ABSTRACT

Title of dissertation: GEOMETRIC ISSUES IN SPATIAL INDEXING

Houman Alborzi, Doctor of Philosophy, 2006

Dissertation directed by: Professor Hanan Samet
Department of Computer Science

We address a number of geometric issues in spatial indexes. One area of interest is spherical data. Two main examples are the locations of stars in the sky and geodesic data. The first part of this dissertation addresses some of the challenges in handling spherical data with a spatial database. We show that a practical approach for integrating spherical data in a conventional spatial database is to use a suitable mapping from the unit sphere to a rectangle. This allows us to easily use conventional two-dimensional spatial data structures on spherical data. We further describe algorithms for handling spherical data. In the second part of the dissertation, we introduce the areal projection, a novel projection which is computationally efficient and has low distortion. We show that the areal projection can be utilized for developing an efficient method for low distortion quantization of unit normal vectors. This is helpful for compact storage of spherical data and has applications in computer graphics. We introduce the QuickArealHex algorithm, a fast algorithm for quantization of surface normal vectors with very low distortion. The third part of the dissertation deals with a CPU time analysis of TGS, an R-tree bulkloading algorithm. And finally, the fourth part of the dissertation analyzes the BV-tree, a data structure for storing multi-dimensional data on secondary storage. Contrary to the popular belief, we show that the BV-tree is only applicable to binary space partitioning of the underlying data space.

GEOMETRIC ISSUES IN SPATIAL INDEXING

by

Houman Alborzi

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

Advisory Committee:

Professor Hanan Samet, Chair
Professor Larry Davis
Professor Shunlin Liang
Professor David Mount
Professor Amitabh Varshney

ACKNOWLEDGMENTS

I would like to express my gratitude to many people whose friendship and support I relied on during my years at the University of Maryland. I thank my friend Jagan Sankaranarayanan, for his commitment in proofreading my dissertation. I thank my friends Jaime Montemayor, Pedram Hovareshti, Banafsheh Keshavarzi, Mehdi Kalantari-Khandani, Nargess Memarsadeghi, Guity Mohammadi, David Omer Horvitz, Bujor Silaghi, and Gabriel Rivera for their encouragement and support to finish my dissertation. I am thankful to my advisor Professor Hanan Samet for his unconditional support of my research and his personal friendship. I would also like to thank Professors Larry Davis, Shunlin Liang, David Mount, and Amitabh Varshney for serving on my advisory committee. In addition, I would like to thank my many teachers from whom I learned much during my years in graduate school. Moreover, I am thankful to Fatima Bangura and Nilo Rubin, as well as other staff member at the University of Maryland who were always eager to help me in one way or another.

I also like to thank my father Shokrollah Alborzi, my mother Ghodsieh-Soltan Javadi-Jahromi, my brother Kamran Alborzi, and my sister Marjan Alborzi for they were my first teachers. And last, but not least, I would like to thank my wife, Maliheh Poorfarhani, for her love, friendship, and emotional support.

Contents

1 Introduction **1**

1.1 Spatial Indexing Methods 2

1.1.1 PMR Quadtree 3

1.1.2 R-tree 3

1.2 SAND: Spatial and Non-spatial Database 4

2 Spherical SAND **6**

2.1 Introduction 6

2.2 Spatial Data Structure to Support Spherical Data 7

2.2.1 The first method of mapping using a cube 10

2.2.2 The second method of mapping using a cube 11

2.2.3 Criteria for of an appropriate mapping 12

2.2.4 Mapping using Lambert's cylindrical equal area projection 12

2.2.5 Mapping using Flattened Octahedron 12

2.3 Spatial Objects in Spherical SAND and Spherical Algorithms 13

2.3.1 Preliminaries and Notations 13

2.3.1.1 Vectors 13

2.3.1.2 Spherical point 14

2.3.1.3 Planes and circles 15

2.3.1.4 Projection of a great circle on the plane of another great circle . . . 16
2.3.1.5 Projection of a small circle on a great circle plane 16
2.3.1.6 Spherical line . . . 17
2.3.1.7 Small arc . . . 17
2.3.1.8 Spherical Polygon . . . 18
2.3.1.9 Excess of a Small Arc . . . 20
2.3.1.10 Lambert Rectangle . . . 24
2.4 Geometrical Operation on Spherical Objects . . . 25
2.4.1 Intersection of Two Spherical Points . . . 25
2.4.2 Distance between Two Spherical Points . . . 25
2.4.3 Distance between a Spherical Point and a Circle . . . 27
2.4.4 Distance of a point set to a circle . . . 27
2.4.5 Distance between a Spherical Point and a Spherical Line . . . 28
2.4.6 Intersection of a Spherical Point and a Spherical Line . . . 29
2.4.7 Intersection of a Spherical Point and a Spherical Polygon . . . 30
2.4.8 Distance between a Spherical Point and a Spherical Polygon . . . 32
2.4.9 Intersection of two Spherical Lines . . . 32
2.5 Extensions to the SAND Browser . . . 32

3 Low distortion normal vector quantization 34

3.1 Introduction . . . 34
3.2 Related Work . . . 36
3.3 Quantization Methods . . . 39
3.3.1 Octahedral Quantization . . . 39
3.3.1.1 Similar Methods . . . 41
3.3.2 Delta Encoding . . . 43
3.3.3 Hexagonal Cells . . . 44

3.3.4 Projections for Octahedral Quantization 46
3.3.4.1 Gnomonic Projection 46
3.3.4.2 Areal Projection 47
3.3.4.3 Buss-Fillmore Projection 49
3.3.4.4 Tegmark Projection 49
3.3.5 QuickAreal Algorithm 49
3.3.6 Quantization Using a Nearest Neighbor Finding Algorithm 51
3.3.6.1 Random Points 51
3.3.6.2 Saff-Kuijlaars Method 51
3.3.6.3 Spherical Centroidal Voronoi Tessellations (SCVT) . . . 52
3.3.7 Table of Quantization Methods 52
3.3.8 QuickArealHex Algorithm 53
3.4 Lower Bounds 55
3.4.1 A Tighter Lower Bound 57
3.5 Comparison of Quantization Methods 60
3.5.1 Encoding and Decoding Times 64
3.6 Rendering a Perfect Sphere 66
3.7 Summary and Conclusion 69

4 Execution time analysis of a top-down R-tree construction algorithm 87

4.1 Introduction 87
4.2 Background 89
4.3 TGS Bulk Loading Algorithm 92
4.4 Bottom-up Packing Versus Top-Down Packing Algorithms 95
4.5 Analysis 98
4.6 Concluding Remarks 101

5 BV-trees, axis aligned rectangles, and binary space partitioning 103

5.1 Introduction 103

5.2 Description of the BV-tree data structure 105

5.3 BV-trees and axis-aligned rectangles 105

5.4 Cordial regions and binary space partitioning 107

6 Conclusion and future work 113

6.1 Directions for future work 114

A 116

A.1 Derivation of the Areal Projection 116

List of Tables

3.1 The four extensions of a point (a,b) in different quadrants of the square. . . 44

3.2 A Summary of Quantization Methods 53

List of Figures

2.1 Flattening a cube on the plane. 11
2.2 The length of a spherical line. 17
2.3 An example of a spherical triangle. 18
2.4 Excess of a spherical arc (shown in thicker line) between spherical points *A* and *B*. The thin lines are great circle arcs. 20
2.5 An example of a Lambert rectangle. 25
2.6 Finding the distance of a point to a circle. 28
2.7 Example of the distance from a spherical point to a spherical line. 30

3.1 A block diagram of Octahedral Quantization, the proposed method for surface normal quantization. 39
3.2 Arrangement of eight right-angled triangles in a square. 41
3.3 Pattern of representative points for an even number of bits. 42
3.4 Pattern of representative points for an odd number of bits. 42
3.5 Extension of Figure 3.2 to a larger square. 44
3.6 First step in constructing the hexagonal pattern for an even number of bits. . 45
3.7 Second step in constructing the hexagonal pattern for an even number of bits. 46
3.8 Hexagonal pattern of representative points for an even number of bits. . . . 46
3.9 The point N inside spherical triangle $\triangle XYZ$. 48
3.10 The neighborhood search of the QuickArealHex algorithm. 54
3.11 Cross section of the unit sphere centered at O. 56

3.12 Tighter lower bounds. 58

3.13 Different error statistics of the Deering, the Geographic, the OQ-Gnomonic, the OQ-Areal, the OQ-Buss-Fillmore, and the OQ-Tegmark quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error. 71

3.14 Different statistics of the Geographic, the OQ-Gnomonic-Hex, the OQ-Areal-Hex, the OQ-Buss-Fillmore-Hex, and the OQ-Tegmark-Hex quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error. 72

3.15 Different statistics of the NN-Deering, the NN-Geographic, the NN-OQ-Areal, the NN-OQ-Buss-Fillmore, and the NN-OQ-Tegmark, quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error. 73

3.16 Different statistics of the NN-Deering, the NN-Geographic, the NN-OQ-Gnomonic-Hex, the NN-OQ-Areal-Hex, the NN-OQ-Buss-Fillmore-Hex, and the NN-OQ-Tegmark-Hex quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error. 74

3.17 Different error statistics of the NN-OQ-Gnomonic-Hex, the NN-OQ-Areal-Hex, the NN-OQ-Buss-Fillmore, the NN-OQ-Tegmark, the NN-Saff-Kuijlaars, and the NN-SCVT quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error. 75

3.18 The quantization time of different quantization methods. (a) Encoding time. (b) Decoding time. 76

3.19 Rendering a perfect sphere with normals quantized with 8 bits, using the Geographic, the NN-Geographic, the Deering, and the NN-Deering quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint. 76

3.20 Rendering a perfect sphere with normals quantized with 8 bits, using the OQ-Gnomonic, the NN-OQ-Gnomonic, the OQ-Tegmark, and the NN-OQ-Tegmark quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint. 77

3.21 Rendering a perfect sphere with normals quantized with 8 bits, using the OQ-Areal, the NN-OQ-Areal, the OQ-Buss-Fillmore, and the NN-OQ-Buss-Fillmore quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint. 78

3.22 Rendering a perfect sphere with normals quantized with 7 bits, using OQ-Gnomonic, NN-OQ-Gnomonic, OQ-Tegmark, and NN-OQ-Tegmark quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint. . . . 79

3.23 Rendering a perfect sphere with normals quantized with 7 bits, using the OQ-Areal, the NN-OQ-Areal, the OQ-Buss-Fillmore, and the NN-OQ-Buss-Fillmore quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint. 80

3.24 Rendering a perfect sphere with quantized normals at 8 bits, using the OQ-Areal, the NN-OQ-Areal, the OQ-Areal-Hex, and the NN-OQ-Areal-Hex quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint. . . 81

3.25 Rendering a perfect sphere with quantized normals using the NN-Random, the NN-Saff-Kuijlaars, the NN-SCVT, and the NN-OQ-Areal-Hex quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint. 82

3.26 Rendering a perfect sphere with normals quantized with the Deering and the NN-OQ-Areal-Hex quantization methods with different bits of quantization. The spheres in the top row are quantized using the Deering method, and the spheres in the bottom row are quantized using the NN-OQ-Areal-Hex. 82

3.27 Rendering of a sphere with specular highlight. 83

3.28 Rendering the sphere in Figure 3.27(a) with surface normals quantized with 14, 16, 18, and 20 bits using the Deering, the OQ-Gnomonic-Hex, the NN-OQ-Areal-Hex, and the NN-SCVT quantization methods. 84

3.29 A comparison of the Deering and QuickArealHex methods. (a) Normalized Maximum Quantization Error. (b) Encoding time. 86

4.1 Arrangement of bounding boxes. (a) A set of five boxes. (b) One bounding box for boxes (A, C) and one for (B, D, E). (c) One bounding box for boxes (A, B) and one for (C, D, E). 90

4.2 Result of applying the TGS bottom-up packing bulk loading algorithm to bulk load a packed R-tree using a cost function that minimizes (a) the overlap area, and (b) the total area. 98

4.3 Result of applying the top-down packing TGS bulk loading algorithm to bulk load an R-tree using a cost function that minimizes (a) the overlap area, and (b) the total area. 99

5.1 A pathological example of axis-aligned rectangles that leads to violation of the BV-tree design assumptions. 106

5.2 Example of a BV-tree with intervals as regions. The BV-tree, shown on the right, has a page capacity of three. Data points and the regions are shown on the left. The regions corresponding to level 0 nodes, level 1 nodes, and level 2 nodes are drawn in solid lines, dash-dot lines, and dash-dot-dot lines, respectively. 108

5.3 Example illustrating the definition of $\mathscr{R}_S^0$. The region S is the outer rectangle, and the sets $\mathscr{R}_S$ and $\mathscr{R}_S^0$ consist of the inner rectangles. 112

List of Algorithms

2.1 DOESINTERSECTPOINTPOINT(p_1, p_2) 26

2.2 DISTPOINTPOINT(p_1, p_2) 26

2.3 DISTPOINTLINE(p, l) 29

2.4 DOESINTERSECTPOINTLINE(q, l) 30

2.5 DOESINTERSECTPOINTPOLYGON(p, g) 31

2.6 INTERSECTIONPOINTSOFPLANES(p, q) 31

2.7 DISTPOINTPOLYGON(p, g) 32

2.8 DOESINTERSECTLINELINE(l_1, l_2) 33

4.1 BULKLOAD(D) 93

4.2 BULKLOADCHUNK$(\mathbf{D}, h)$ 94

4.3 PARTITION$(\mathbf{D}, m)$ 94

4.4 BESTBINARYSPLIT$(\mathbf{D}, m)$ 95

4.5 COMPUTEBOUNDINGBOXES(D, m) 96

4.6 SPLITONKEY(D, s, t) 96

Chapter 1

Introduction

We address a number of geometric issues in spatial indexes. One area of interest is spherical data which consists of geometric objects lying on a sphere. Two main examples are the locations of stars in the sky and geodesic data. The first part of this dissertation addresses some of the challenges in handling spherical data with a spatial database. We show that a practical approach for integrating spherical data in a conventional spatial database is to find a mapping from the unit sphere to a rectangle. This allows us to easily use conventional two-dimensional spatial data structures on spherical data. We further describe algorithms for handling spherical data.

In the second part of the dissertation, we introduce the areal projection, a novel projection which is computationally efficient and has low distortion. We show that the areal projection can be utilized for developing an efficient method for low distortion quantization of unit normal vectors. This is helpful for compact storage of spherical data and has applications in computer graphics. We discuss different normal quantization methods and provide an in-depth comparison of the methods. We introduce the QuickArealHex algorithm, a fast algorithm for quantization of surface normal vectors with very low distortion.

In the third part of the dissertation, a detailed CPU execution-time analysis and implementation are given for a bulk loading algorithm to construct R-trees due to García, López,

and Leutenegger [31] which is known as the top-down greedy split (TGS) bulk loading algorithm. The TGS algorithm makes use of a classical bottom-up packing approach. In addition, an alternative packing approach termed top-down packing is introduced which may lead to improved query performance, and it is shown how to incorporate it into the TGS algorithm. A discussion is also presented of the tradeoffs of using the bottom-up and top-down packing approaches.

The fourth part of the dissertation analyzes the BV-tree, a data structure for storing multi-dimensional data on secondary storage. Contrary to the popular belief, we show that the BV-tree is only applicable to binary space partitioning of the underlying data space.

In the rest of this chapter, we provide some background material about spatial indexes. We further introduce SAND, a software for spatial database management.

1.1 Spatial Indexing Methods

The B-tree and its variants (Comer [15] provides a comprehensive survey) are the data structure of choice for implementing indexes for databases. The B-tree design assumes that there is a total ordering of the keys, and hence, stores the keys in order on secondary storage. The B-tree recursively splits the data into smaller blocks. Associated with each block B is a key range (k^-, k^+), such that B contains all data elements with key $k : k^- \leq k \leq k^+$. A nice property of the B-tree is that it also allows efficient range searches. Moreover, if the each key is associated with a point of a one dimensional line, then the ordering of keys also preserves their proximity. That is, the closest key to each key is either its predecessor or its successor in the ordering. Spatial data, usually referring to geometric data in a 2-d or 3-d space, is not inherently suitable to be stored using the B-tree, as there does not exist a total ordering on spatial data that also preserves proximity. Hence, a few data structures have been designed to overcome the difficulties of efficiently organizing spatial data. Some of these spatial data structures, such as the linear quadtree [32], use a B-tree as an underlying

data structure. Nevertheless, almost all spatial data structures use similar concepts as the B-tree. Each data block is an aggregate of smaller data blocks. And each data block *B* with a region *R* of the space, called a bounding object of *B*. The data blocks which are not subdivided any further are called leaf nodes. Samet [65] provides an extensive survey of spatial data structures.

1.1.1 PMR Quadtree

The PMR quadtree [54] is a variant of the region quadtree [43,46,62] that can handle spatial objects of arbitrary dimensionality (i.e., including 2-d and 3-d). For example, in a two-dimensional space, the PMR quadtree subdivides the underlying rectangular space *r* into four congruent rectangular areas whenever the number of objects overlapping *r* exceeds a predefined value *s*, termed the splitting threshold. Each of the resulting areas contains references (via pointers) to the spatial descriptions of the objects that overlap it. The PMR quadtree is different from other bucketing methods. In particular, when the number of objects that overlap *r* exceeds the splitting threshold, then *r* is only subdivided once even though some of the resulting areas, say *a*, may still be overlapped by more than *s* objects. The area *a* will be subdivided the next time an object is inserted that overlaps it. This way, regions are not repeatedly subdivided when more than *s* objects lie very close to each other.

1.1.2 R-tree

The R-tree [39], originally designed for handling rectangles, is now widely used for indexing all kinds of spatial data. Associated with each data block of R-tree is a bounding rectangle *R* such that the spatial extents of all data in that the block are contained in *R*. The bounding rectangles associated with the children of a data block may overlap. Consequently, searches on an R-tree may involve traversing more than one path of the data structure.

1.2 SAND: Spatial and Non-spatial Database

SAND [20,21,66] is an interactive spatial database and browser developed at the University of Maryland. SAND combines a graphical user interface with a spatial and non-spatial database engine. It supports queries on both spatial and non-spatial data. Examples of queries are spatial selections and spatial joins. SAND supports the PMR quadtree, the R-tree, and the PK-tree [78] spatial indexing methods, and uses the B^+-tree index for non-spatial data. Although some of the SAND's spatial operations (e.g., selections and joins) are only implemented for few spatial indexing methods, most of the spatial operations are supported by the PMR quadtree.

Spatial selections in SAND involve finding all data objects whose spatial attribute overlaps the search region. Of particular interest are spatial range queries in SAND where a user queries the data set for objects whose distance from another data object is within a given distance range. For example, this feature enables a user to find all warehouses that are between 100 and 200 miles of a particular retail store. Another query feature of SAND allows users to search for all objects that have a certain orientation with respect to another data object. For example, a user can locate all warehouses that are north of a given location.

SAND also supports the join operation. There are many variants of this operation. The operation generates a subset of the Cartesian product of the two given relations *R* and *S* that satisfy a specified join condition. When the join condition is imposed on spatial attributes, the operation is known as a *spatial join*. The join condition often restricts tuples to lie within a given distance of each other. In particular, the *distance join* [40] orders the resulting tuples according to their spatial proximity. The *distance semi-join* [40] is a special case of the distance join in which each element of set *R* is paired up only with the closest member of set *S*. The resulting tuples are ordered by the distance between their constituent spatial attributes. For example, consider two data sets *R* and *S*, such that *R* contains the locations of the warehouses of a merchant, and *S* contains the locations of retail stores of the merchant. Using the distance semi-join, a user can find the closest warehouse for

each retail store. SAND implements distance semi-joins using an *incremental* algorithm developed by Hjaltason and Samet [42].

Chapter 2

Spherical SAND

2.1 Introduction

This chapter discusses the design and implementation of a spherical data model for SAND. SAND is a spatial database developed at University of Maryland that combines a graphical user interface with a spatial and non-spatial database engine. SAND supports geometric operations on a few common geometric objects, such as points, lines, rectangles, and polygons. Operations that are supported on these objects are (i) measuring distance between two objects; (ii) determining if two objects intersect; and (iii) determining if an object contains another object. Additionally, SAND can compute the length of a line, and the areas of rectangles and polygons. These geometric operations are fundamental to SAND as the spatial indexes and the spatial queries support by SAND are constructed with these geometric operations. The geometric objects of SAND were originally implemented for data lying in a 2-d or a 3-d space.

We extended SAND to support geometric objects that lie on a sphere as SAND was not able to correctly handle spherical data such as data that lie on the surface of the Earth. In particular, SAND was not able to correctly calculate the distance between data objects on the surface of the Earth, and the planar data model only provided reasonably accurate

responses to a small portion of the Earth. The main shortcoming of SAND was that the distance function did not take into account the curvature of the Earth. The addition of a spherical data model gives SAND the ability to correctly perform queries on a spherical data model, *i.e.*, data that represent features on the surface of the Earth.

The rest of this chapter is organized as follows. Section 2.2 discusses the different considerations that were taken into account in designing the spatial data structure that supports spherical data. Section 2.3 presents the spatial objects that are needed in spherical SAND. Section 2.4 describes the algorithms needed to deal with spherical geometry, specially computing distances between spherical primitives, while Section 2.5 indicates changes that were made to the SAND Browser to enable its use for viewing spherical data.

