, 21(4):303–310, 1987.

[OM87] M. Ohta and M. Maekawa. Ray coherence theorem and constant time ray tracing algorithm. *Computer Graphics 1987 (Proc. of CG International '87)*, pages 303–314, 1987.

[OR97] M. Ohlberger and M. Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 56:365–385, 1997.

[OR98] E. Ofek and A. Rappoport. Interactive reflections on curved objects. *Computer Graphics (Proc. of SIGGRAPH 98)*, 14(3):333–342, July 1998.

[Paj98] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proc. IEEE Visualization'98*, pages 19–26, 1998.

[Paj02] R. Pajarola. Overview of quadtree-based terrain triangulation and visualization. Technical report, UCI-ICS-02-01, Information & Computer Science, University of California Irvine, 2002.

[PMS+99] S. Parker, W. Martin, P.J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *ACM Symposium on Interactive 3D Graphics*, pages 119–126, April 1999.

[PPS99] A.J. Preetham, P.Shirley, and B. Smits. A practical analytic model for daylight. In *Proc. of SIGGRAPH 99*, pages 91–100, 1999.

[PRM] PRMan. Photorealistic renderman application note#20: Writing fancy atmosphere shaders. http://graphics.stanford.edu/lab/soft/prman/Toolkit/-AppNotes/appnote.20.html.

[PZBG00] H. Pfister, M. Zwicker, J. Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Computer Graphics (Proc. of SIGGRAPH 2000)*, pages 335–342, 2000.

[Riv91] M.C. Rivara. Local modification of meshes for adaptive and/or multigrid finite-element methods. *J. Comput. Appl. Math.*, 36:79–89, 1991.

[RL92] M.C. Rivara and C. Levin. A 3-d refinement algorithm suitable for adaptive and multi-grid techniques. *Comm. Appl. Numer. Meth.*, 8:281–290, 1992.

[RL00] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Computer Graphics (Proc. of SIGGRAPH 2000)*, pages 343–352, 2000.

[RW80] S. Rubin and T. Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics (Proc. of SIGGRAPH 80)*, 14(3):110–116, July 1980.

[Sam90a] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.

[Sam90b] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[Sam92] H. Samet. Neighbor finding techniques for images represented by quadtrees. *Comput. Vision, Graph. and Image Processing*, 18(1):37–57, 1992.

[SAWG91] F. X. Sillion, J. Arvo, S. H. Westin, and D. P. Greenberg. A global illumination solution for general reflectance distributions. *Computer Graphics (Proc. of SIGGRAPH 91)*, 25(4):187–196, 1991.

[SCG97] P. P. Sloan, M. F. Cohen, and S. J. Gortler. Time critical lumigraph rendering. In *Proc. of 1997 Symp. on Interactive 3D Graphics*, pages 17–24, 1997.

[Sch92] G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Underst.*, 55(3):221–230, 1992.

[SD96] S. M. Seitz and C. R. Dyer. View morphing. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 21–30, 1996.

[SDB85] L. R. Speer, T. D. DeRose, and B. A. Barsky. A theoretical and empirical analysis of coherent ray tracing. *Graphics Interface '85*, pages 11–25, May 1985.

[Sew72] E.G. Sewell. *Automatic generation of triangulations for piecewise polynomial approximation*. PhD thesis, Purdue University, West Lafayette, IN, 1972.

[SGHS98] J. W. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered depth images. *Computer Graphics (Proc. of SIGGRAPH 98)*, 32:231–242, 1998.

[Shi90] P. Shirley. A ray tracing method for illumination calculation in diffuse-specular scenes. In *Proc. of Graphics Interface '90*, pages 205–12, 1990.

[SHS99] H. Schirmacher, W. Heidrich, and H. P. Seidel. Adaptive acquisition of lumigraphs from synthetic scenes. *Computer Graphics Forum (Eurographics '99)*, 18(3):151–160, September 1999.

[Sib81] R. Sibson. A brief description of natural neighbour interpolation. In Vic Barnet, editor, *Interpreting Multivariate Data*, pages 21–36. John Wiley & Sons, Chichester, 1981.

[SJ00] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. In *Rendering Techniques 2000 (11th Eurographics Workshop on Rendering)*, pages 319–328, 2000.

[Som34] D. M. Y. Sommerville. *Analytical Geometry in Three Dimensions*. Cambridge University Press, Cambridge, 1934.

[SS92] R. Sivan and H. Samet. Algorithms for constructing quadtree surface maps. In *Proc. 5th Int. Symp. on Spatial Data Handling*, pages 361–370, 1992.

[Sta99] J. Stam. Stable fluids. In *Proc. of SIGGRAPH 99*, pages 121–128, 1999.

[SZ00] P. Schröder and D. Zorin. Subdivision for modeling and animation. *SIGGRAPH 2000 Course Notes*, 2000.

[Tod76] M.J. Todd. The computation of fixed points and applications. In *vol. 124 of Lecture Notes in Economics and Mathematical Systems*, Berlin, 1976. Springer.

[WDP99] B. Walter, G. Drettakis, and S. Parker. Interactive rendering using the render cache. In *10th Eurographics Workshop on Rendering*, June 1999.

[WE98] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proc. of SIGGRAPH 98*, pages 169–178, 1998.

[Web84] R.E. Webber. *Analysis of quadtree algorithms*. PhD thesis, Department of Comp. Science, University of Maryland, College Park, MD, 1984.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Commun. of ACM*, 23(6):343–349, June 1980.

[ZCK97] Y. Zhou, B. Chen, and A. Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In *Proc. IEEE Visualization'97*, pages 135–142, 1997.

[Zha95] S. Zhang. Successive subdivisions of tetrahedra and multigrid methods on tetrahedral meshes. *Houston J. Math.*, 21:541–556, 1995.

[ZS01] D. Zorin and P. Schröder. A unified framework for primal/dual quadrilateral subdivision schemes. *Computer Aided Geometric Design*, 18:429–454, 2001.

[ZSS96] D. Zorin, P. Schröder, and W. Sweldens. Interpolating subdivision for meshes with arbitrary topology. *Computer Graphics (Proc. of SIGGRAPH 96)*, pages 189–192, 1996.