2.2 Spatial Data Structure to Support Spherical Data

In this section, we describe the various approaches that we undertook to extend SAND to support spherical data. Earlier versions of SAND supported data primitives such as polygons, lines, and points on a plane. We enhanced the SAND data objects to support spherical polygons, spherical lines, and points on a sphere. SAND consists of a large codebase that supports spatial data structures for two-dimensional and three-dimensional spaces. Instead of redesigning all the spatial data structures and operations for the spherical data, we focused on different ways in which a sphere could be mapped to a two-dimensional space (plane). In the following discussion, we use the term *data space* to describe the space in which the data resides, and *grid space* for the space in which the data structure manipulations take place. For example, if the sphere is mapped onto a plane p, and a quadtree decomposition is subsequently performed on the plane, then the plane p is the 'grid space'. Obviously, there should exist a *mapping* between the data space and the grid space. Ideally, the mapping would need to permit efficient operations on the data.

There are two ways to implement the mapping. One is to immediately map the data ob-

jects onto the grid space whenever they are modified or inserted into the database. Using the above method requires computing a mapping of spherical shapes into planar shapes upon performing insertion or update operations. This approach could be advantageous when the required geometric operations are more expensive to perform in the data space than they are in the grid space. An alternative approach is to map the partitions on the grid space into the data space. In our example, this approach requires finding spherical curves which when mapped to the grid space form the quadtree partitions of the plane. This approach is advantageous if the geometric operations performed in the grid space are more complex than the same operations performed in the data space. In the first approach, an object in the data set must be mapped from the data space onto the grid space, whereas in the second approach only the grid partitions must be mapped from the grid space onto the data space. Assuming that there are more data objects than partition lines, it appears that mapping the grid space onto the data space is cheaper from a computational complexity standpoint than mapping the data space onto the grid space. This is especially true if the geometric operations performed in the grid space maps the data back to the data space. For example, a possible algorithm for determining the intersection of two spherical lines mapped as planar lines on the grid space, maps the planar lines back to sphere and determines the intersection on sphere.

Mapping the grid space onto the data space can be made more efficient by storing the result of the mappings of the grid space onto the data space in the data structure. For example, in the case of a quadtree-like subdivision in the grid space, we can maintain the result of mapping the partition lines from the grid space to the data space in the data structure. In addition, we should bear in mind that even for a database with a few insertions or updates, mapping data objects onto the grid space may not be computationally feasible. For example, in the case of mapping a sphere into a plane, a spherical line may not necessarily be mapped onto a line on the plane. Hence, performing computations on the result of the mappings is not a straightforward task. Similar problems can be encountered when

designing a mapping from the grid space onto the data space. The mappings of partitions of the grid should make use of simple geometrical primitives where geometrical algorithms to compute the distance and intersection between objects are easy to implement.

In the case of spherical SAND, we use the second approach where we map the grid into the data space. In fact, we map the data structure grid onto the data space, and perform the geometrical operations directly in the data space. We investigate four different mappings between the data space and the grid space. The first two approaches are based on embedding a cube in the sphere and projecting points on the cube to the sphere, or vice versa as they are equivalent since the mappings are one-to-one and onto. The map of a point P on the sphere in this approach is calculated by shooting a ray from the center of sphere to P, the intersection of the ray with the cube is the map of P. These approaches were based on the ideas proposed by Scott [67]. A cube is a polyhedron with the property that all faces of it are squares and are regular polygons. A polyhedron such that all faces are equal regular polygons is called a Platonic solid. There are only five platonic solids, namely the *tetrahedron* with four triangle faces and four vertices of degree three, the *cube* with six square faces and eight vertices of degree three, the *octahedron* with eight triangle faces and six vertices of degree four, the *dodecahedron* with twelve pentagon faces and twenty vertices of degree three, and the *icosahedron* with twenty triangle faces and twelve vertices of degree five. Notice that if we take the cube and replace each face by a vertex in the middle of the face, and connect the vertices whose corresponding adjacent faces are adjacent, we will obtain an octahedron. Hence, we consider the cube and the octahedron as duals. Similarly, the dodecahedron and the icosahedron are duals, while the tetrahedron is the dual of itself. Projecting a sphere into a platonic solid is a common practice in spatial data structures [23, 33, 80].

The third approach is based on an equal area cylindrical projection of the plane onto the sphere (also known as Lambert's cylindrical equal area projection [69]). Tobler and Chen [12, 74] have used the same approach for building spherical quadtrees.

A fourth approach presented here is based on projecting a sphere into an octahedron and then flattening the octahedron into a square. Praun and Hoppe [55] suggest mapping a sphere into flattened octahedron as well. We independently came up with the same scheme for the purpose of projecting spherical data into a square.

2.2.1 The first method of mapping using a cube

In the first approach that we tried, we mapped the sphere onto the cube based on an idea developed by Scott [67]. We modeled each face of the resulting cube with a quadtree data structure thereby using six quadtree structures. It should be noted that Scott's method for calculating the mapping between the subdivisions of cubical faces and their spherical counterparts is incorrect. In particular, Scott uses a parallel projection that results in certain portions of the sphere not being covered on the cube. We solved this problem by projecting through the center of the sphere (*i.e.*, Gnomonic projection). This mapping has the property that any line on a face of the cube maps to a spherical line on the sphere (*i.e.*, a great circle arc). Therefore, we only needed to implement geometrical algorithms dealing with spherical lines. However, lines parallel to the Equator, when projected onto the cube will not be a line anymore.

Another interesting feature of this mapping is that the spherical polygons are mapped as polygons on the cube, and by storing the projection of each data object we can use faster planar geometric algorithms instead of the spherical ones. However, a drawback of this mapping is that it is not an equal area projection. This means that data uniformly-distributed in the data space are not uniformly-distributed after projection into the grid space.

Even though implementing this approach appears straightforward, we encountered considerable difficulties when we tried to modify many parts of SAND to incorporate it. For example, in the case of a general spatial join, we would have to perform 36 pairwise intersections — one for every possible pair of faces of the two cubes that correspond to the two

joined sets. Moreover, in many of the standard functions in SAND there is an implicit assumption that the data is stored in a single quadtree. Finding and debugging all the related code seemed impractical for this task.

2.2.2 The second method of mapping using a cube

Observing the infeasibility of using six quadtrees for any dataset, we used an alternative approach where we flattened the cubic faces on a plane. In other words, the grid space was considered to be a single rectangle which contains all resultant six faces of result of projecting the sphere onto the cube (see Figure 2.1). This approach allowed us to reuse many of the SAND routines with no extra effort. However, the main drawback of this approach was that some of the regions in the grid space did not have a corresponding region (*i.e.*, were undefined) in the data space. Thus some of the algorithms in SAND failed to work properly without further modification. In particular, not every connected region in the grid space had a corresponding connected region on the sphere. This was a problem because some of the operations in SAND examined every block spanned by the region in the grid space and some of these blocks were not well defined on the sphere, and hence difficult to deal with. The dotted rectangle in Figure 2.1 shows such a block.

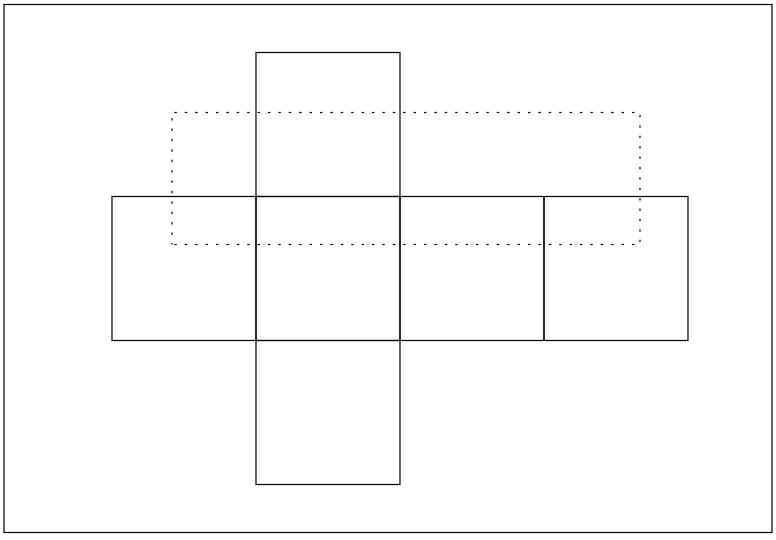


Figure 2.1: Flattening a cube on the plane.

2.2.3 Criteria for of an appropriate mapping

Based on our experience with the first two approaches, we concluded that an appropriate mapping for SAND should have the following properties:

1. Maps the sphere into a single rectangle,

2. Any axis-aligned rectangle on the plane should be mapped to a simple shape on the sphere.

2.2.4 Mapping using Lambert's cylindrical equal area projection

We use Lambert's cylindrical equal area projection as the mapping. This mapping is also an equal area projection and hence preserves the uniformity of data points. However, it has singularities at the poles, where the poles will be mapped into lines in the grid space. A side effect is that data primitives around the poles will be elongated in the projection. A horizontal line in this mapping maps to an arc of a small arc on the sphere and a vertical line in this mapping maps to an arc of great circle on the sphere. Hence, using this mapping, a rectangle in the grid space will be mapped to a spherical quadrilateral, such that two of its edges are small arcs (see Section 2.3.1.7) and the other two are spherical lines (see Section 2.3.1.6).

2.2.5 Mapping using Flattened Octahedron

In Section 3.3.1 we describe how an octahedron can be flattened to a square. We also introduce the Areal projections. Using the flattened octahedron in combination with using either the Gnomonic projection or the Areal projection satisfy our criteria for a suitable mapping. If we use the Gnomonic projection line in one of the eight triangle of the square map to a spherical line on the unit sphere. Thus, any polygon, including a rectangle, in the grid space will map to a polygon in the data space. However, if we use the Areal

projection, then only the vertical and horizontal lines and the diagonal lines parallel to the partition lines of Figure 3.4 will map to small arcs of the sphere.

In the sections that follow, we present more details about spherical data primitives and geometrical algorithms needed for the implementation.

2.3 Spatial Objects in Spherical SAND and Spherical Algorithms

This section introduces the spatial objects in the Spherical SAND. The spherical objects include spherical points, spherical lines, spherical polygons, and Lambert rectangles. In the following section, we describe the spherical objects and some of their properties. We follow by describing the algorithms that operate on spherical objects.

2.3.1 Preliminaries and Notations

All objects in spherical SAND reside on the surface of a *sphere* of unit radius. Let O denote the center of the sphere which is also the origin of the coordinate system. S^2 denotes the surface of the sphere. While all the objects reside on S^2, it is convenient to also use three-dimensional Euclidean space $\mathbb{R}^3$ whenever needed. We freely use a Cartesian coordinate system, and/or a spherical coordinate system, to specify the coordinates of objects. The triple (x,y,z) and the pair (λ,ϕ) denote a point in Cartesian and spherical coordinate systems respectively λ is known as the *longitude* of the point, and ϕ is known at its *latitude*. For p and q points in $\mathbb{R}^3$, $|pq|$ denotes the Euclidean distance between p and q.

2.3.1.1 Vectors

For two vectors $\vec{u}$ and $\vec{v}$, $\vec{u}\cdot\vec{v}$ denotes their dot product and $\vec{u}\times\vec{v}$ denotes their cross product. NORMALIZE$(\vec{v})$ denotes the unit vector corresponding to $\vec{v}$. The cosine of angle $\angle\vec{u},\vec{v}$

between two unit vectors $\vec{u}$ and $\vec{v}$ is equal to the dot product of the unit vectors. Equally, $\angle\vec{u},\vec{v} = \arccos(\vec{u}\cdot\vec{v})$.

2.3.1.2 Spherical point

The basic unit of data is a *spherical point* which is a point on S^2, the surface of the unit sphere. There is a one-to-one mapping between points on S^2 and the unit vectors in $\mathbb{R}^3$. The corresponding unit vector from O to a point p is denoted by $\vec{p}$. For any point P with coordinates (x,y,z), its *antipodal* point is the point $\overline{P}$ with coordinates $(-x,-y,-z)$. A point, its antipodal, and O are collinear. Furthermore, the distance from a point P to O equals the distance from P's antipodal point $\overline{P}$ to O, or formally $|OP| = |O\overline{P}|$. Given a spherical point P with Cartesian coordinates (x,y,z), and spherical coordinates (λ,ϕ), the following relationships hold,

$$\lambda = \arctan(y,x) \tag{2.1}$$

$$\phi = \arcsin z \tag{2.2}$$

$$x = \cos\lambda\cos\phi \tag{2.3}$$

$$y = \sin\lambda\cos\phi \tag{2.4}$$

$$z = \sin\phi \tag{2.5}$$

$$1 = x^2+y^2+z^2. \tag{2.6}$$

Recall that λ is also known as the longitude of the spherical point, and ϕ is known at its latitude. Notice that the spherical coordinates defined here are different from the convention in adopted in some calculus textbooks, where $z = \cos\delta$, and $\delta = \frac{\pi}{2} - \phi$ is called the *colatitude*.

For spherical points p and q, DISTPOINTPOINT(p,q) denotes the spherical distance between p and q, which is the length of the shortest arc of the unit sphere connecting p and q.

2.3.1.3 Planes and circles

The intersection of the unit sphere with a plane forms a circle. If the plane passes through the center of the sphere, then the intersection is called a *great circle*, and it has a unit radius. If the center of the sphere does not occur on the intersecting plane, the resulting circle is termed a *small circle*. A plane can be represented as $(\vec{n}, d)$ where $\vec{n}$ is its normal vector and d is its distance to the origin O, such that a point p is on the plane if and only if $\vec{p} \cdot \vec{n} = d$. Notice that both $(\vec{n}, d)$ and $(-\vec{n}, -d)$ represent the same plane. The plane $(\vec{n}, d)$ and the unit sphere intersect, if and only if, $d \leq 1$. Notice that, in case $d = 0$, the circle is a great circle; and otherwise it is a small circle. The small circle will be called a small circle of displacement d or a small circle of radius r. The intersection of the plane $(\vec{n}, d)$ with the unit sphere is a circle with radius $r = \sqrt{1 - d^2}$. Note that the radius of a a great circle is always 1. For the plane $q = (\vec{n}, d)$, NORMALVECTOR(q) denotes $\vec{n}$ its normal vector and DISPLACEMENT(q) denotes d its distance to O. Fir a circle c, PLANE(c) denotes the plan containing c. Moreover, CENTER(c) and RADIUS(c) denote the center and the radius of the circle respectively. We also use DISPLACEMENT(c) and NORMALVECTOR(c) to refer to DISPLACEMENT(PLANE(c)) and NORMALVECTOR(PLANE(c)) respectively.

Any three non-collinear points in $\mathbb{R}^3$ specify one and only one plane passing through them. Hence, the center of the sphere and any two non-antipodal points p and q on the sphere specify exactly one plane whose normal is NORMALIZE$(\vec{p} \times \vec{q})$ and hence, exactly one great circle of the sphere. A small circle can be specified by the plane normal $\vec{n}$ and a point p on it. The distance d of the small circle to the great circle parallel to it is $|\vec{p} \cdot \vec{n}|$. The center o of a small circle is $d\vec{n}$. Or,

$$\text{CENTER}(c) = \text{DISPLACEMENT}(c)\text{NORMALVECTOR}(c).$$

In case we are representing a great circle only, we can omit the parameter d to save storage.

It is interesting to notice that a great circle and a spherical point are duals. Where, the

dual of a spherical point p is the great circle with normal $\vec{p}$. In fact, any two antipodal spherical points share the same great circle as their duals. Based on this observation, we may as well define *oriented great circles*. Every great circle overlaps two directional great circles with their directions going in opposite directions.

2.3.1.4 Projection of a great circle on the plane of another great circle

The projection of a great circle c_1 on the plane of a great circle c_2 is an ellipse centered at O and having two radii r_1 and r_2 such that

$$r_1 = 1$$

and

$$r_2 = r_1 |\text{NORMALVECTOR}(c_1) \cdot \text{NORMALVECTOR}(c_2)|.$$

2.3.1.5 Projection of a small circle on a great circle plane

The projection of a small circle c on the plane of a great circle C is an ellipse centered at q and having two radii r_1 and r_2 such that

$$\begin{aligned} q &= \text{CENTER}(c) - (\text{CENTER}(c) \cdot \text{NORMALVECTOR}(C))\text{NORMALVECTOR}(C) \\ r_1 &= \text{RADIUS}(c) \\ r_2 &= r_1 |\text{NORMALVECTOR}(c) \cdot \text{NORMALVECTOR}(C)| \end{aligned}$$

The ellipse is spread along the vector $\text{NORMALVECTOR}(C) \times \text{NORMALVECTOR}(c)$. The ellipse has no intersection with the great circle if and only if $\text{RADIUS}(c) < |\text{NORMALVECTOR}(C) \cdot \text{NORMALVECTOR}(c)|$

2.3.1.6 Spherical line

A *spherical line* is the collection of all points on the sphere on the shortest path between two spherical points that are termed its two endpoints. Note that if the two endpoints of a spherical line are antipodals, then there are many spherical lines defined by them. Any two non-antipodal spherical points specify a unique spherical line. The great circle formed by two non-antipodal spherical points is divided into two arcs of non-equal length where the spherical line is the shorter arc. Figure 2.2 shows two spherical points A and B and the circle of radius r is the circle passing through O, A and B. The length of the arc (drawn using bold line) is 2α.

The endpoints of a spherical line l, are specified by $\text{ENDPOINTONE}(l)$ and $\text{ENDPOINTTWO}(l)$. As each spherical line l is an arc of a great circle, we can also specify the plane $\text{PLANE}(l)$ that contains l. The spherical line between two spherical points A and B is denoted by $\text{LINE}(A,B)$.

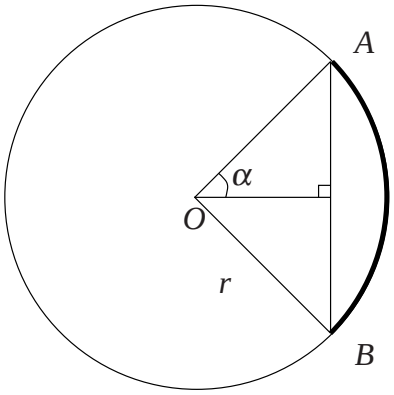


Figure 2.2: The length of a spherical line.

2.3.1.7 Small arc

There are many small circles passing through two spherical points p and q. The shortest arc between p and q on a small circle of displacement d and radius $r = \sqrt{1-d^2}$ is called a *small arc* of displacement d between p and q. $\text{DISTPOINTPOINT}(p,q,d)$ denotes the length of such an arc. Consider the small circle of radius r containing the small arc as in

Figure 2.2. We have

$$\text{DISTPOINTPOINT}(p,q,d) = 2r\alpha = 2r\arcsin\frac{|pq|/2}{r}.$$

For $\gamma = \arcsin d = \arccos(r)$, and $\beta = \arccos(\vec{p}\cdot\vec{q})$, using Equation 2.11, we obtain

$$\text{DISTPOINTPOINT}(p,q,d) = 2\cos\gamma\arcsin\frac{\sin\beta/2}{\sin\gamma}. \tag{2.7}$$

2.3.1.8 Spherical Polygon

A *spherical polygon* is a closed region on the sphere bounded by non-intersecting spherical lines. We represent a spherical polygon by a circular list of spherical points ordered in such a way that two adjacent spherical points in the list specify a spherical line (edge) bounding the spherical polygon. Figure 2.3 shows an example of a spherical triangle. For a spherical polygon g, $\text{NSIDES}(g)$ denotes the number of edges of g which is also equal to the number of vertices of g. Moreover, $\text{SIDE}(g,i)$ denotes the i^{th} edge of g.

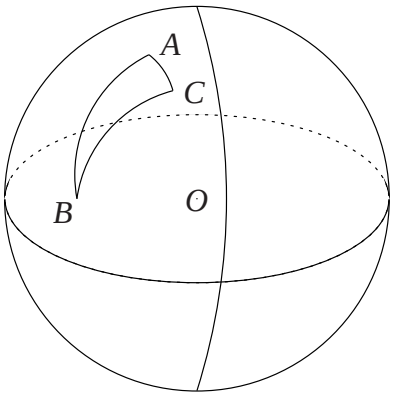


Figure 2.3: An example of a spherical triangle.

The angle between two intersecting spherical lines is defined as the angle between the tangents of the great circles of the spherical lines that pass through the intersection point. If the intersection point of two adjacent spherical line segments of a spherical polygon is denoted by B and the other endpoints of the spherical line segments are denoted by A and

C, then the angle at vertex B of the spherical polygon is equal to the angle between the planes containing the great circles of the spherical lines, which is

$$\pi - \arccos(\text{NORMALIZE}(\vec{A} \times \vec{B}) \cdot \text{NORMALIZE}(\vec{B} \times \vec{C})). \tag{2.8}$$

Girard's spherical excess formula [79] derives the area of a spherical polygon using to the sum of its angles. Assume the spherical polygon has n vertices $v_1 \ldots v_n$. Let α_i denote internal angle of vertex v_i. Then,

$$\text{Area} = \sum_{i=1}^{n} \alpha_i - (n-2)\pi \tag{2.9}$$

A spherical polygon divides the sphere in two parts, one assumed to be the interior of the polygon, and the other one the exterior of the polygon. There are many ways to designate interior of a polygon, for example, one may assume that the inside area of a polygon should always be of smaller area than the outside area. However, this may cause ambiguities in cases where the polygon divides the sphere into two equal area sections. Considering that the area of the unit sphere is 4π, this representation implies that the area of a spherical polygon is always less than 2π. This approach is used in the current implementation of spherical SAND. We term this representation *small area spherical polygon* or *SA spherical polygon*.

Another way to designate the interior of a spherical polygon is to associate a spherical point p such that p is properly inside the spherical polygon. This requires storing an additional spherical point for each spherical polygon. We term this representation as *explicit interior point spherical polygon* or *EIP spherical polygon.*

A better option that does not require any additional data stored is to define *side-oriented spherical polygons* or *SO spherical polygons* by proper ordering of the polygon vertices. Consider a person walking along the spherical polygon starting from the first vertex in the list, moving to the second vertex, and so on. The interior of the spherical polygon

is defined to be on the left hand side of the observer. In case we want to complement a spherical polygon, we need to change the order of spherical points in its representation. However, a more efficient scheme is to assign a binary flag indicating whether the interior of the spherical polygon is at the left hand side of the observer or at right hand side of the observer.

We can also allow small arcs as edges of a spherical polygon, which is especially useful in modeling areas of the Earth between two parallels. In order to compute the area of a spherical polygon where some of its edge are small arcs, we first calculate the area of the spherical polygon assuming all its edges are spherical lines, we then compute a small arc excess as defined in Section 2.3.1.9 for each small arc edge and add it to (or subtract it from) the area of the spherical polygon.

2.3.1.9 Excess of a Small Arc

The *excess of small arc* with endpoints A and B and displacement d is the area of lune bounded between the spherical line A and B and the small arc of displacement d with endpoints A and B. The excess of small arc is denoted by $S(A,B,d)$.

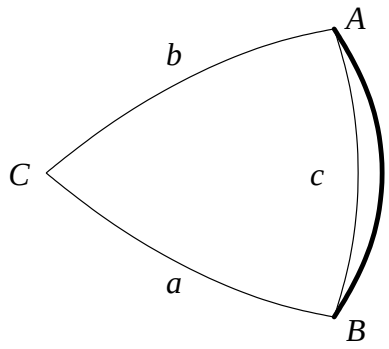


Figure 2.4: Excess of a spherical arc (shown in thicker line) between spherical points A and B. The thin lines are great circle arcs.

Song et. al [70] describe a method for deriving $S(A,B,d)$ using spherical coordinates. We use the method described by Song et. al [70] to derive $S(A,B,d)$ using Cartesian co-ordinates. Let the plane of small circle be $(\vec{C},d)$. Point C is on the unit sphere and its

projection on the small circle is the center of the small circle (see Figure 2.4). S_{cap}, the area of the spherical cap between C and the small circle is $2\pi(1-d)$. S_2, the area of the part of this spherical cap which is between spherical lines CA and CB is $\frac{\angle C}{2\pi}$ of S_{cap}, we have $S_2 = (1-d)\angle C$.

We have $S(A,B,d) = S_2 - S_1$, where S_1 is the area of spherical triangle $\triangle ABC$ and S_2 is the area of the spherical section bounded between spherical lines AC, BC and small arc AB.

Let the edges of the spherical triangle $\triangle ABC$ be a, b and c. We have $\angle A = \angle B$, $a = b$, $\cos\angle C = \frac{p-d^2}{1-d^2}$ where $d = \cos a$, $p = \cos c = \vec{A}\cdot\vec{B}$.

Using law of cosines [79] for the spherical angles we get:

$$\begin{aligned}
\cos\angle C &= -\cos^2\angle A + \sin^2\angle A\cos c \\
\cos\angle C &= \sin^2\angle A - 1 + p\sin^2\angle A \\
\cos\angle C &= \sin^2\angle A(1+p) - 1.
\end{aligned}$$

Hence,

$$\sin^2\angle A = \frac{1+\cos\angle C}{1+p}. \tag{2.10}$$

On the other hand,

$$\begin{aligned}
S_1 &= \angle A + \angle B + \angle C - \pi \\
&= 2\angle A + \angle C - \pi \\
S_2 &= (1-d)\angle C \\
S(A,B,d) &= S_2 - S_1.
\end{aligned}$$

Hence,

$$\begin{aligned} S(A,B,d) &= 2\left(\frac{\pi}{2}-\angle A\right)-d\angle C \\ &= \arccos\left(\cos\left(2\left(\frac{\pi}{2}-\angle A\right)\right)\right)-d\angle C \\ &= \arccos\left(2\cos^2\left(\frac{\pi}{2}-\angle A\right)-1\right)-d\angle C \\ &= \arccos\left(2\sin^2\angle A-1\right)-d\angle C \\ &= \arccos\left(2\frac{1+\cos\angle C}{1+p}-1\right)-d\angle C \end{aligned}$$

$$\begin{aligned} S(A,B,d) &= \arccos\left(2\frac{1+\frac{p-d^2}{1-d^2}}{1+p}-1\right)-d\angle C \\ &= \arccos\left(2\frac{1+\frac{p-d^2}{1-d^2}}{1+p}-2+1\right)-d\angle C \\ &= \arccos\left(2\left(\frac{1+\frac{p-d^2}{1-d^2}}{1+p}-1\right)+1\right)-d\angle C \\ &= \arccos\left(2\frac{1+\frac{p-d^2}{1-d^2}-1-p}{1+p}+1\right)-d\angle C \end{aligned}$$

$$\begin{aligned}
S(A,B,d) &= \arccos\left(2\frac{\frac{p-d^2}{1-d^2}-p}{1+p}+1\right)-d\angle C \\
&= \arccos\left(2\frac{\frac{p-d^2-p+pd^2}{1-d^2}}{1+p}+1\right)-d\angle C \\
&= \arccos\left(2\frac{\frac{-d^2+pd^2}{1-d^2}}{1+p}+1\right)-d\angle C \\
&= \arccos\left(2\frac{\frac{(p-1)d^2}{1-d^2}}{1+p}+1\right)-d\angle C \\
&= \arccos\left(2\frac{d^2}{1-d^2}\frac{p-1}{p+1}+1\right)-d\angle C.
\end{aligned}$$

$S(A,B,d)$ can be further simplified in terms of trigonometric functions:

$$\begin{aligned}
S(A,B,d) &= \arccos\left(2\frac{d^2}{1-d^2}\frac{p-1}{1+p}+1\right)-d\angle C \\
&= \arccos\left(2\frac{\cos^2 a}{\sin^2 a}\frac{\cos b-1}{\cos b+1}+1\right)-d\angle C \\
&= \arccos\left(2\frac{\cos^2 a}{\sin^2 a}\frac{1-2\sin^2\frac{b}{2}-1}{2\cos^2\frac{b}{2}-1+1}+1\right)-d\angle C \\
&= \arccos\left(2\frac{\cos^2 a}{\sin^2 a}\frac{-2\sin^2\frac{b}{2}}{2\cos^2\frac{b}{2}}+1\right)-d\angle C \\
&= \arccos\left(-2\frac{\tan^2\frac{b}{2}}{\tan^2 a}+1\right)-d\angle C
\end{aligned}$$

However, we know that, $\arccos(1-2x^2)=2\arcsin(x)$, and also that $\sin\frac{C}{2}=\frac{\sin\frac{f}{2}}{\sin a}$. Hence:

$$\begin{aligned}
S(A,B,d) &= \arccos\left(1-2\left(\frac{\tan\frac{b}{2}}{\tan a}\right)^2\right)-d\angle C \\
&= 2\arcsin\frac{\tan\frac{b}{2}}{\tan a}-d\angle C \\
&= 2\arcsin\frac{\tan\frac{b}{2}}{\tan a}-2\cos a\arcsin\frac{\sin\frac{b}{2}}{\sin a}.
\end{aligned}$$

2.3.1.10 Lambert Rectangle

In the two-dimensional SAND, the space is divided into rectangles. Rectangular subdivisions have two desirable properties. First of all, it is relatively easy to test for inclusion of a point in a rectangle. Second, it is easy to subdivide them into smaller rectangles. Re-examining Equation 2.9 in Section 2.3.1.8, we see that a four-sided spherical polygon with four right angles has an area of 0. In other words, a spherical rectangle with four sides covers just a single point of sphere. In other words, a right angled quadrilateral cannot be defined non-trivially on a sphere. To overcome this, we define a rectangle in an appropriate planar projection of sphere. We used Lambert's cylindrical equal-area projection [69] to define such a rectangle, which we term a *Lambert rectangle*.

A *Lambert rectangle* is a collection of spherical points with their longitudes and latitudes in a given range $((\lambda_1,\lambda_2),(\phi_1,\phi_2))$. That is, a spherical point with spherical coordinates (λ,ϕ) is inside a Lambert rectangle $((\lambda_1,\lambda_2),(\phi_1,\phi_2))$ if and only if $\lambda_1 \leq \lambda \leq \lambda_2$ and $\phi_1 \leq \phi \leq \phi_2$. The area of such a rectangle is

$$\int_{\lambda_1}^{\lambda_2}\int_{\phi_1}^{\phi_2} dS = \int_{\lambda_1}^{\lambda_2}\int_{\phi_1}^{\phi_2} \cos\phi \, d\phi \, d\lambda = (\lambda_2-\lambda_1)(\sin\phi_2 - \sin\phi_1).$$

Considering the premise that the z-coordinate value of any spherical point is equal to the sine of its latitude ($z = \sin\phi$), and also the fact that $\sin(\cdot)$ is a monotonically increasing function from $-\pi$ to π, we can represent the range with $((\lambda_1,\lambda_2),(z_1,z_2))$. The area of such a Lambert rectangle is $(\lambda_2-\lambda_1)(z_2-z_1)$. Figure 2.5 is an example of a Lambert rectangle.

One of the benefits of using Lambert rectangles is that we can specify the whole sphere with a single Lambert rectangle with longitudinal range of $(-\pi,\pi)$ and latitudinal range of $(-\frac{\pi}{2},\frac{\pi}{2})$. A Lambert rectangle is also easily divisible into smaller Lambert rectangles. Another incidental property that make Lambert rectangles natural choices for a spherical quadtree are the subdivision rules. A Lambert rectangle subdivides into four *equal*

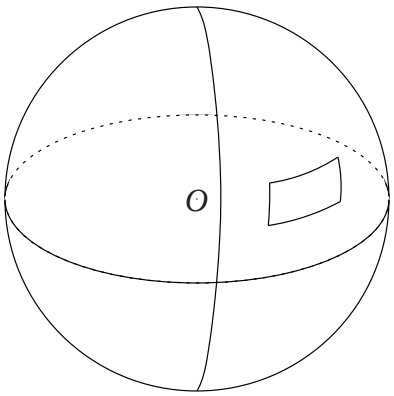


Figure 2.5: An example of a Lambert rectangle.

area smaller rectangles which can be done effortlessly by using the center of the rectangle, $\frac{\lambda_1+\lambda_2}{2}, \frac{z_1+z_2}{2}$ as center of subdivision. If that the point data objects on a sphere are uniformly-distributed, the Lambert rectangles provide the same performance for quadtree-based data structures as in the planar case (*i.e.*, in 2D).

2.4 Geometrical Operation on Spherical Objects

In this section, we describe some of the algorithms used in the development of spherical SAND. In particular, SAND needed algorithms for determining if two spatial objects intersect and for calculating the distance between two spatial objects. For any pair of object types in spherical SAND, we had to implement the distance and intersection functions which will be described in the following sections.

2.4.1 Intersection of Two Spherical Points

Two spherical points intersect if and only if they have the same coordinates (Algorithm 2.1).

2.4.2 Distance between Two Spherical Points

The distance between two spherical points A and B is α, the length of the arc between A and B. We have $\sin\alpha = |AB|/2$. On the other hand, we know that for $\beta = 2\alpha$, $\cos 2\alpha =$

Algorithm 2.1 DOESINTERSECTPOINTPOINT(p_1, p_2)

```
(* Determine whether two spherical points p1 and p2 intersect. *)
if p1 = p2 then
  return true
else
  return false
end if
```

$\cos\beta = \vec{A}\cdot\vec{B}$.

Hence,

$$\frac{|AB|}{2} = \sin\frac{\beta}{2}. \tag{2.11}$$

And,

$$\begin{aligned} d_S(A,B) &= \beta = \arccos(\vec{A}\cdot\vec{B}) && (2.12)\\ &= 2\alpha = 2\arcsin\frac{|AB|}{2}. && (2.13) \end{aligned}$$

Both Equation 2.12 and Equation 2.13 can be used to compute DISTPOINTPOINT(A,B), however Equation 2.13 is preferable for very small values of $|AB|$ due to the loss of precision in limited precision arithmetic [35]. For example for $A = (0,0,1)$ and $B = (2^{-13}, 2^{-13}, 1 - 2^{-26})$, Equation 2.13 results in 0.00017263348854612559 which is precise up to 10 digits, while the Equation 2.12 results in zero.

The spherical distance between two spherical points is the length of the spherical arc between the two spherical points (Algorithm 2.2).

Algorithm 2.2 DISTPOINTPOINT(p_1, p_2)

```
(* Calculate the distance between two spherical points p1 and p2. *)
return 2 arcsin |p1p2| / 2
```

Notice that arcsin is a monotonically increasing function in the range $[0,1]$. Therefore, DISTPOINTPOINT(p_1, p_2) is a monotonically increasing function of $|p_1, p_2|$ as well. Hence, for queries such nearest neighbor queries, that only require a relative ordering of

distances, calculating the Euclidean distance is sufficient.

2.4.3 Distance between a Spherical Point and a Circle

The closest point to spherical point p on circle c can be obtained by first projecting p on the plane of c to obtain point p^T and then extending p^T to the circle through the center of c. For $d = \text{DISPLACEMENT}(c)$ and $r = \text{RADIUS}(c)$, we have $r^2 + d^2 = 1$. In the following equations, D is the distance of the point p from circle c, and X is the distance of p from the plane of c, which can be obtained by $\text{NORMALIZE}(\vec{p} - \vec{p}_c) \cdot \text{NORMALVECTOR}(c)$, where p_c is some point on the plane of circle.

Consider the case depicted in the left hand side of Figure 2.6 where p and O, the center of sphere, lie on the same side of the plane of c. Using the additional symbols a and b from Figure 2.6, we have,

$$\begin{aligned} D^2 &= X^2 + (r-b)^2 \\ &= X^2 + \left(r - \sqrt{1-a^2}\right)^2 \\ &= X^2 + \left(r - \sqrt{1-(X-d)^2}\right)^2. \end{aligned} \tag{2.14}$$

In case p and O are on opposite sides of the plane of c as depicted in the right hand side of Figure 2.6, we get

$$D^2 = X^2 + \left(r - \sqrt{1-(X+d)^2}\right)^2 \tag{2.15}$$

2.4.4 Distance of a point set to a circle

Consider a set of points S and a circle c such that all points S lie on the same side of plane of c. From Equation 2.14 and Equation 2.15, we can observe that the distance from a point to a circle is directly related to its distance to the plane of the circle. Hence, the closest point of the point set S to circle c is the closest point in S to the plane of c.

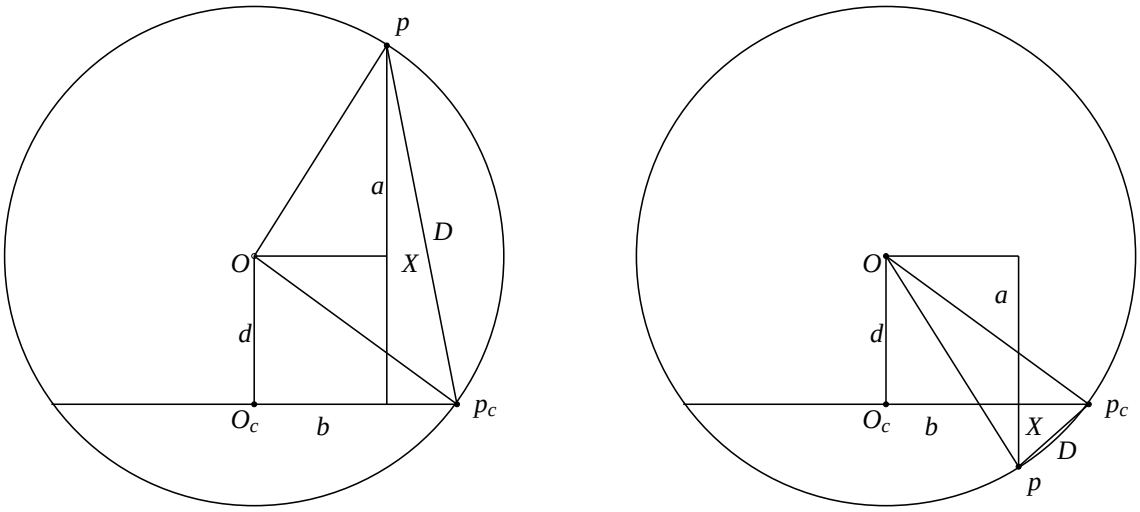


Figure 2.6: Finding the distance of a point to a circle.

We use this observation to find the closest point of a point set S to a circle c, by first partitioning S into sets S_1 and S_2, such that all points in S_1 are on one side of the plane of c and all points in S_2 are on the other side of the plane of c. We can then find the point p_1 the closest point in S_1 to the plane of c, and point p_2 the closest point in S_2 to the plane of c. Finally, we use Equation 2.14 and Equation 2.15 to find the closest point among p_1 and p_2 to c.

2.4.5 Distance between a Spherical Point and a Spherical Line

Consider a plane T, a point p not on the plane T, and a set $D \subseteq T$. Let p^T denote the projection of p on plane T. For any point d in D we have, $|pd|^2 = |pp^T|^2 + |p^T d|^2$. Therefore, in order to find the closest (or farthest) point in D to p it suffices to find the closest or farthest point in D to p^T. This observation will be used to find the distance of a spherical point to a spherical line.

The distance from a spherical point p to a spherical line l with endpoints A and B is the distance from p to A, B, or some other point q which lies on l. q has the property that it is co-linear with the line joining the origin O and the projection C of p on the plane containing

l. Let $\vec{m}$ denote the unit normal vector of the plane containing the spherical line l.

$$\vec{m} = \text{NORMALIZE}(\vec{A} \times \vec{B}).$$

we have

$$\vec{C} = \vec{p} - (\vec{p} \cdot \vec{m})\vec{m},$$

and

$$\vec{q} = \text{NORMALIZE}(\vec{C}).$$

We should also test if q lies on the spherical line. Based on these considerations, in order to find the distance between a spherical point p and a spherical line l with endpoints A and B, function DISTPOINTLINE, given in Algorithm 2.3, first finds q and then returns the shortest distance from p to either A, B, or q.

Algorithm 2.3 DISTPOINTLINE(p,l)

```
(* Calculate the distance of the spherical point p to the spherical line l. *)
A ← ENDPOINTONE(l)
B ← ENDPOINTTWO(l)
n⃗ ← NORMALIZE(A⃗ × B⃗)
q⃗ ← NORMALIZE(p⃗ − (p⃗ · n⃗)n⃗)
if DOESINTERSECTPOINTLINE(q, LINE(A,B)) then
   return DISTPOINTPOINT(p,q)
else
   return min(DISTPOINTPOINT(p,A), DISTPOINTPOINT(p,B))
end if
```

2.4.6 Intersection of a Spherical Point and a Spherical Line

The function DOESINTERSECTPOINTLINE(Algorithm 2.4) determines whether a spherical point q lies on the great circle of a spherical line l with endpoints of A and B. It is easy to see that q lies on the spherical line l, if and only if, the angle $\angle AqB$ in the triangle $\triangle AqB$ is obtuse (see Figure 2.7). This check is simple to make in the sense that $\angle AqB$ is 90 degrees if the sum S of the squares of the lengths of the two edges that comprise it is equal to the

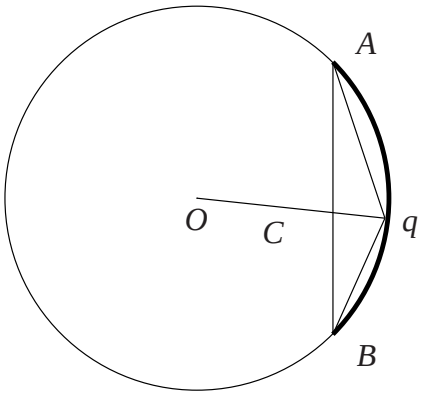


Figure 2.7: Example of the distance from a spherical point to a spherical line.

square of the length of the edge AB denoted by H. The angle is acute (obtuse) if S is less (greater) than H.

Algorithm 2.4 DOESINTERSECTPOINTLINE(q,l)

(* *Determine whether spherical point q lies on the spherical line l.* *)
$A \leftarrow$ ENDPOINTONE(l)
$B \leftarrow$ ENDPOINTTWO(l)
return $|qA|^2 + |qB|^2 < |AB|^2$

2.4.7 Intersection of a Spherical Point and a Spherical Polygon

Throughout this section, we assume that we the spherical polygons are small area spherical polygons. That is, the area of the interior of a spherical polygon is less than the area of its exterior. The algorithms related to the interior of spherical polygon should be modified in case a different convention for specifying the interior of a spherical polygon is used.

In order to check if a spherical point p is inside a spherical polygon r we construct a great circle c through p and another point q chosen at random and check if c intersects r. If no, then p lies outside r. If yes, then we examine the edge e containing the closest intersection point to p. If p is on the side of e that is inside r, then p is indeed inside r; otherwise, p is outside r. The function DOESINTERSECTPOINTPOLYGON, given in Algorithm 2.5 achieves this test.

Algorithm 2.5 DOESINTERSECTPOINTPOLYGON(p,g)

(* *Determine whether the spherical point p intersects the spherical polygon g.* *)
Let q be a plane passing through p and O.
Let r be a point on q such that $p \neq r$.
minDistance $\leftarrow \infty$
for $i = 1$ **to** NSIDES(g) **do**
  $l \leftarrow$ SIDE(g,i)
  $p_1 \leftarrow$ ENDPOINTONE(l).
  $p_2 \leftarrow$ ENDPOINTTWO(l).
  $(x_1,x_2) \leftarrow$ INTERSECTIONPOINTSOFPLANES$(q,$PLANE$(l))$
  for $j = 1$ **to** 2 **do**
    if DOESINTERSECTPOINTLINE(x_j,l) **then**
      intersects $\leftarrow$ **true**
      distance $\leftarrow |x_j r|$
      if distance $<$ minDistance **then**
        minDistance $\leftarrow$ distance
        inside $\leftarrow$ sign(NORMALVECTOR(PLANE$(l)) \cdot q$)
      end if
    end if
  end for
  $\alpha \leftarrow \arccos($NORMALVECTOR(SIDE$(g,i)) \cdot$ NORMALVECTOR(SIDE$(g,i+1)))$
  $s \leftarrow$ sign(ENDPOINTONE(SIDE$(g,i)) \times$ ENDPOINTTWO(SIDE$(g,i+1)) \cdot$
  ENDPOINTTWO(SIDE$(g,i)))$
  area $\leftarrow$ area $+s\alpha$
end for
areaSign $\leftarrow \left\lfloor \frac{area}{2\pi} \right\rfloor \bmod 2$
return intersects **and** (inside **xor** areaSign)

The function DOESINTERSECTPOINTPOLYGON makes use of the function INTERSECTIONPOINTSOFPLANES given in Algorithm 2.6 to determine two points on the sphere corresponding to the endpoints of the spherical line formed by the intersection of two planes p and q of two great circles.

Algorithm 2.6 INTERSECTIONPOINTSOFPLANES(p,q)

(* *Find the intersection points of two planes p and q and the unit sphere* *)
$\vec{x} \leftarrow$ NORMALIZE(NORMALVECTOR$(p) \times$ NORMALVECTOR$(q))$
return $(x,\overline{x})$

2.4.8 Distance between a Spherical Point and a Spherical Polygon

In order to find the distance between a spherical point and a spherical polygon, we need to consider two cases; either (i) the spherical point is on the polygon or (ii) it is not on the polygon. In the first case, the distance is simply zero. In the second case, the point is not on the polygon and the distance is the minimum of all distances from the point to the edges of the polygon. The function DISTPOINTPOLYGON, given in Algorithm 2.7 correctly computes the distance between a spherical point and a spherical polygon.

Algorithm 2.7 DISTPOINTPOLYGON(p,g)

```
(* Find the distance between the spherical point p and the spherical polygon g. *)
if DOESINTERSECTPOINTPOLYGON(p,g) then
   return 0
else
   return min_i DISTPOINTLINE(p, SIDE(g,i))
end if
```

2.4.9 Intersection of two Spherical Lines

Two distinct spherical lines can only intersect at the intersection points of their corresponding great circles. Hence, two spherical lines l_1 and l_2 intersect if and only if at least one of the two intersection points of their corresponding great circles lies on both l_1 and l_2. Function DOESINTERSECTLINELINE, given in Algorithm 2.8, achieves this test.

2.5 Extensions to the SAND Browser

The SAND Browser is a graphical user interface for SAND that uses a two-dimensional display system for displaying data. The GUI is used to compose and perform queries. Using our design, incorporating the spherical data type into the SAND Browser was straightforward. The main modification to the SAND Browser's graphical user interface was the addition of the ability to render spherical lines. In the current implementation, a spherical

Algorithm 2.8 DOESINTERSECTLINELINE(l_1, l_2)

(* *Determine whether two spherical lines l_1 and l_2 intersect.* *)
$p_1 =$ ENDPOINTONE(l_1)
$p_2 =$ ENDPOINTTWO(l_1)
$q_1 =$ ENDPOINTONE(l_2)
$q_2 =$ ENDPOINTTWO(l_2)
for $j = 1$ **to** 2 **do**
  if DOESINTERSECTPOINTLINE$(x_i,$ LINE$(p_1, p_2))$ **and**
  DOESINTERSECTPOINTLINE$(x_i,$ LINE$(q_1, q_2))$ **then**
    return *true*
  end if
end for
(* *p lies on the great circle arc between spherical points p_1 and p_2.* *) ¡x_1, x_2¿ = INTERSECTIONPOINTSOFPLANES$(l_1.plane, l_2.plane)$
return *False*

line is approximated by many short line segments on the display. We use a heuristic to decide how many segments are needed for a good visual approximation of the spherical line. The heuristic uses the latitude of the two endpoints and the length of the line. If the endpoints are far from poles or the line is long, then the heuristic uses more line segments.

An additional feature of the SAND Browser is the spatial selection operation which enables a user to select data items that are located in a sector. A sector is represented by a point and two rays emanating from that point. To support the sector on a sphere, we use a spherical lune [79], which allows users to select the spherical data that is located on a spherical lune. In order to specify the lune, the user selects one endpoint p of the lune, two spherical lines having p and the antipodal of p as their endpoints. Notice that only p need be specified (*i.e.*, the antipodal of p is not specified by the user). Since there are infinite number of spherical lines between p and the antipodal of p, the user specifies the spherical lines served to demarcate the lune.

Future work could involve the incorporation of additional primitives into the spherical model of SAND. Examples include great circles and their arcs, small circles and their arcs, and spherical polygons that cover more than half of the sphere. Also, the ability to perform spherical visualizations is also an interesting feature for future implementation.

Chapter 3

Low distortion normal vector quantization

3.1 Introduction

Compressing geometry models has recently been a subject of great interest [16, 17, 29, 44, 72]. The goal is to reduce the number of bits required to represent a geometry model in order to lower the storage space, or to lower the transmission time of the model across the network, or from the CPU to the GPU.

Realistic rendering of a geometry model requires knowledge of the surface normals at various points of the model. The surface normals are either stored explicitly as part of the geometry model or derived from other components of the model during rendering. For example, it is straightforward to compute the surface normal of an oriented triangle during rendering. The surface normals could then be used to calculate color and texture information at various points of the model. If the surface normals are stored explicitly, they are usually stored as the surface normals of the vertices in a mesh model, or the surface normals of each point in point clouds [59]. Moreover, some texture models such as bump maps [6] store a normal vector as part of the texture information. The efficient encoding of

surface normals could also be utilized to lower the storage cost of such texture models.

Deering [16] points out that the usual practice of storing a surface normal using 96 bits — three 32-bit floating point numbers — is wasteful. He proposed selecting 100,000 representative surface normals on the unit sphere and then quantizing each surface normal to a nearby — but not necessarily nearest — representative surface normal. Each of the 100,000 surface normals can be encoded using $\lceil \log_2 100000 \rceil = 17$ bits. Deering also suggested using a *delta encoding* scheme to further compress a stream of surface normals that have spatial correlation.

In this work, we present a framework for quantizing surface normals to any arbitrary number of bits. Every normal vector quantization method includes an *encoding* component, where a surface normal n with Cartesian coordinates $(x,y,z) : x^2+y^2+z^2 = 1$ is represented by Q bits; and a *decoding* component where the Q bits representing n will be used to compute n_q the *representative normal* of n, $n_q = (x_q, y_q, z_q) : x_q^2 + y_q^2 + z_q^2 = 1$. The one to one correspondence between three-dimensional unit normal vectors and the surface of the unit sphere allows us to consider each surface normal as a *spherical point*, *i.e.*, a point on the surface of the unit sphere. Hence, normal vector quantization is the same as quantizing the surface of the unit sphere. In this article, we do not distinguish between a three-dimensional unit normal vector and a spherical point. Furthermore, we define *compression* of a stream of unit normal vectors to be the process of converting a collection of normal vectors, as part of a geometry model, to a stream of bits. The key difference between a quantization technique and a compression technique is that the former is applied to a single datum, while the latter is applied to a data collection.

Traditionally, methods for normal vector compression of surface normals have been considered in conjunction with the techniques for compressing whole geometry models. Techniques for compressing geometry models usually use a surface normal quantization technique for efficient storage of surface normals. Usually the Deering method is chosen for such applications (*e.g.*, [14, 72]). This article provides a comprehensive study and

analysis of many surface normal quantization techniques, and also proposes different error measures in order to evaluate the relative merits of the discussed techniques. In particular, we present a new quantization method called QuickArealHex which is better than the Deering method in terms of (1) the quantization error, (2) the rendering quality, and (3) the computational efficiency as measured by the time and memory needed to encode and decode the normalized values. This article, to the best of our knowledge, is the only comprehensive study of surface normal quantization methods in the computer graphics literature. This article only discusses surface normal quantization techniques, and does not address surface normal compression techniques. Nevertheless, a surface normal quantization method could be used as part of a statistical compression scheme, such as those proposed by Gandoin and Devillers [17, 29] who use an arithmetic coding [82] scheme.

The rest of this paper is organized as follows. Section 3.2 describes previous work in this area. Section 3.3 presents two methods for surface normal quantization. Section 3.4 derives a loose theoretical lower bound for the quantization error. Section 3.5 compares different quantization methods in terms of the quantization error and the computation efficiency, while Section 3.6 describes how the different quantization methods affect the rendering quality. Concluding remarks are drawn in Section 3.7.

3.2 Related Work

As the surface of a sphere is a two dimensional surface, it is possible to find a mapping from a spherical point (x,y,z), to a two dimensional point (u,v). Such mappings have been extensively studied in the map projection and cartography literature, as in [69]. A primary goal of a cartographer is to visually present geographical information on paper which is a two dimensional medium. Most map projections are categorized based on their inherent properties that are used in different applications. For example, an *equal area projection* does not distort the area measure of a shape when projected to the plane, while a *conformal*

projection preserves the local angles of shapes. Computer graphics applications also use map projection techniques. Environment mapping techniques [7] utilize a projection of the sphere onto a two-dimensional surface, such as the surface of a cube [36]. Arvo and Kirk [2] use a sphere-to-cube projection to speed up ray tracing applications.

A natural way of quantizing the surface of a sphere is to first project the sphere onto a plane and then to quantize the plane. A similar technique is adopted by Deering [16] who divides the unit sphere into eight equal octants, and then subdivides each octant into six equal sextants. Each octant is an equilateral spherical triangles with three 90°internal angles. Each sextant is a spherical triangle with internal angles of 90°, 60°, and 45°. The eight octants of the sphere naturally define a regular octahedron. However, other Platonic solids, such as the cube and the icosahedron have also been used for the purpose of quantizing the surface of a sphere [2, 73].

Deering [16] uses two six-bit values ϕ_n and θ_n to encode 2,145 representative surface normals on each sextant. Each octant is encoded with three bits, and each sextant of an octant is encoded with another three bits. Therefore, Deering uses 18 bits to represent 102,960 surface normals. As we can see, the Deering method wastes at least one bit, as 102,960 surface normals could have been represented using only 17 bits, *i.e.*, $2^{17} > 102,960$. The Deering algorithm requires the precomputation of all 2,145 representative surface normals of a sextant. The encoding algorithm performs linear search among these surface normals to find the closest representative normal to the given surface normal. Thus, the encoding algorithm would require a considerable amount of CPU time. The Deering algorithm for decoding a surface normal first determines the octant and the sextant of the quantized normal (encoded by six bits) and subsequently uses the remaining 12 bits as an index to a lookup table of the precomputed surface normals. Furthermore, each coordinate value of a normal is stored using 16-bit fixed precision number. Thus, the size of the lookup table is 24 kilobytes, which could be costly for a hardware implementation. Cignoni et al. [14] describe a model used for visualizing tetrahedral meshes. They use the Deering

method for compact storage of the normal vectors of a model. The Deering method is also used in [34] for storing quantized normals of a triangular mesh. Willmott [81] also proposes using the Deering method for reducing the memory requirements of hierarchical radiosity.

An alternative quantization method represents a normal vector by its geographic coordinates, *i.e.*, its latitude and longitude. It then quantizes each coordinate into the desired number of bits. Kugler [48] uses this method for hardware rendering. A disadvantage of this method is that the quantization error of a normal is highly skewed. That is, the normals around the equator of the unit sphere on average have a higher quantization error than the normals near the poles of the unit sphere.

Given a spherical point (x,y,z), another quantization method calculates (x_0,y_0,z_0), such that

$$(x_0,y_0,z_0)\cdot(|x|+|y|+|z|)=(x,y,z).$$

This is equivalent to the *gnomonic projection* of the unit sphere on an octahedron. Notice that $|x_0|+|y_0|+|z_0|=1$, and thus it suffices to only store x_0, y_0, and the sign of z_0. Furthermore, x_0 and y_0 will each be quantized to the desired number of bits. If $2n+1$ bits are used for quantizing a normal, n bits are used for quantization of the x_0 and y_0 coordinates. However, not all possible (x_0,y_0) values are valid. In particular, all values of (x_0,y_0) such that $|x_0|+|y_0|>1$ are not valid. For example, we cannot have $(0.9,0.9)$ for (x_0,y_0), as $0.9+0.9>1$. Hence, only half of the 2^{2n} possible values for (x_0,y_0) are valid. Thus, this method, similar to the Deering method, wastes one bit. A similar method [59] projects the unit sphere onto a cube and then uses a uniform grid on the surface of the cube.

In [8,37,71,75] recursive subdivisions of an octahedron are used to encode each surface normal with the triangle containing the surface normal. The encoding in this method is iterative, and hence would be slow in practice. For example, Botsch et al. [8] use 13 bits per surface normal for a point sampled rendering application. Three bits are used to encode the face of the octahedron, and two bits are used for each quaternary subdivision. Thus, the

recursive subdivision of the octahedron is five levels deep, and hence, encoding a surface normal requires five iterations.

3.3 Quantization Methods

In this section, we discuss two distinct methods for normal vector quantization. We first introduce our proposed method, called Octahedral Quantization. Subsequently, we describe quantization methods that are based on a nearest neighbor finding algorithm. However, for the sake of completeness, we also mention *Geographic Quantization* in this section. In Geographic Quantization, a normal vector is first converted to its Geographic coordinates (*i.e.*, latitude and longitude), and then each geographic coordinate is quantized to the desired number of bits.

3.3.1 Octahedral Quantization

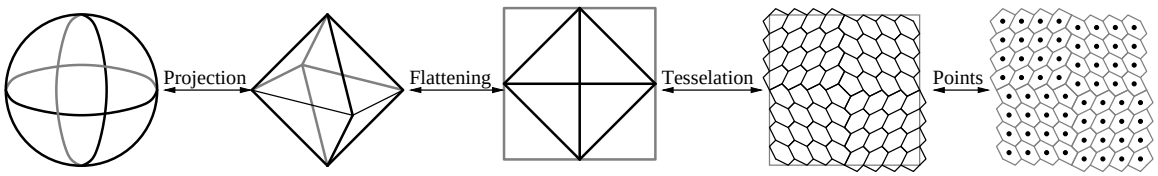


Figure 3.1: A block diagram of Octahedral Quantization, the proposed method for surface normal quantization.

Our proposed framework for surface normal quantization, called the *Octahedral Quantization* uses projections of the unit sphere onto a regular octahedron. Figure 3.1 is a functional diagram summarizing the steps of the Octahedral Quantization method. After projecting the sphere onto the octahedron, the faces of the octahedron are flattened onto a plane, such that they form a square. Then the square is tessellated using an appropriate pattern. For each cell of the tessellation, a representative point is chosen.

Recall that a surface normal is a point on the surface of the unit sphere. The encoding process thus maps each surface normal to a cell of the tessellation and hence, the represen-

tative point of the cell. The decoding process projects the representative point back to the octahedron and to the unit sphere. Note that any appropriate projection or tessellation can be chosen — independent of each other — in the framework.

We use a regular octahedron placed around the unit sphere such that the vertices of the octahedron are placed on the coordinate axes. The *positive face* of the octahedron is the triangle T with vertices $X = (1,0,0)$, $Y = (0,1,0)$, and $Z = (0,0,1)$. Notice that, for all points (x,y,z) on the positive face of the octahedron, we have $x,y,z \geq 0$ and $x+y+z = 1$. Similarly, the *positive spherical octant* of the unit sphere is the set of points (x,y,z) on the sphere such that $x,y,z \geq 0$. In the rest of the discussion, we only consider the encoding of a normal vector in the positive face of the octahedron as the treatment of the other faces is similar. We choose an arbitrary projection that projects the positive spherical octant of the sphere to the positive face of the octahedron. We suggest using projections that are computationally efficient. In Section 3.3.4, we describe a few such projections, namely, the Gnomonic projection, the Areal projection, the Buss-Fillmore projection, and the Tegmark projection.

In general, the spherical point (x,y,z), will be mapped to a point $P = (a,b,c)$ on the positive face of the octahedron, where $a+b+c = 1$ and $0 \leq a,b,c \leq 1$. (a,b,c) are also the barycentric coordinates of P with respect to the triangle T. Now as $a+b+c = 1$, it suffices to only store and encode two of the three barycentric coordinates, as the third can be obtained from the remaining two coordinates. We choose to store a and b, as this maps the vector $(1,0,0)$ to the point $(1,0)$, the vector $(0,1,0)$ to the point $(0,1)$, and the vector $(0,0,1)$ to the point $(0,0)$. We use the equation $c = 1-a-b$, when the value of c is needed. We thus have $0 \leq 1-a-b \leq 1$, or equally, $0 \leq a+b \leq 1$. The locus of the points (a,b) with $0 \leq a+b \leq 1$ is a right-angled isosceles triangle with vertices $(1,0)$, $(0,1)$, and $(0,0)$. That is, we have shown how to project a face of the octahedron to a right-angled isosceles triangle. Thus the eight faces of the octahedron map to eight equal right-angled isosceles triangles, and we can arrange them in a square with side length of two as in Figure 3.2.

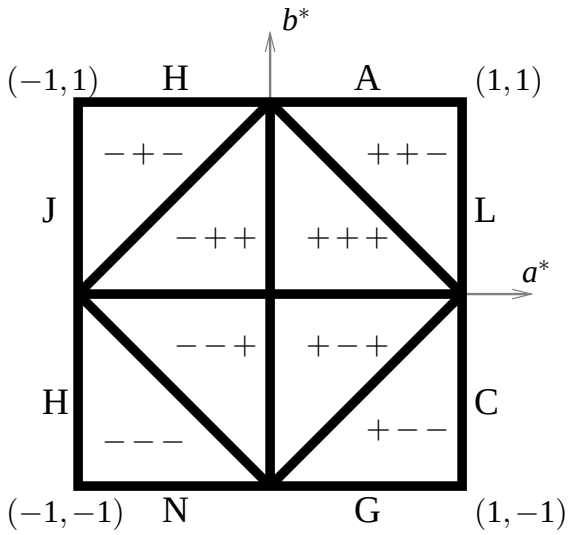


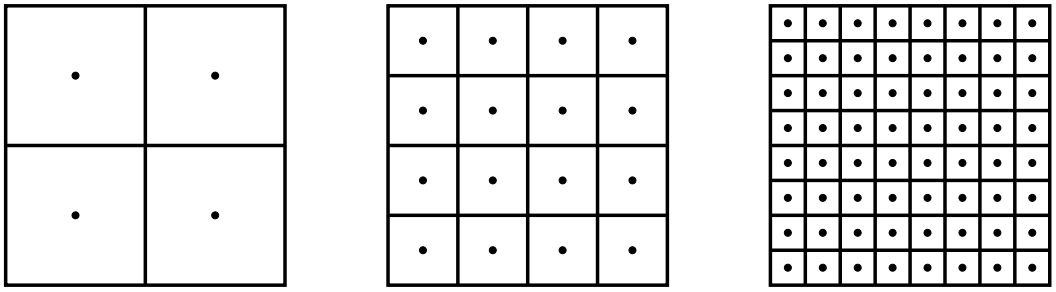
Figure 3.2: Arrangement of eight right-angled triangles in a square.

The placement of the eight triangles is such that the edges that are adjacent on the square are also adjacent on the octahedron. We have labeled each triangle with the signs of the *x*, *y*, and *z* coordinates of the spherical points corresponding to each triangle. For example, the $-+-$ in the upper-left triangle denote that points corresponding to that triangle have negative *x*, positive *y* and negative *z* coordinates. We use (a^*, b^*) to denote the coordinates of a point in the square of Figure 3.2.

In the next step, we quantize (a^*, b^*) to the desired number of bits. For an even number of bits, we use the pattern shown in Figure 3.3, where each dot denotes a quantized point. For an odd number of bits, we use a slightly more complex pattern shown in Figure 3.4. Notice that the representative points in Figures 3.3 and 3.4 are the centroids of their corresponding cells.

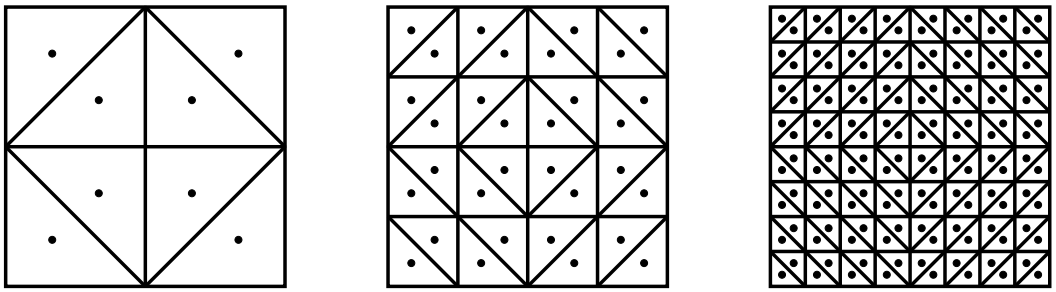
3.3.1.1 Similar Methods

The Deering [16] method for surface normal quantization also uses a regular octahedron. Moreover, our method partly resembles the work of Praun and Hoppe [55] for spherical parameterization of mesh models. The technique of projecting the sphere onto a square using an octahedron was previously proposed by Dutton [19] and Praun and Hoppe [55].



(a) 2 bits (b) 4 bits (c) 6 bits

Figure 3.3: Pattern of representative points for an even number of bits.



(a) 3 bits (b) 5 bits (c) 7 bits

Figure 3.4: Pattern of representative points for an odd number of bits.

That is, they project the unit sphere onto a regular octahedron, and then arrange the faces of the octahedron to form a square. In spite of the similarities, Dutton [19] and Praun and Hoppe [55] apply the flattening technique to different applications other than surface normal quantization. The focus of Dutton [19] was to devise a method for multi-resolution encoding of geographical data. While Dutton's work is closely related to surface normal quantization, we examines the effect of a variety of projections on the quality of the quantization for computer graphics applications. In contrast, Praun and Hoppe [55] propose techniques for spherical parameterization of a mesh, and hence the quality measure used in their work does not directly correspond to the quantization error of surface normals.

3.3.2 Delta Encoding

To further compress a stream of quantized normal vectors, a delta encoding scheme could be used. The delta encoding scheme is a simple compression method that is widely used in computer graphics applications (*e.g.*, [16]). While the compression of normal vectors is not the focus of this article, we still mention the correct way to apply delta encoding to surface normals that are quantized using the Octahedral Quantization method. Instead of directly encoding the data, the delta encoding scheme encodes the differences between successive data elements. The vector (a^*, b^*) is encoded as $(a^* - a^*_p, b^* - b^*_p)$, where (a^*_p, b^*_p) is the previous vector in the stream. Thus, if successive normal vectors have spatial locality, then the delta vector has a smaller range and can be encoded more compactly.

The outer edges of the square in Figure 3.2 fold and touch at their midpoints. For example, the half-edges H and A are actually the same edge of the octahedron. Similarly, the half-edges L, C; N, G; and J, E are the same edge of the octahedron. Therefore, the square could be extended to cover part of a larger square as shown in Figure 3.5. Notice that each point in the smaller square will appear four times in the larger square. This can be observed by looking at the star or the triangle symbols which are placed in Figure 3.5 to illustrate this property. In other words, a point $(a^*, b^*), 0 \leq a^*, b^* \leq 1$ extends to the points

$$\begin{cases} (a^*, b^*), \\ (2 - a^*, -b^*), \\ (-a^*, 2 - b^*), \\ (a^* - 2, b - 2^*). \end{cases}$$

Hence, the correct delta vector for point $(a^*, b^*), 0 \leq a^*, b^*, \leq 1$ will be the shortest delta

vector among

$$\begin{cases} (a^* - a_p^*, b^* - b_p^*), \\ ((2-a)^* - a_p^*, -b^* - b_p^*), \\ (-a^* - a_p^*, (2-b)^* - b_p^*), \\ ((a-2)^* - a_p^*, (b-2)^* - b_p^*). \end{cases}$$

Table 3.1 tabulates the four extensions of a point in different quadrants of the square.

$a^* > 0, b^* > 0$	$a^* > 0, b^* < 0$	$a^* < 0, b^* > 0$	$a^* < 0, b^* < 0$
(a^*, b^*),	(a^*, b^*),	(a^*, b^*),	(a^*, b^*),
$(2-a^*, -b^*)$,	$(2-a^*, -b^*)$,	$(-2-a^*, -b^*)$,	$(-2-a^*, -b^*)$,
$(-a^*, 2-b^*)$,	$(-a^*, -2-b^*)$,	$(-a^*, 2-b^*)$,	$(-a^*, -2-b^*)$,
(a^*-2, b^*-2).	(a^*-2, b^*+2).	(a^*+2, b^*-2).	(a^*+2, b^*+2).

Table 3.1: The four extensions of a point (a,b) in different quadrants of the square.

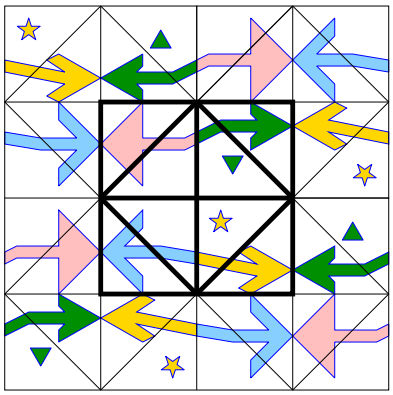


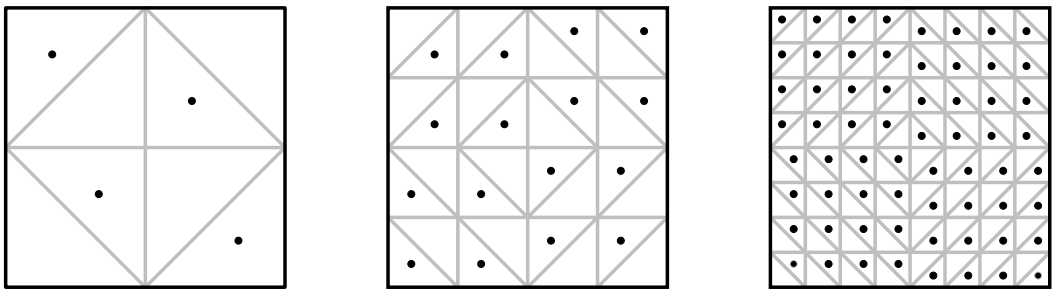
Figure 3.5: Extension of Figure 3.2 to a larger square.

3.3.3 Hexagonal Cells

In Figure 3.3 we showed that when an even number of bits is used for quantization. we can use square-shaped cells for tessellating the square. In this section, we describe an alternative tessellation pattern that uses hexagonal cells instead of square-shaped cells. Notice that each square-shaped cell in Figure 3.3 exactly overlaps two triangular cells in Figure 3.4, and hence two representative points of the same figure. We construct a pattern that uses

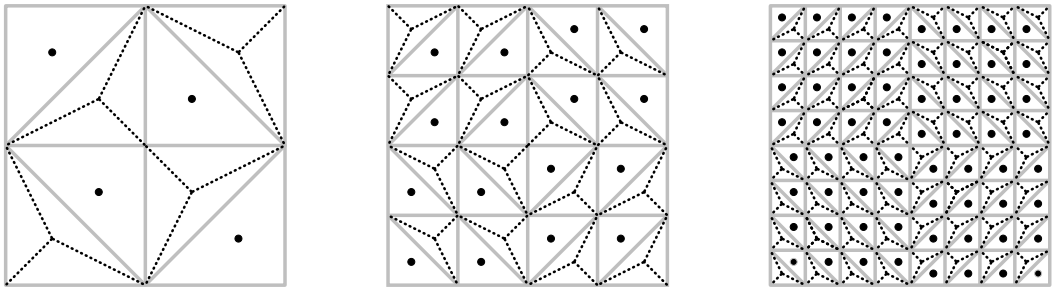
hexagonal cells using a three step process. We start with Figure 3.4 and remove one of the two representative points that are co-located in the same square-shaped cell of Figure 3.3. The resulting pattern is shown in Figure 3.6. Notice that the representative points shown in Figure 3.6(a) appear at the same positions in Figure 3.6(b). Similarly, the representative points shown in Figure 3.6(b) appear at the same positions in Figure 3.6(c). In the second step, the hexagonal tessellation is constructed by replacing each removed point by three edges that connect the point to the vertices of the triangle containing it. These edges are shown using dotted lines in Figure 3.7. From Section 3.3.2 and Figure 3.5, we know that we can extend the image inside the square to a bigger square. In the final step of building the hexagonal pattern we form proper cells using the extension of the square. The resulting pattern is shown in Figure 3.8.

Notice that exactly four cells of each pattern are in fact pentagons, and the rest of the cells are hexagons. In particular, the pattern corresponding to two bits has four pentagons and no hexagons, while the pattern corresponding to four bits has four pentagons and 12 hexagons. If the pattern in Figure 3.8 is projected back to the sphere using an appropriate projection, the spherical hexagonal cells will be almost regular. We later show that this will improve the quantization error.



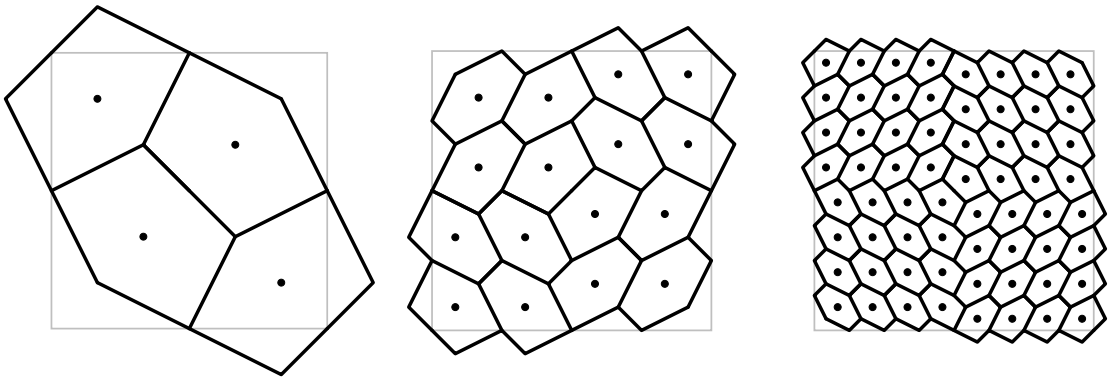
(a) 2 bits (b) 4 bits (c) 6 bits

Figure 3.6: First step in constructing the hexagonal pattern for an even number of bits.



(a) 2 bits (b) 4 bits (c) 6 bits

Figure 3.7: Second step in constructing the hexagonal pattern for an even number of bits.



(a) 2 bits (b) 4 bits (c) 6 bits

Figure 3.8: Hexagonal pattern of representative points for an even number of bits.

3.3.4 Projections for Octahedral Quantization

In this section, we describe a few projections that project the positive spherical octant to the positive face of the octahedron. We describe the Gnomonic, the Areal, the Buss-Fillmore, and the Tegmark projections.

3.3.4.1 Gnomonic Projection

In this projection, the map of a point $P = (a,b,c)$ on the positive face of the octahedron to the point $N = (x,y,z)$ on the unit sphere is the intersection of (i) the line that passes through

the center of the sphere and P, and (ii) the sphere itself. Thus we have,

$$a = \frac{x}{x+y+z} \quad b = \frac{y}{x+y+z} \quad c = \frac{z}{x+y+z},$$

and the inverse relation,

$$x = \frac{a}{\sqrt{a^2+b^2+c^2}} \quad y = \frac{b}{\sqrt{a^2+b^2+c^2}} \quad z = \frac{c}{\sqrt{a^2+b^2+c^2}}.$$

3.3.4.2 Areal Projection

In this section, we describe the Areal projection which is a projection devised by us having the following distance-preserving property. In particular, let p be a point on one of the 12 edges of the octahedron. Let A and B be the end vertices of the edge containing p. Let r, C, and D be the spherical points such that their Areal projections are p, A, and B respectively. The Areal projection has the property that the ratio of the distance between p and A to the distance between p and B is the same as the ratio of the spherical distance between r and C to the spherical distance between r and D:

$$\frac{d(p,A)}{d(p,B)} = \frac{d_S(r,C)}{d_S(r,D)},$$

where $d(\cdot)$ denotes the Euclidean distance on a face of the Octahedron and $d_S(\cdot)$ denotes the Geodesic distance on the sphere.

The Areal projection of a point $N = (x,y,z)$ located on the positive spherical triangle of the unit sphere is the point $P = (a,b,c)$, such that a, b, and c are the ratios of the areas of the three spherical triangles formed by N and the points $X = (1,0,0)$, $Y = (0,1,0)$, and $Z = (0,0,1)$ to the area of the spherical triangle $\triangle XYZ$ as shown in Figure 3.9. Notice that the area of spherical triangle $\triangle XYZ$ is $\frac{\pi}{2}$, *i.e.*, it equals one eighth of the surface area of the

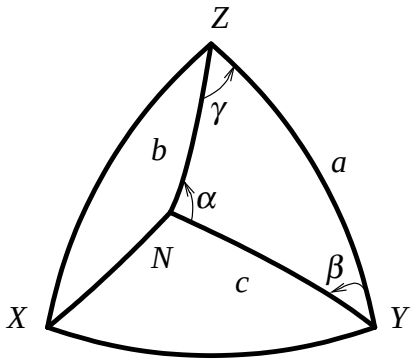


Figure 3.9: The point N inside spherical triangle $\triangle XYZ$.

unit sphere. Letting $A_S(t,u,v)$ denote the area of the spherical triangle $\triangle tuv$, we have:

$$a = \frac{A_S(N,Y,Z)}{\frac{\pi}{2}}; \quad b = \frac{A_S(X,N,Z)}{\frac{\pi}{2}}; \quad c = \frac{A_S(X,Y,N)}{\frac{\pi}{2}}.$$

We can simplify the above equations to (see Appendix A.1):

$$a = \frac{4}{\pi}\arctan\frac{x}{y+z+1}; \quad b = \frac{4}{\pi}\arctan\frac{y}{x+z+1}; \quad c = \frac{4}{\pi}\arctan\frac{z}{x+y+1}. \tag{3.1}$$

To compute $N = (x,y,z)$ from $P = (a,b,c)$, we use the inverse relation (see Appendix A.1):

$$x = \frac{s(a)}{1-s(a)-s(b)-s(c)}; \quad y = \frac{s(b)}{1-s(a)-s(b)-s(c)}; \quad z = \frac{s(c)}{1-s(a)-s(b)-s(c)}, \tag{3.2}$$

where $s(\cdot)$ is defined as:

$$s(u) = \frac{\tan\frac{\pi}{4}u}{\tan\frac{\pi}{4}u+1}. \tag{3.3}$$

Compared to the Gnomonic projection, the Areal projection uses trigonometric functions which are more expensive to compute. However, as we discuss later, the Areal projection has lower quantization errors compared to the Gnomonic projection. The difference in encoding time between the two projections is negligible except encoding for real-time applications. In Section 3.3.5, we introduce the QuickAreal algorithm, a fast implementation of the Octahedral Quantization using the Areal projection. The QuickAreal algorithm

does not use trigonometric functions and hence is suitable for real-time applications.

3.3.4.3 Buss-Fillmore Projection

In this section, we briefly describe the Buss-Fillmore projection which, similar to the Areal projection, preserves the distances along the edges of the Octahedron. Consider a point $N = (x, y, z)$, located on the positive spherical triangle of the unit sphere. we define the Buss-Fillmore projection of N to be the point $P = (a, b, c)$, such that N is the weighted spherical centroid of the points $X = (1, 0, 0)$, $Y = (0, 1, 0)$, and $Z = (0, 0, 1)$ with weights a, b, and c respectively.

Buss and Fillmore [9] describe an iterative algorithm to compute the weighted spherical centroid of an arbitrary number of spherical points. This algorithm is used for the inverse projection of a point from the octahedron to the sphere. On the other hand, the projection from the sphere to the octahedron has a closed form solution. This implies that the Octahedral Quantization using the Buss-Fillmore projection is slow for decoding, but fast for encoding.

3.3.4.4 Tegmark Projection

We also introduce the Tegmark [73] projection, which is an equal area projection that was initially designed for describing equal area pixels on a sphere. Although the Tegmark projection was designed for an Icosahedron, it is straightforward to apply the same technique for an Octahedron, which is used in our Octahedral Quantization method.

3.3.5 QuickAreal Algorithm

In this section, we describe the QuickAreal algorithm, which is a fast algorithm for quantizing unit normal vectors using the Areal projection. The QuickAreal algorithm uses table lookups instead of computing trigonometric functions. We need to mention that hexagonal

cells cannot be used with the QuickAreal algorithm. The discussion here is restricted to triangular and square-shaped cells.

The encoding algorithm works as follows. Equation 3.1 is symmetric with respect to the three coordinates x, y, and z, as well as, a, b, and c, and hence it sufficient to show how a is computed using x, y, and z. Notice that

$$a = \frac{4}{\pi} \arctan \frac{x}{y+z+1}.$$

Hence, to compute a given x, y and z, it suffices to have an algorithm that quickly computes the $\arctan(\cdot)$ function. Moreover, we are only interested in values of a that are quantized. That is, they only have n bits of precision, where n depends on the number of bits used for quantization of the normal vector. Hence, if we precompute the function

$$t(\hat{x}) = \tan \frac{\pi}{4} \hat{x}$$

for all $\hat{x}$, such that $\hat{x}$ is a positive fractional number with n bits of precision. We can search the table for values of $\hat{x}$ that are close to $\frac{x}{y+z+1}$. This can be implemented using binary search, as the function $t(\hat{x})$ monotonic for $0 \leq \hat{x} \leq 1$.

To decode a quantized normal vector, the QuickAreal algorithm uses another table for storing values of

$$s(\hat{x}) = \frac{\tan \frac{\pi}{4} \hat{x}}{\tan \frac{\pi}{4} \hat{x} + 1}$$

for all $\hat{x}$, such that $\hat{x}$ is a positive fractional number with n bits of precision, because the values of a, b and c used in Equation 3.2 have fixed precision.

Unlike the Deering algorithm, the decoding algorithm uses a very compact table. For example, for a quantization using 18 bits, the QuickAreal algorithm needs to precompute $s(\hat{x})$ function for all $2^{18/2-1}$ values of $\hat{x}$. This only requires a 1-kilobyte table, while the Deering algorithm uses a 24-kilobyte table.

3.3.6 Quantization Using a Nearest Neighbor Finding Algorithm

A general method of surface normal vector quantization uses an appropriately chosen set S of representative normals. To quantize a normal n, the nearest neighbor of n in S is chosen as the representative normal of n. In contrast, the Octahedral Quantization method does not guarantee that the representative normal of n is its nearest neighbor. However, we can use a nearest neighbor finding algorithm such as those proposed by Hjaltason and Samet [41] in conjunction with the Octahedral Quantization method. That is, the set S of representative normals are to be the set of representative normals used in the Octahedral Quantization method.

Quantization methods that use a nearest neighbor finding algorithm are not suitable for real-time applications because the encoding process requires searching S for the nearest neighbor of n, which can be expensive. Moreover, the decoding process may require a lot of storage as the three coordinates for each point in S need to be computed and stored.

In this section, we describe a few methods that could be used for generating the set of representative normals S. These methods include Octahedral Quantization based methods (*i.e.*, Gnomonic, Areal, Bass-Fillmore, and Tegmark), the Deering method, the Geographic, Saff-Kuijlaars [60], and the Spherical Centroidal Voronoi Tessellations (SCVT) [18] method.

3.3.6.1 Random Points

Note that the set S can be a set of randomly generated spherical points. We use a random set of points as benchmark to evaluate the quantization quality of other quantization methods that use a nearest neighbor finding algorithm.

3.3.6.2 Saff-Kuijlaars Method

Saff and Kuijlaars [60] describe a mathematical process which places a sequence of points on the unit sphere. The points are placed in a spiral, that is the first point in the sequence is placed on the south pole, and each successive point is placed north of the previous point,

with a suitable longitudinal displacement. Notice that no two points in the sequence have the same latitude (*i.e.*, *z* coordinate). Although the point set generated by this method is easy to compute, it is not straightforward to compute a closed form for deriving the coordinates of the i^{th} point in the sequence without computing all the preceding points.

3.3.6.3 Spherical Centroidal Voronoi Tessellations (SCVT)

Du, Gunzburger, and Ju [18] describe an iterative process for choosing an arbitrary number of points on the unit sphere, such that the points are also the centroids of the cells of the Voronoi [3] tessellations. This method has the desirable property that the generated points are very well distributed on the sphere, and hence suitable for quantization purposes. However, the computation of the points is rather expensive as the algorithm to do so is iterative. In our experiments we used an offline process to generate the set of representative normals, and then used this set for quantizing normals. We need to mention that due to the computational complexity of SCVT, the largest set that we managed contained only 2^{16} normals.

3.3.7 Table of Quantization Methods

Table 3.2 summarizes the different quantization methods discussed in this paper. Quantization methods based on Octahedral Quantization are prefixed by 'OQ' and labeled by the projection technique used and their use of of hexagonal cells. For example, 'OQ-Areal-Hex' refers to an Octahedral Quantization method that uses both the Areal projection and the hexagonal cells. On the other hand, names of methods which use a nearest neighbors algorithm start with 'NN'. The methods names that start with 'NN-OQ' are quantization methods that use a nearest neighbor finding algorithm such that the set *S* of the representative points is the same as an Octahedral Quantization methods. For example, the NN-OQ-Areal-Hex method uses the representative normals of the OQ-Areal-Hex method. Notice that the encoding process of NN-OQ-Areal-Hex method ensures that the quantization error

is minimized, but there is no such guarantee for the OQ-Areal-Hex method.

The Tegmark and Tegmark-Hex methods are the only *equal area* methods in Table 3.2. That is, each representative normal in these methods represents normals that cover an equal area of the unit sphere. Equal areal quantization methods are especially important as they can be used to uniformly sample the unit sphere.

Method's Name	Octahedral	Hexagonal	Nearest Neighbors	Equal Area
Geographic				
Deering				
OQ-Gnomonic	✓			
OQ-Areal	✓			
OQ-Buss-Fillmore	✓			
OQ-Tegmark	✓			✓
OQ-Gnomonic-Hex	✓	✓		
OQ-Areal-Hex	✓	✓		
OQ-Buss-Fillmore-Hex	✓	✓		
OQ-Tegmark-Hex	✓	✓		✓
NN-OQ-Gnomonic			✓	
NN-OQ-Areal			✓	
NN-OQ-Buss-Fillmore			✓	
NN-OQ-Tegmark			✓	
NN-OQ-Gnomonic-Hex		✓	✓	
NN-OQ-Areal-Hex		✓	✓	
NN-OQ-Buss-Fillmore-Hex		✓	✓	
NN-OQ-Tegmark-Hex		✓	✓	
NN-SCVT			✓	
NN-Saff-Kuijlaars			✓	
NN-Random			✓	

Table 3.2: A Summary of Quantization Methods

3.3.8 QuickArealHex Algorithm

In Section 3.5.1 we show that the encoding time of normal quantization methods that use nearest neighbor finding algorithms increases rapidly with the number of quantization bits. In other words, such methods are not suitable when a large number of quantization bits is

required. Moreover, the nearest neighbor based quantization methods are not suitable for applications that require real-time encoding of normal vectors.

In this section, we describe the QuickArealHex algorithm, a fast implementation of the NN-OQ-Areal-Hex quantization method. The QuickArealHex algorithm first uses the QuickAreal algorithm to quickly encode a normal vector q using the OQ-Areal method. Then the nine neighbors of the quantized normal are tested to find the closest representative normal to q. The QuickArealHex algorithm always searches for nine neighbors of a quantized normals, hence it can perform fast encoding compared to implementing the NN-OQ-Areal-Hex algorithm when using a general nearest neighbor finding algorithm.

Figure 3.10 provides an example. The gray dots are the representative points of the OQ-Areal-Hex algorithm. The symbol $\times$ is the projection of the normal q onto the square of Figure 3.8(c). q is mapped to middle square using the QuickAreal algorithm, and the nine neighbors of q are shown with black dots.

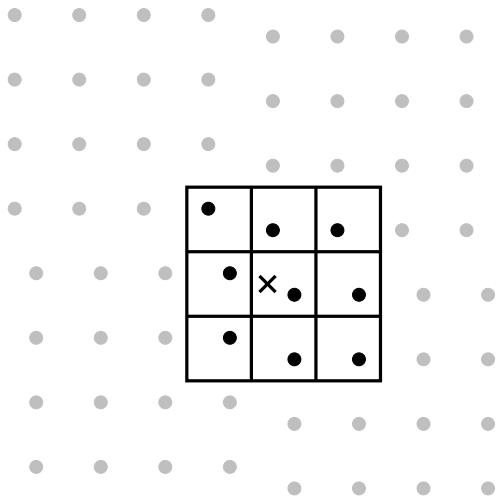


Figure 3.10: The neighborhood search of the QuickArealHex algorithm.

3.4 Lower Bounds

In this section, we provide a loose lower bound for the quantization error. Given a normal vector n and its corresponding quantized normal vector n_q, the quantization error e_n is the defined as the geodesic distance between n and n_q on the unit sphere. We have $e_n = d_S(n, n_q)$, and $e_n = \arccos n \cdot n_q$, where $d_S(n, n_q)$ is the geodesic distance between n_q and n. Three error statistics that are of interest to us are:

1. The Maximum Quantization Error (MQE) is the largest possible value of e_n for a quantization method;

2. The Average Quantization Error (AQE) is the average quantization error for all the normals; and

3. The Root Mean Square Quantization Error (RMSQE) is the square root of the average of the squares of the quantization error for all the normals.

We assume that we have placed M representative normals on the surface of the unit sphere. Consider an arbitrary representative normal q. Let θ_q denote the largest quantization error of normals represented by q. Moreover, let S_{θ_q} denote the surface area of the unit sphere corresponding to the normals represented by q. Notice that, S_{θ_q} is smaller than the surface area of a spherical disk with radius θ_q. On the other hand, the surface area of a spherical disk with radius θ is [79] $2\pi(1-\cos\theta)$, hence $S_{\theta_q} \leq 2\pi(1-\cos\theta_q)$.

Let θ denote the largest θ_q of all representative normals. As we require all the normals on the unit sphere to have corresponding representative normals, we have $4\pi \leq \sum_q S_{\theta_q}$, and hence $4\pi \leq \sum_q 2\pi(1-\cos\theta_q)$. Therefore, as θ is the largest θ_q, we have $4\pi \leq M2\pi(1-\cos\theta)$. The Maximum Quantization Error is equal to θ and is at least

$$\arccos\left(1-\frac{2}{M}\right). \tag{3.4}$$

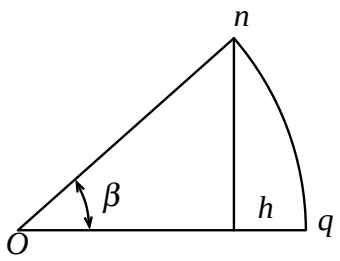


Figure 3.11: Cross section of the unit sphere centered at O.

We now derive the quantization error statistics of normals in a spherical disk with radius θ. Consider an arbitrary normal n in the spherical disk centered at q and radius θ. Assume that the geodesic distance of normal n to its representative normal q is β (as shown in Figure 3.11). The corresponding quantization error of n is β. The total quantization error for all normals located in the disk centered at q and geodesic radius θ is

$$\int_0^\theta \beta \frac{\partial S_\beta}{\partial \beta} d\beta,$$

where S_β is the surface area of the spherical disk of height h. We have [79], $S_\beta = 2\pi h$ and $h = 1 - \cos\beta$.

The Average Quantization Error (AQE) is

$$\frac{1}{S_\theta} \int_0^\theta \beta \frac{\partial S_\beta}{\partial \beta} d\beta = \frac{M}{2}(\sin\theta - \theta\cos\theta). \tag{3.5}$$

Similarly, the mean square quantization error is

$$\frac{1}{S_\theta} \int_0^\theta \beta^2 \frac{\partial S_\beta}{\partial \beta} d\beta = \frac{M}{2}((2-\theta^2)\cos\theta + 2\theta\sin\theta - 2),$$

and the Root Mean Square Quantization Error (RMSQE) is

$$\sqrt{\frac{M}{2}((2-\theta^2)\cos\theta + 2\theta\sin\theta - 2)}. \tag{3.6}$$

3.4.1 A Tighter Lower Bound

We can derive a tighter lower bound for the maximum quantization error (MQE) by observing the geometric properties of the spherical Voronoi diagram of the representative normals. A spherical Voronoi diagram of the spherical point set S is a tessellation on the surface of the unit sphere, such that all the points in each cell of the tessellation have the same nearest neighbor in S. The cells of a spherical Voronoi tessellation could be of different shapes. We point out that the lowest MQE is achieved by having cells that are as close to a circle as possible. In general, tessellation of a sphere by using only circles is not possible. Furthermore, the sphere cannot be tessellated by any regular polygon with more than six sides. In this section, we can derive an MQE lower bound by assuming a tessellation of the unit sphere with equal regular hexagons. This lower bound is tighter than the lower bound in Equation 3.4. However, as it is not possible to completely cover the sphere only with regular hexagons, the lower bound derived in this section is not the tightest possible bound. We derive the bounds for the general case of tessellating the sphere into M equal N-sided regular spherical polygons. The MQE lower bound corresponds to the the case of $N = 6$.

The area of each cell is $\frac{4\pi}{M}$. Letting 2γ denote the interior angle of the cells, the area of the cell is [79] $2N\gamma - (N-2)\pi$. Hence,

$$\frac{4\pi}{M} = 2N\gamma - (N-2)\pi.$$

Or,

$$\gamma = \frac{\pi}{2} - \frac{\pi}{N}\left(1 - \frac{2}{M}\right).$$

Let A denote the center of a cell, and let B and C be two adjacent vertices of the cell. Without loss of generality, we assume that the sphere is oriented such that, for properly

chosen values of α and ψ, we have:

$$\begin{aligned} A &= (0,0,1) \\ B &= (\sin\alpha\cos\psi, \sin\alpha\sin\psi, \cos\alpha) \\ C &= (\sin\alpha\cos\psi, -\sin\alpha\sin\psi, \cos\alpha)\,. \end{aligned}$$

Consider the spherical triangle ABC depicted in Figure 3.12. Let $\angle A$ denote the spherical angle at vertex A and let a denote the spherical side BC. Hence, as A is the center of a regular spherical polygon, we have $\angle B = \gamma$, and $\angle A = \frac{2\pi}{N}$. Moreover, notice that the maximum quantization error is equal to α.

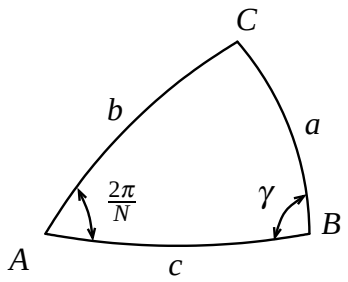


Figure 3.12: Tighter lower bounds.

As a is the side of the spherical triangle between B and C, we have:

$$\begin{aligned} \cos a &= B\cdot C = \sin^2\alpha\cos^2\psi - \sin^2\alpha\sin^2\psi + \cos^2\alpha \\ 1 - 2\sin^2\frac{a}{2} &= \sin^2\alpha(1-\sin^2\psi) - \sin^2\alpha\sin^2\psi + 1 - \sin^2\alpha \\ &= 1 - 2\sin^2\alpha\sin^2\psi \\ \sin\frac{a}{2} &= \sin\alpha\sin\psi\,. \end{aligned}$$

Similarly, for b and c, we have: $\cos b = A\cdot C = \cos\alpha$ and $\cos c = A\cdot B = \cos\alpha$. Hence, $b = c = \alpha$.

Using the law of cosines in the spherical triangle ABC, we have: $\cos\angle A\sin c\sin b +$

$\cos b \cos c = \cos a$, hence:

$$\begin{aligned}\cos\angle A &= \frac{\cos a - \cos b\cos c}{\sin b \sin c} \\ &= \frac{\sin^2\alpha\cos^2\psi - \sin^2\alpha\sin^2\psi + \cos^2\alpha - \cos^2\alpha}{\sin^2\alpha} \\ &= \cos^2\psi - \sin^2\psi = \cos 2\psi\,.\end{aligned}$$

Hence, $\psi = \frac{\pi}{N}$.

Similarly, we have:

$$\begin{aligned}\cos\angle B &= \frac{\cos b - \cos a\cos c}{\sin a\sin c} = \frac{\cos\alpha - \cos\alpha\cos a}{\sin a\sin\alpha} \\ &= \frac{\cos\alpha}{\sin\alpha}\frac{1-\cos a}{\sin a} = \frac{\cos\alpha}{\sin\alpha}\frac{2\sin^2\frac{a}{2}}{2\sin\frac{a}{2}\cos\frac{a}{2}} \\ &= \frac{\cos\alpha}{\sin\alpha}\frac{\sin\frac{a}{2}}{\cos\frac{a}{2}} = \frac{\cos\alpha}{\sin\alpha}\frac{\sin\alpha\sin\psi}{\cos\frac{a}{2}} \\ &= \sin\psi\frac{\cos\alpha}{\cos\frac{a}{2}}\,.\end{aligned}$$

Hence, as $\angle B = \gamma$, we have

$$\cos\frac{a}{2} = \sin\psi\frac{\cos\alpha}{\cos\gamma}\,.$$

Moreover, as $\sin^2\frac{a}{2} + \cos^2\frac{a}{2} = 1$, we have

$$\begin{aligned}\sin^2\alpha\sin^2\psi + \sin^2\psi\frac{\cos^2\alpha}{\cos^2\gamma} &= 1 \\ \sin^2\psi\left(\sin^2\alpha + \frac{1-\sin^2\alpha}{1-\sin^2\gamma}\right) &= 1,\end{aligned}$$

which simplifies to

$$\sin^2\alpha = \frac{\sin^2\psi + \sin^2\gamma - 1}{\sin^2\psi\sin^2\gamma}\,.$$

In case of a hexagonal cells, we have $N = 6$, $\psi = \frac{\pi}{6}$, $\gamma = \frac{\pi}{3}(1+\frac{1}{M})$, and hence the maximum

quantization error (α) is obtained from

$$\sin^2\alpha = 4 - \frac{3}{\sin^2\frac{\pi}{3}\left(1+\frac{1}{M}\right)}.$$

This simplifies to:

$$\cos\alpha = \frac{\sqrt{3}}{\tan\frac{\pi}{3}\left(1+\frac{1}{M}\right)}. \tag{3.7}$$

Equations 3.5–3.7 could be used to design the minimum number of quantized normals to achieve a specific quantization error. For example to achieve a maximum quantization error (α) less than 1 degree, requires at least 15,879 representative normals. Moreover, we know from Equation 3.5 that the average quantization error for 15,879 representative normals will be at least 0.6062 degrees and from Equation 3.6 we know that the root mean square quantization error for 15,879 representative normals will be at least 0.6430 degrees.

3.5 Comparison of Quantization Methods

Our test setup generates 128×2^Q random unit normal *test vectors* with a uniform distribution, where Q is the number of quantization bits. Note that the number of test vectors is proportional to the number of the representative normals. For example, if $Q = 12$, we generate 524,288 random unit test vectors. We then use Q bits to encode each test vector and then decode it to derive its corresponding *quantized vector*. The quantization error for each test vector is the angle between the test vector and its corresponding quantized vector and is measured in degrees. The quantization errors are then aggregated over all the test vectors to produce the various error statistics. We also measure the average time taken for decoding and encoding a test vector. The experiments were conducted on an IBM Thinkpad T43 machine with an Intel Centrino 750 (1.86 GHz and 2 MB Level-2 cache) and 1GB of RAM running Windows XP using the Visual C++ optimizing compiler.

In Figure 3.13, we compare the quantization error of several quantization methods.

In particular, we compare the quantization errors of the Geographic, the Deering, the OQ-Areal, the OQ-Gnomonic, the OQ-Tegmark, and the OQ-Buss-Fillmore quantization methods. In particular, we compare the Maximum Quantization Error (MQE), the Average Quantization Error (AQE), and the Root Mean Square Quantization Error (RMSQE) of the above mentioned quantization methods and the corresponding lower bounds. Notice that the MQE lower bound is derived in Section 3.4.1, while the AQE and the RMSQE lower bounds are derived in Section 3.4. The quantization error is shown on the left *y*-axis of Figure 3.13, while on the right *y*-axis of Figures 3.13(a, c, e), the quantization error is represented in terms of *effective bits*. We use the concept of a *perfect quantizer* to explain the term "effective bits". A perfect quantizer is a quantizer whose error statistics are the same as the lower bounds. The effective bits corresponding to a quantization error is the number of bits required by a perfect quantizer to obtain the same quantization error. For example, we can see that the MQE error of the Deering method when using 12 quantization bits corresponds to 9 effective bits. That is, the same quantization error could have been achieved if 9 bits were used with a perfect quantizer. In effect, this shows that the Deering method wastes three bits. We also define a corresponding normalized error metric for each of the three error metrics by dividing each error statistics by its corresponding lower bound. The normalized error metrics are dimensionless and are used in Figures 3.13(b, d, e)–3.17(b, d, e).

In Figure 3.13(a), we see that the Deering method has much higher quantization error compared to other methods. In particular, the Deering method wastes three bits while other methods waste between one and two bits. Interestingly, the Geographic method has the lowest MQE error among all the methods. Note that the Normalized Maximum Quantization Error shown in Figure 3.13(b) depends on the parity of the number of bits. In particular, all the methods have lower normalized error when the number of bits is an odd number. Note that the Deering method is only defined for an even number of quantization bits. Moreover, the normalized quantization errors of the Deering method rapidly increases

when the number of bits is increased. In contrast, the normalized quantization errors of other methods remain the same.

From Figures 3.13(c, d) we immediately notice that the Deering method wastes two bits for the AQE and the RMSQE errors, while other methods waste less than one bit. However, Figures 3.13(d, f) show that the Geographic method performs worse than other methods for the AQE and the RMSQE error metrics. Also note that the variation of the errors with respect to the parity of the number of bits is only observed in the Geographic method.

In Figure 3.14, we compare the quantization error of Octahedral Quantization methods that use hexagonal cells and the Geographic method. Note that hexagonal cells are only defined for an even number of bits. In particular, we compare the quantization error of the Geographic, the OQ-Gnomonic-Hex, the OQ-Areal-Hex, the OQ-Buss-Fillmore-Hex, and the OQ-Tegmark-Hex quantization methods.

In Figure 3.14(a), we see that the Geographic method has a higher quantization error compared to other methods. In particular, the Geographic method wastes almost 1.5 bits, while other methods waste a little less than one bit. By comparing Figure 3.14(a) with Figure 3.13(a), we conclude that using hexagonal cells improves the quantization by almost one bit. This improvement is more evident by comparing Figure 3.14(b) with Figure 3.13(b). Note that the OQ-Buss-Fillmore and the OQ-Areal methods have very similar quantization errors. This is not entirely surprising as the Buss-Fillmore and the Areal projections approximate each other. Figure 3.14(b) also shows that the NMQE error metric of the OQ-Tegmark-Hex method increases with an increasing number of quantization bits, while the converse is true for the OQ-Buss-Fillmore-Hex and the OQ-Areal-Hex methods. Moreover, the NMQE error metric of the OQ-Gnomonic-Hex method remains the same. Notice that in Figures 3.14(c)–(f), the OQ-Buss-Fillmore-Hex and the OQ-Areal-Hex methods are very close to the lower bounds.

In Figures 3.15–3.17, we compare the quantization error of quantization methods that

use a nearest neighbor finding algorithm. In particular, we compare the quantization error of the NN-Geographic, the NN-Deering, the NN-OQ-Gnomonic, the NN-OQ-Areal, the NN-OQ-Buss-Fillmore, the NN-OQ-Tegmark, the NN-OQ-Gnomonic-Hex, the NN-OQ-Areal-Hex, the NN-OQ-Buss-Fillmore-Hex, the NN-OQ-Tegmark-Hex, the NN-SCVT, and the NN-Saff-Kuijlaars quantization methods. By comparing Figures 3.15 and 3.16 with Figures 3.13 and 3.14, we observe that methods which use a nearest neighbor finding algorithm have better quantization error. For example, the MQE error metric of the Deering method improves by at least half a bit by using an nearest neighbor finding algorithm.

By comparing Figure 3.15(b) with Figure 3.13(b), we observe that the NN-OQ-Gnomonic method when using an even number of quantization bits, has a lower NMQE error metric compared to OQ-Gnomonic method. A similar observation can be made regarding the OQ-Areal, the OQ-Buss-Fillmore, and the OQ-Tegmark methods. Note that OQ-Tegmark method gains the most from using a nearest neighbor finding algorithm. However the NN-OQ-Buss-Fillmore method has the lowest error metrics. Another interesting observation is that the Geographic quantization does not gain from using a nearest neighbor finding algorithm.

By comparing Figure 3.16(b) and Figure 3.14(b), we observe that the NN-OQ-Gnomonic-Hex method has the same NMQE error metric compared to the OQ-Gnomonic-Hex method. However, the MQE error metric of the OQ-Areal-Hex, the OQ-Buss-Fillmore-Hex, and the OQ-Tegmark-Hex methods gain close to half a bit by using a nearest neighbor finding algorithm.

Finally, we observe from Figure 3.17(b) that the MQE error of the NN-Saff-Kuijlaars method is slightly more than the NN-OQ-Tegmark-Hex method. However the quantization error of the NN-SCVT method is slightly better than both the NN-Saff-Kuijlaars and the NN-OQ-Tegmark-Hex method. The most important observation that we make is that the NN-OQ-Buss-Fillmore-Hex method has the lowest MQE error metric among all the quantization methods discussed in this article. Notice that the MQE error of the NN-OQ-Areal-

Hex method is just a little higher than the NN-OQ-Bass-Fillmore-Hex method. Moreover, we note that the AQE and the RMSQE errors shown in Figures 3.17(c)–(f), are better for the NN-SCVT and NN-Saff-Kuijlaars methods. However, the AQE and RMSQE errors of the NN-OQ-Buss-Fillmore-Hex and the NN-OQ-Areal-Hex methods are just slightly higher than the NN-SCVT and the NN-Saff-Kuijlaars methods.

Based on our observation of the errors of the different quantization methods, we claim that the NN-OQ-Areal-Hex method is consistently among the better methods with respect all the error metrics we considered in our study. We later show that the time and the storage required for quantization using the OQ-Areal and the NN-OQ-Areal-Hex, when implemented using the QuickAreal and QuickArealHex algorithms compares favorably to all other methods.

3.5.1 Encoding and Decoding Times

We now discuss the computation time required for normal quantization. In Figure 3.18, we have selected a few of the quantization methods for the evaluation of the encoding and decoding performance. The methods chosen cover a variety of the methods discussed before. The methods are the Geographic, the Deering, the OQ-Tegmark, the OQ-Gnomonic, the OQ-Areal, the OQ-Areal-Hex the NN-OQ-Areal-Hex, the NN-SCVT, the QuickAreal, the QuickArealHex. We mention the following regarding the evaluation:

- The implementation of the Deering method used in our evaluation is based on the conversion of Deering's Java implementation to C++.
- We used the ANN [53] library and used its k-d tree [5] based nearest neighbor finding algorithm for implementing quantization methods which use a nearest neighbor finding algorithm.
- The SCVT point set was computed offline.

Figure 3.18(a) shows the encoding time. We remind the reader that *encoding* refers to the process of converting a normal vector to its quantized representation. We observe that the Deering method performs poorly for a higher number of quantization bits. Techniques that use a nearest neighbor finding algorithm, with the exception of QuickArealHex, show a drastic degradation in performance when the number of bits increases beyond a threshold (14 bits in our experiments). One reason for this performance degradation is the large amount of memory needed to store the set of representative normals and the associated data structure. While QuickArealHex is slower than the methods that do not use a nearest neighbor algorithm, it has a constant time performance and hence is comparable to such methods. We also point out that the QuickAreal algorithm has the lowest encoding time, even when compared with the OQ-Gnomonic method.

Figure 3.18(b) shows the decoding time. We observe that with the exception of QuickArealHex, the methods that use a nearest neighbor finding algorithm are very fast. This is not surprising as the decoding component of these algorithms is implemented using a table look-up algorithm. However, this speed comes at the expense of the extra storage required to keep the table in memory. That size of the table is 12 bytes times the number of representative normals. For example, if we use 18 bits for quantization, the table has a size of 3 Megabytes. Note that the decoding component of the Deering method also uses a table lookup. However, the size of the table used in the Deering method (24 kilobytes) is 128 times smaller as the Deering method takes advantage of the symmetry of the representative normals. QuickAreal and QuickArealHex have essentially the same decoding time, although QuickArealHex is slightly slower as it requires a few extra operations. The OQ-Gnomonic method is faster than the OQ-Areal method, but is slower than QuickAreal, as it uses a square root operation. The OQ-Areal method uses the more expensive trigonometric functions. The QuickAreal algorithm uses a compact table instead of computing trigonometric functions used in the OQ-Areal method. The Geographic method also uses trigonometric functions, however it is faster than the OQ-Areal method. Note that the OQ-

Areal-Hex and the OQ-Areal methods have similar decoding times. Finally, OQ-Tegmark has the slowest decoding time as it uses a very complex mathematical formulation. Note that the decoding time of all methods could be made similar to the methods that use a nearest neighbor finding algorithm by storing the quantized normal vectors in a table.

3.6 Rendering a Perfect Sphere

In this section, we show how the different quantization methods discussed in this article affect the quality of rendered geometry models. A sphere is the only geometry model discussed in this article. We chose a sphere as it is a smooth surface, and hence the artifacts resulting from the quantization are easily observable. Moreover, the surface of a sphere is equivalent to the entire set of unit normal vectors, and it allows visualization of the quantization artifacts on the entire set of unit normal vectors. We used the POV-Ray software — modified to quantize the surface normals of a sphere — to produce the images in this section.

Figures 3.19–3.26 show a sphere with different quantization methods. Rendering a sphere with quantized surface normals produces visual artifacts which are equivalent to a tessellation of the sphere. In order to show the complete tessellation of the sphere, the sphere is rendered from two different viewpoints. The rendered sphere has a unit radius and is centered at $(0,0,0)$. The two viewpoints are chosen such that the center of each image corresponds to the points $(0,0,1)$ and $(\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}})$.

Figure 3.19(a, b) shows the rendered sphere using the Geographic, and the NN-Geographic quantization methods with 8 quantization bits. The Geographic method produces a tessellation such that its cells are rectangular around the equator and triangular at the poles. The cells around the poles are much smaller than the ones around the equator. We observe that the NN-Geographic method does not noticeably affect on the shape of the cells.

Figures 3.19(c, d) show the rendered sphere using the Deering, and the NN-Deering

methods with 8 quantization bits. The Deering method produces a tessellation such that its cells are rectangular, but they are larger than the tessellation cells of the Geographic method. The Deering method produces cells that are irregular in shape, although the shape of the cells is similar to a square. Notice that using the NN-Deering method considerably changes the shape of the cells.

Figures 3.20–3.23 shows the rendered sphere using Octahedral Quantization methods with 8 and 7 bits of quantization. The OQ-Tegmark, the NN-OQ-Tegmark, the OQ-Gnomonic, and the NN-OQ-Gnomonic methods are used in Figures 3.20 and 3.22, while the OQ-Areal, the NN-OQ-Areal, the OQ-Buss-Fillmore, and the NN-OQ-Buss-Fillmore methods are used in Figures 3.21 and 3.23. Octahedral Quantization methods that do not use a nearest neighbor finding algorithm produce cells that vary from diamonds to squares when 8 bits are used for quantization and produce triangular cells when 7 bits are used for quantization. However, when a nearest neighbor finding algorithm is used, the cells are of varying shapes. Moreover, the tessellation of the sphere in the bottom row of Figures 3.20, and 3.21 contain pentagons and hexagons.

Notice the similarity of the tessellations produced by the Areal and the Buss-Fillmore projections, as shown in Figures 3.21 and 3.23. This is not surprising as the two projections approximate each other.

Notice that the OQ-Gnomonic method produces cells that greatly vary in size at different areas of the sphere (Figures 3.20(a) and 3.22(a)). On the other hand, the OQ-Tegmark method always produces cells of the exact same size, although of different shapes (Figures 3.20(c) and 3.22(c)). The OQ-Areal and the OQ-Buss-Fillmore methods produce cells that do not vary in size as much as the OQ-Gnomonic method (Figures 3.21 and 3.23).

Figures 3.24(a, b) show the rendered sphere with 8 bits of quantization, using using the OQ-Areal, the NN-OQ-Areal quantization methods. Figures 3.24(c, d) show the rendered sphere using using the OQ-Areal-Hex, the NN-OQ-Areal-Hex quantization methods. Notice that Figure 3.24(c) which uses hexagonal cells is very similar to Figure 3.24(b) which

does not use hexagonal cells but instead uses a nearest neighbor finding algorithm. Moreover, by comparing Figure 3.24(c) with Figure 3.24(d), we observe that the shape of the cells in Figure 3.24(d) are more regular.

Figure 3.25 shows the rendered sphere using various quantization method that use a nearest neighbor finding algorithm. Figure 3.25 is generated by using 8 quantization bits. Figure 3.25(a) shows the sphere when random points on the sphere are chosen as the set of the representative normals. Although the cells are very irregular in shape, their sizes are smaller than the Deering method shown in Figure 3.19(c).

The NN-Saff-Kuijlaars method is shown in Figure 3.25(b). Notice that the image in the top row shows that the cells are arranged in a spiral starting from the center of the image. Moreover, the cells of the NN-Saff-Kuijlaars method are also irregularly shaped.

The NN-SCVT method, shown in Figure 3.25(c), has cells which are the most regularly shaped among all the other methods shown in Figure 3.25.

Figure 3.26 shows the rendered sphere using the Deering and the NN-OQ-Areal-Hex quantization method for 10, 12, 14, 16, and 18 bits of quantization. As we can see, the spheres rendered with the NN-OQ-Areal-Hex method with 14 or more quantization bits is almost visually comparable to a perfect sphere whose surface normals are not quantized. However, the Deering method needs at least 16 quantization bits to produce the same effect. Consequently, we claim that the NN-OQ-Areal-Hex method is a two bit improvement over the Deering algorithm.

Notice that the surface color of the sphere in Figure 3.26 varies very gradually over the the sphere. Hence, small quantization errors were not observable. We use a sphere illuminated with a specular light in order to reveal finer visual artifacts of a quantization method. The addition of a specular light to the previously discussed sphere model results in Figure 3.27(a). Notice that there is a small specular highlight in the center of the image. The specular highlight has been magnified and shown in Figure 3.27(b). Figure 3.28 illustrates discuss the fine-grained visual artifacts that could be produced when quantizing

surface normals. For each quantization method, the sphere, is rendered from two different viewpoints. The rendered sphere has a unit radius and is centered at $(0,0,0)$. The two viewpoints are chosen such that the center of each image corresponds to the points $(0,0,1)$ and $(\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}})$ (labeled by 'R'). The first two rows of Figure 3.28 are rendered using the Deering method. Notice the sudden change in the shape of the tessellation when using 18 bits compared to when using 16 bits. The third and fourth rows of Figure 3.28 show the artifacts produced by the OQ-Gnomonic-Hex method. Notice that the OQ-Gnomonic-Hex method does not produce similarly sized cells at different locations of the sphere. On the other hand, the OQ-Areal-Hex-NN algorithm produces similarly sized cells across the sphere. The NN-SCVT method produces similarly shaped and similarly sized cells across the sphere. However, the NN-SCVT method is not practical for more than 16 bits, as we were unable to a generate the set of representative normals for a quantization of more than 16 bits.

3.7 Summary and Conclusion

In this paper, we have investigated a wide variety of techniques for unit normal vector quantization. We reviewed some techniques that are currently in use, and also proposed several novel methods. We provided loose theoretical lower bounds on the quantization error using three different error metrics. We also discussed how different quantization methods affect the rendering of geometry models. Our main finding is the recommendation of using the QuickArealHex algorithm as it has a low quantization error, and is computationally efficient. The factors contributing to the low quantization error of the QuickArealHex algorithm are:

1. It uses hexagonal cells, and we have shown that using hexagonal cells in an Octahedral Quantization method lowers the quantization error.

2. It uses a nearest neighbor algorithm for encoding each normal vector, which means

that it provides the lowest quantization error over all error metrics for the normal vector.

3. Its use of the Areal projection leads to an approximation of the Buss-Fillmore projection, whose incorporation in the NN-OQ-Buss-Fillmore-Hex method resulted in the lowest quantization error as measured by the MQE error metric.

Moreover, the QuickArealHex algorithm is fast with low memory requirements. This is because it uses the QuickAreal algorithm, which as pointed out earlier, is very fast and uses little memory. In particular, for 18 bits of quantization. the QuickArealHex algorithm requires only 1 kilobytes of memory for decoding and an extra 1 kilobytes of memory for encoding, while the Deering method requires 24 kilobytes for each of the decoding and encoding processes. The extra memory used by the Deering method enables a slightly faster decoding time than the QuickAreal method due to the use of table lookup, which could also be used by the QuickAreal method to obtain comparable behavior.

In addition, the nearest neighbor algorithm used in the QuickArealHex algorithm takes advantage of the fact that the representative normals are almost regularly distributed, and hence makes it possible to find the nearest neighbor of a point in constant time.

Figure 3.29 compares the Deering and the QuickArealHex methods. As illustrated in Figure 3.29(a) (which is in terms of relative magnitude—that is, a factor of x), the quantization error of the QuickArealHex is far better than the Deering methods as the number of bits gets large with the change coincidentally occurring at 18 bits, which is the number of bits for which the Deering method is usually used. This increase in quality is achieved while having a faster encoding time (see Figure 3.29(b)).

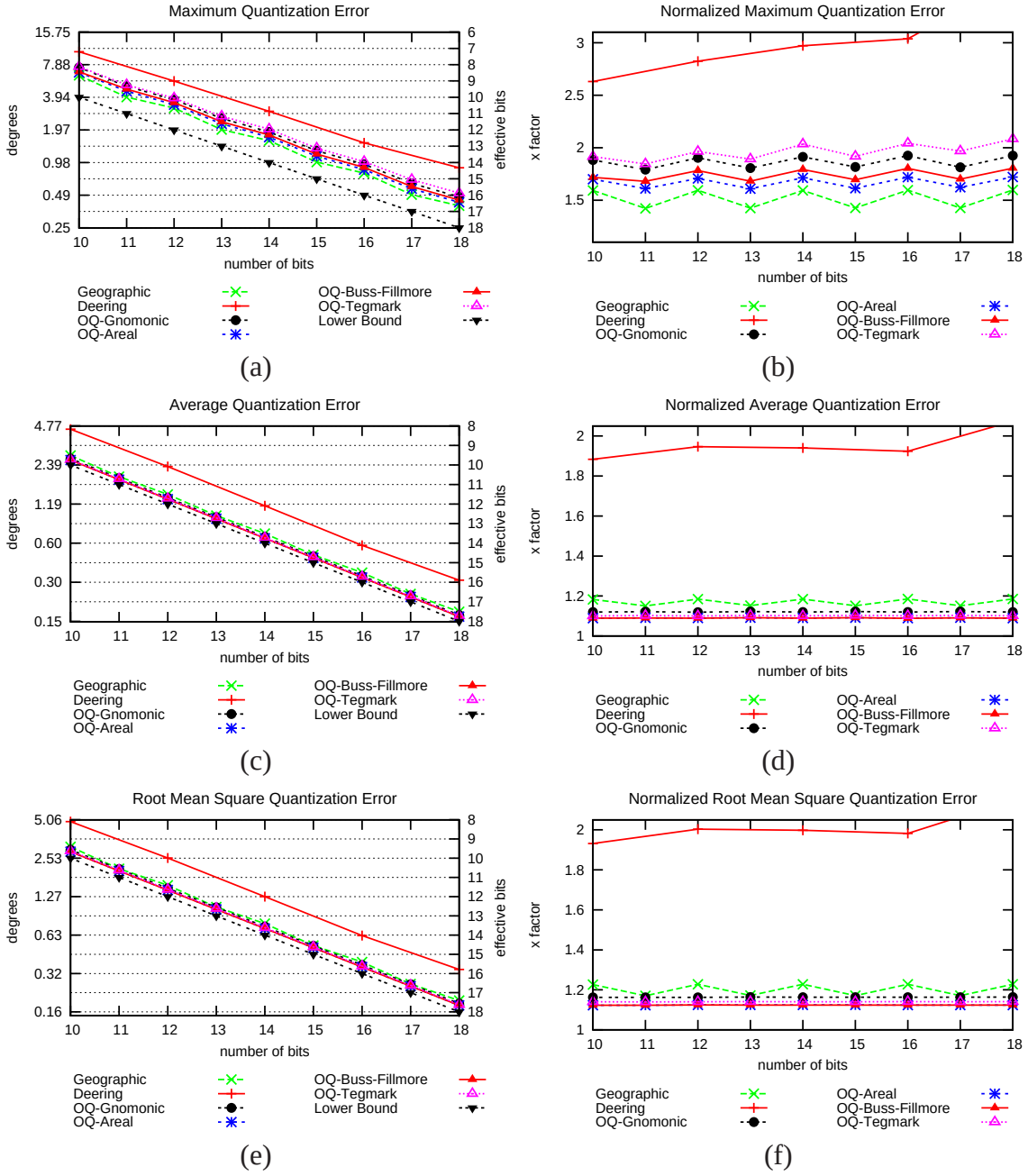


Figure 3.13: Different error statistics of the Deering, the Geographic, the OQ-Gnomonic, the OQ-Areal, the OQ-Buss-Fillmore, and the OQ-Tegmark quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error.

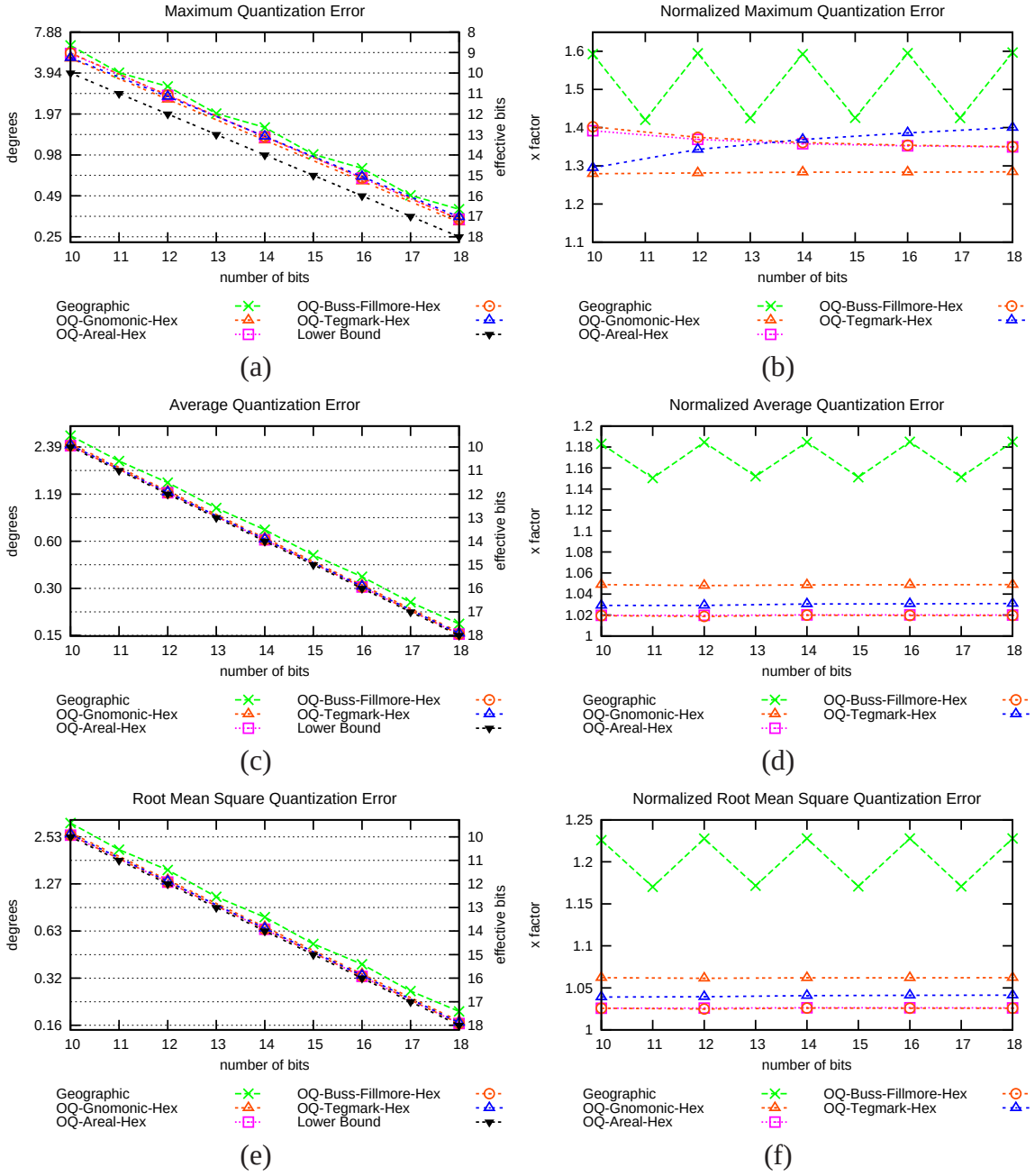


Figure 3.14: Different statistics of the Geographic, the OQ-Gnomonic-Hex, the OQ-Areal-Hex, the OQ-Buss-Fillmore-Hex, and the OQ-Tegmark-Hex quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error.

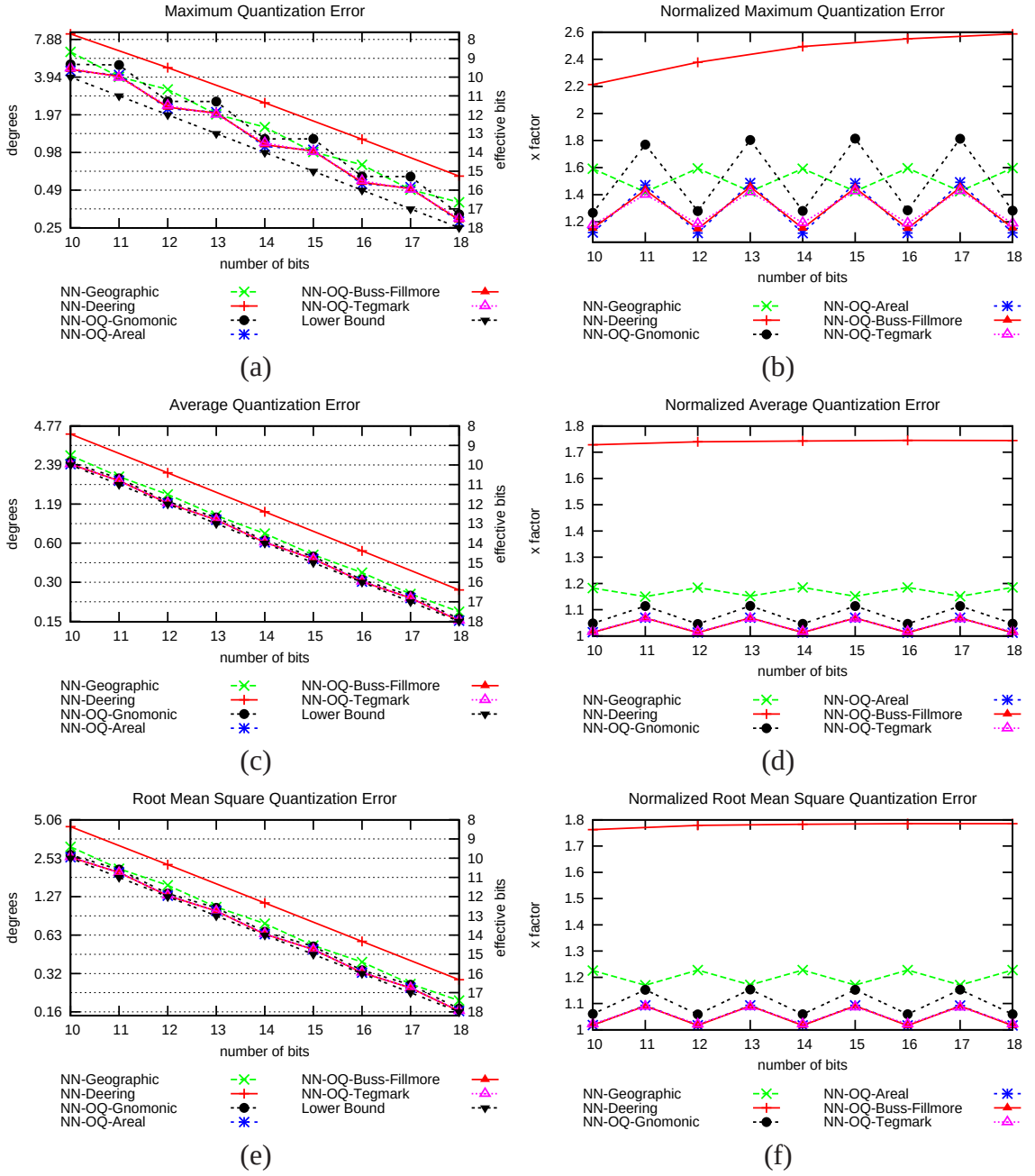


Figure 3.15: Different statistics of the NN-Deering, the NN-Geographic, the NN-OQ-Areal, the NN-OQ-Buss-Fillmore, and the NN-OQ-Tegmark, quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error.

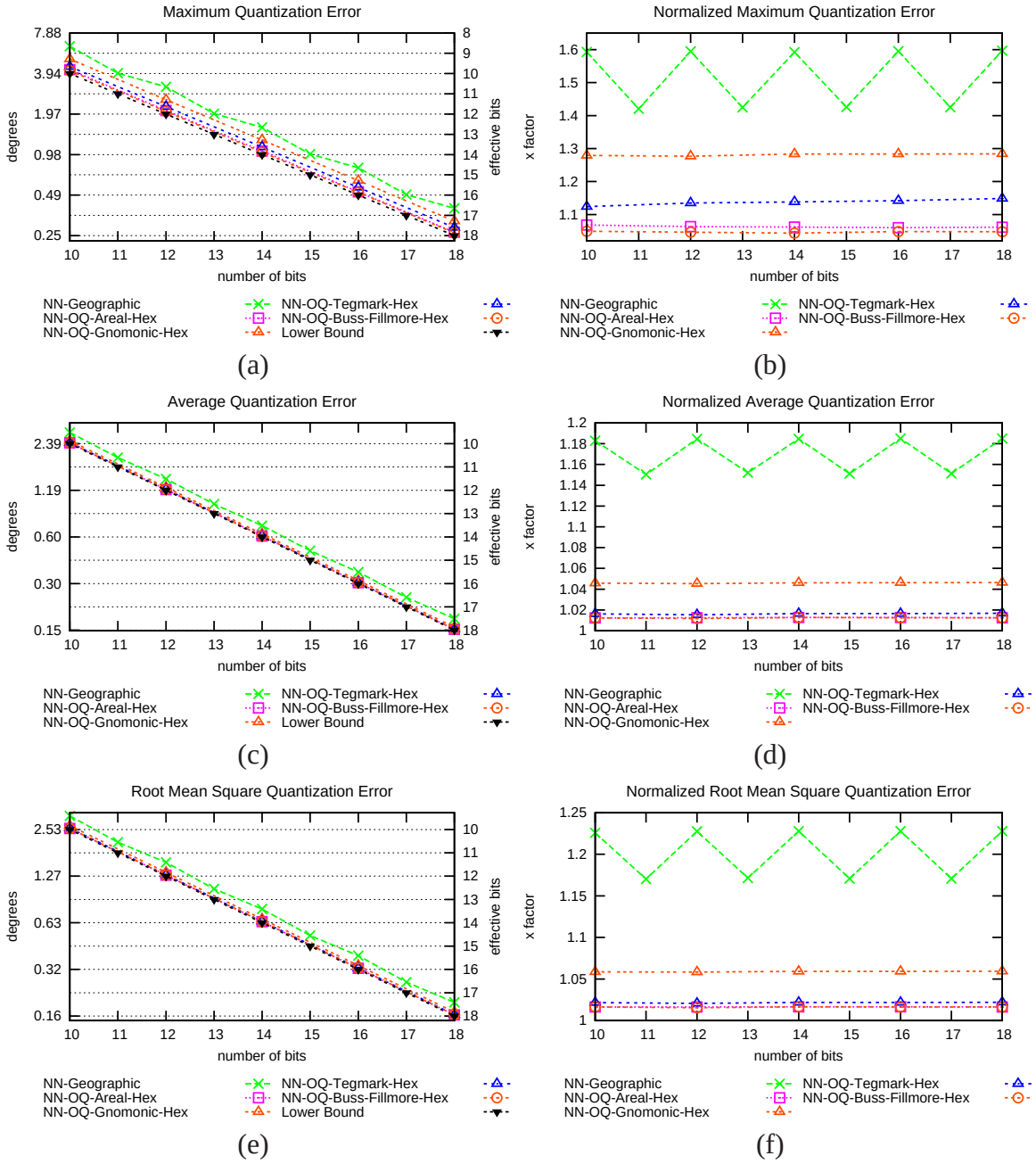


Figure 3.16: Different statistics of the NN-Deering, the NN-Geographic, the NN-OQ-Gnomonic-Hex, the NN-OQ-Areal-Hex, the NN-OQ-Buss-Fillmore-Hex, and the NN-OQ-Tegmark-Hex quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error.

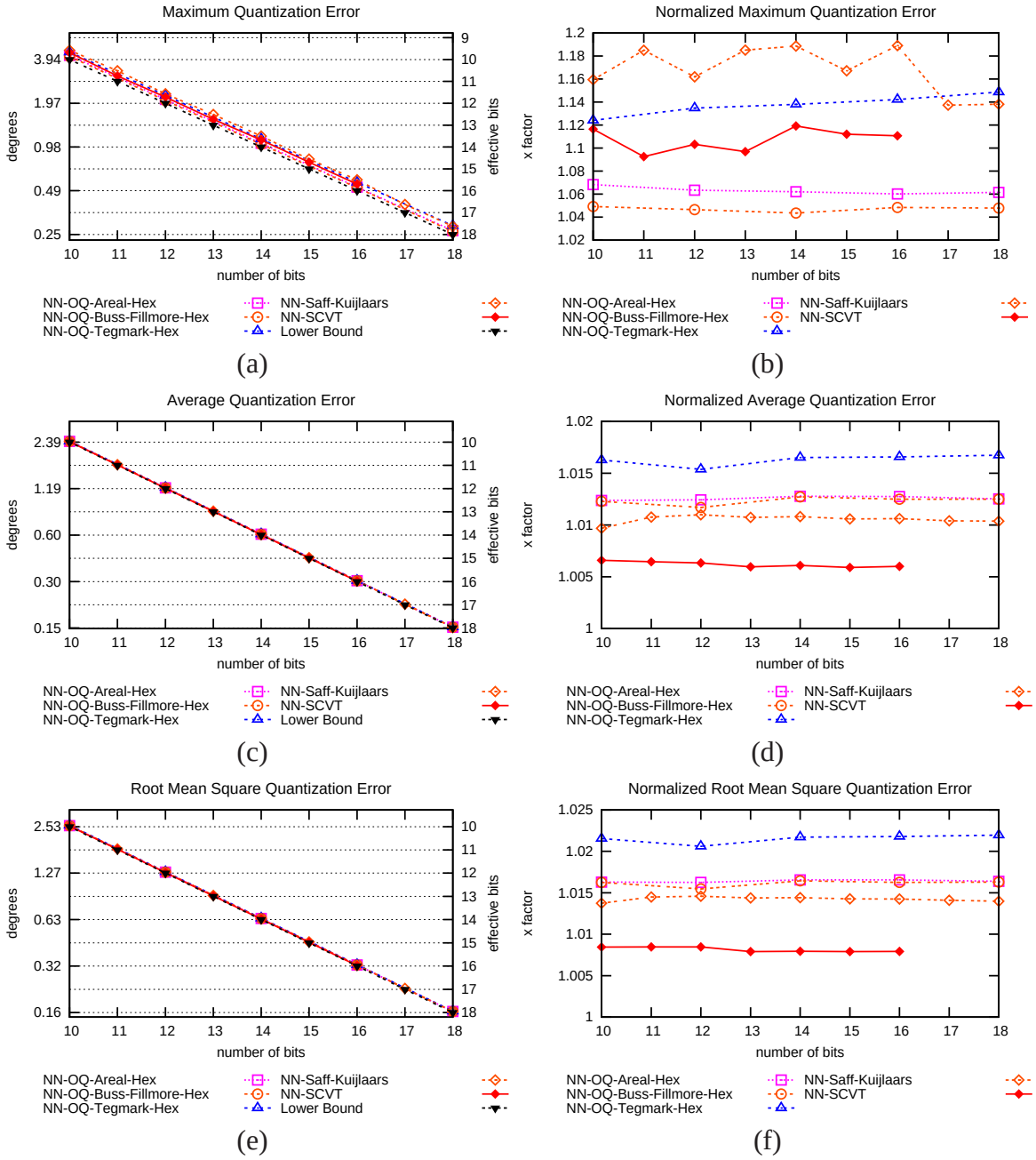
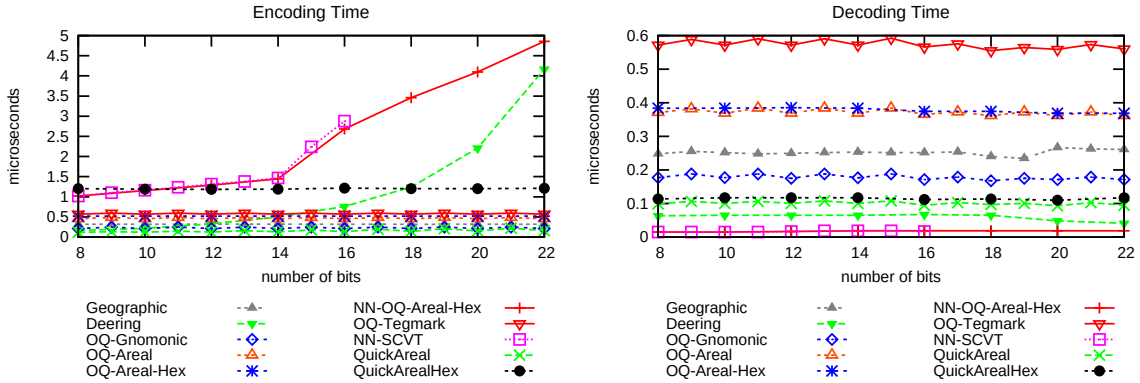
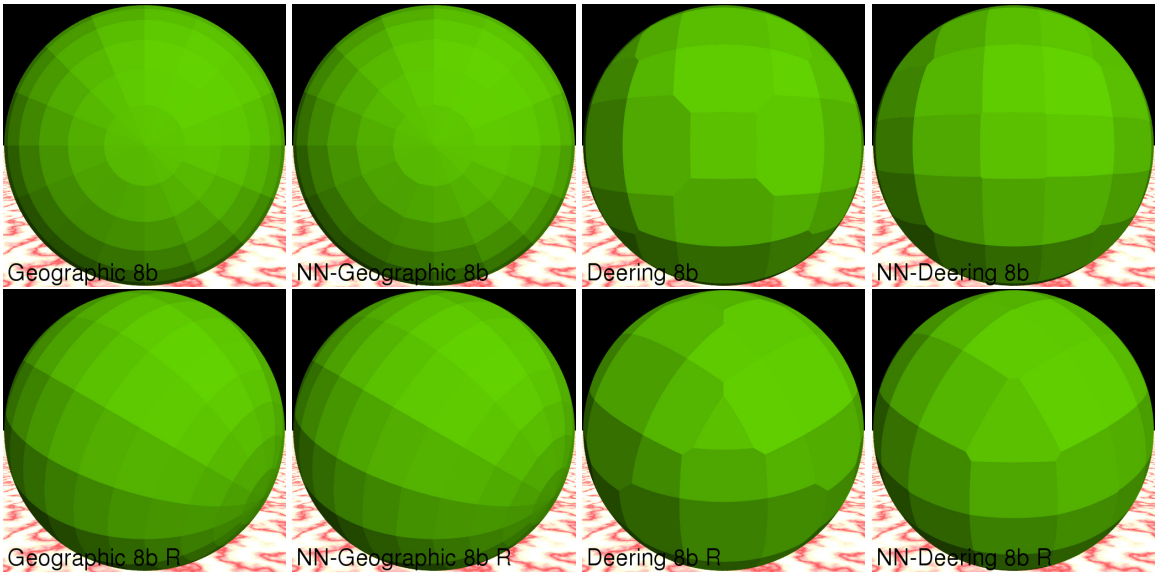


Figure 3.17: Different error statistics of the NN-OQ-Gnomonic-Hex, the NN-OQ-Areal-Hex, the NN-OQ-Buss-Fillmore, the NN-OQ-Tegmark, the NN-Saff-Kuijlaars, and the NN-SCVT quantization methods. (a,b) Maximum Quantization Error. (c,d) Average Quantization Error. (e,f) Root Mean Square Quantization Error.



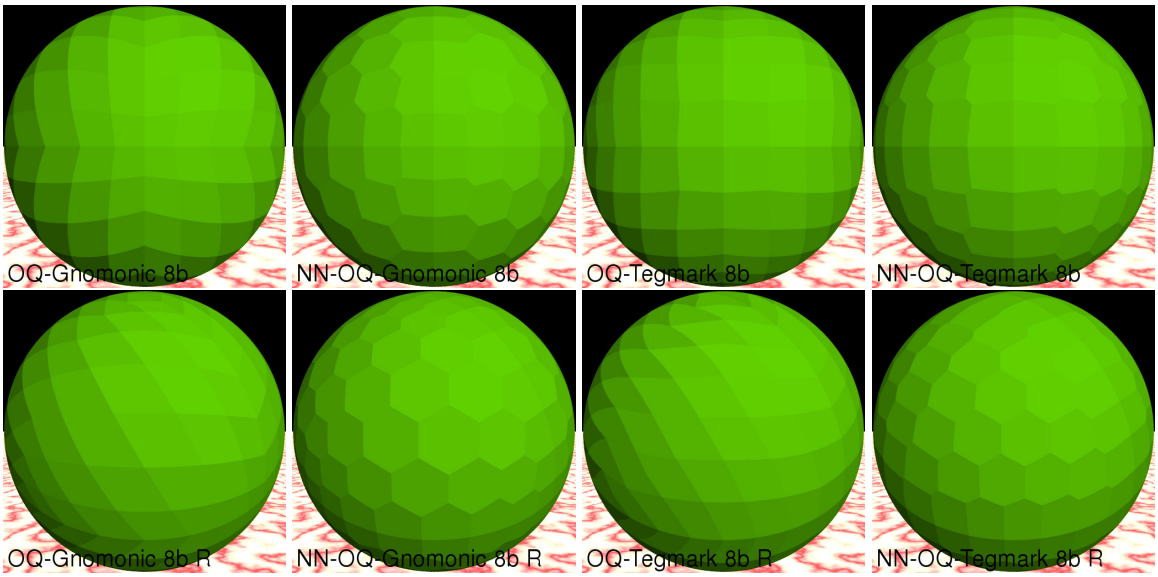
(a) (b)

Figure 3.18: The quantization time of different quantization methods. (a) Encoding time. (b) Decoding time.



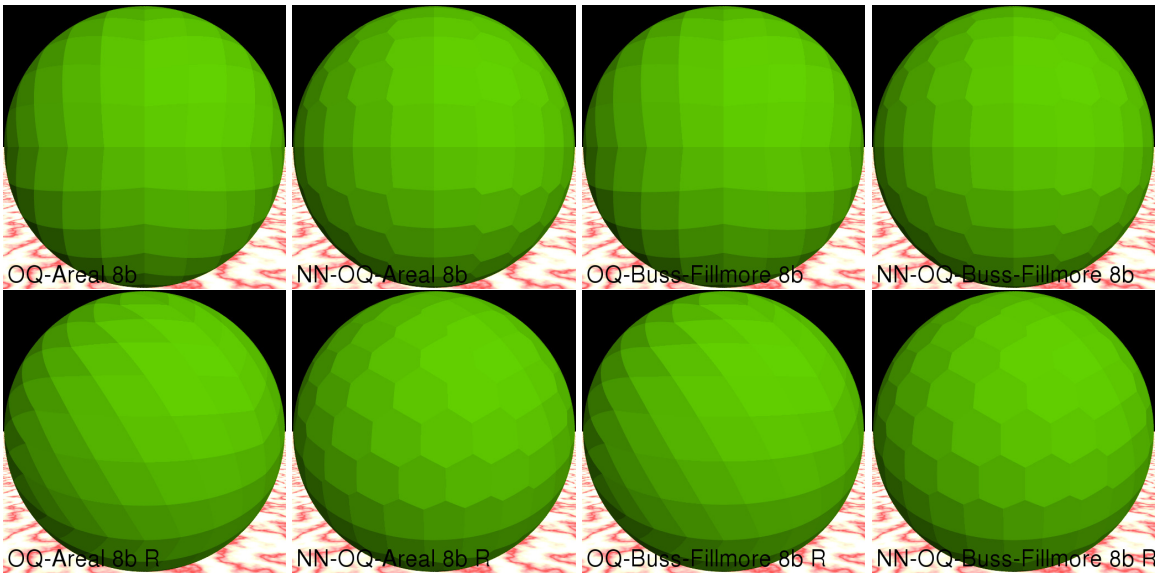
(a) Geographic (b) NN-Geographic (c) Deering (d) NN-Deering

Figure 3.19: Rendering a perfect sphere with normals quantized with 8 bits, using the Geographic, the NN-Geographic, the Deering, and the NN-Deering quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint.



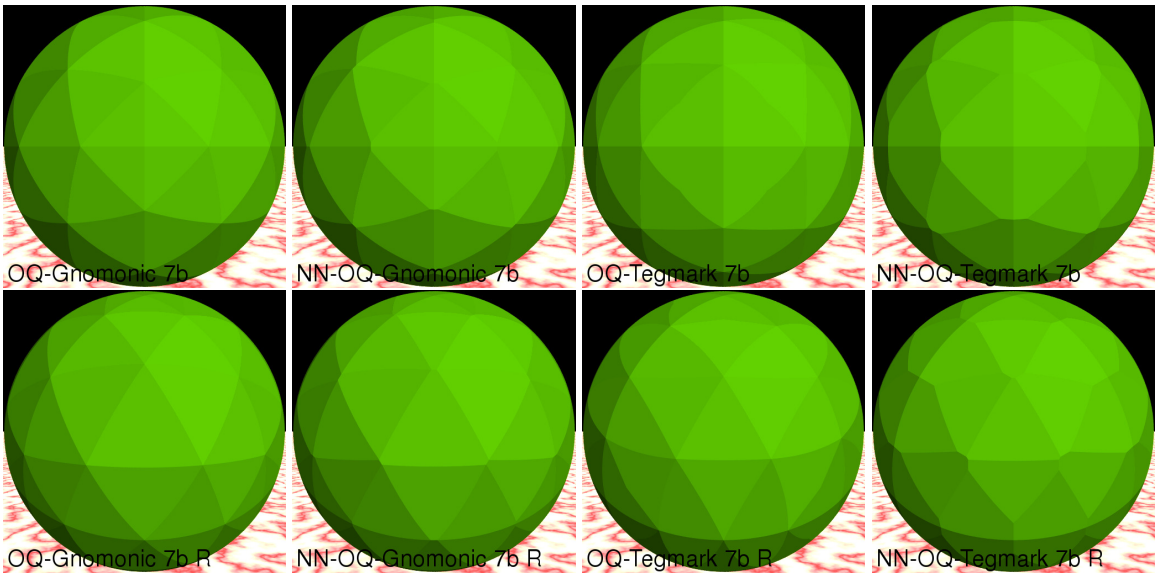
(a) OQ-Gnomonic (b) NN-OQ-Gnomonic (c) OQ-Tegmark (d) NN-OQ-Tegmark

Figure 3.20: Rendering a perfect sphere with normals quantized with 8 bits, using the OQ-Gnomonic, the NN-OQ-Gnomonic, the OQ-Tegmark, and the NN-OQ-Tegmark quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint.



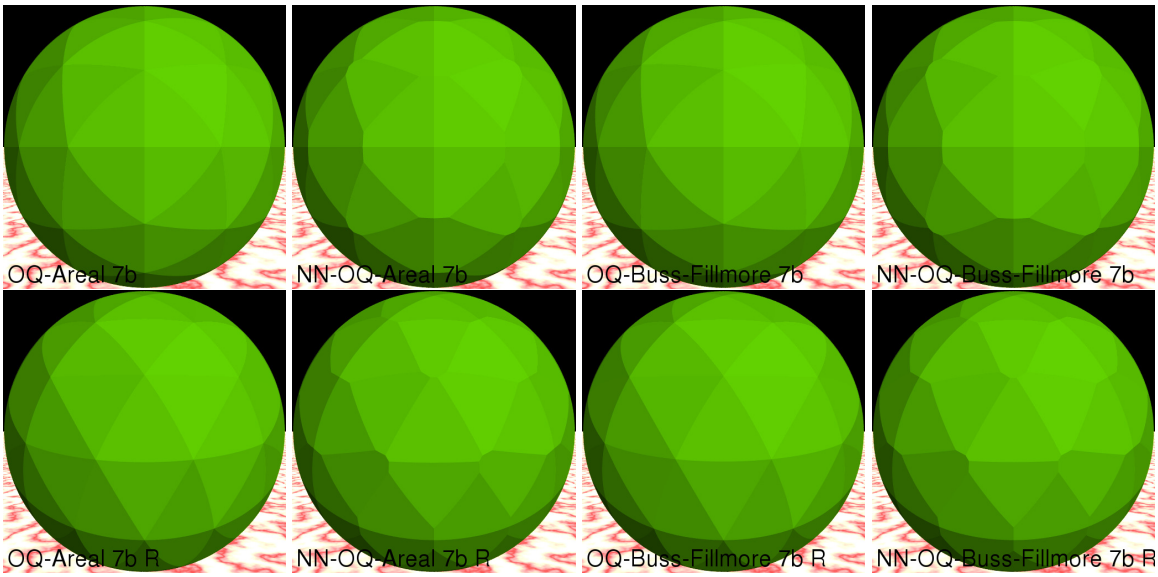
(a) OQ-Areal (b) NN-OQ-Areal (c) OQ-Buss-Fillmore (d) NN-OQ-Buss-Fillmore

Figure 3.21: Rendering a perfect sphere with normals quantized with 8 bits, using the OQ-Areal, the NN-OQ-Areal, the OQ-Buss-Fillmore, and the NN-OQ-Buss-Fillmore quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint.



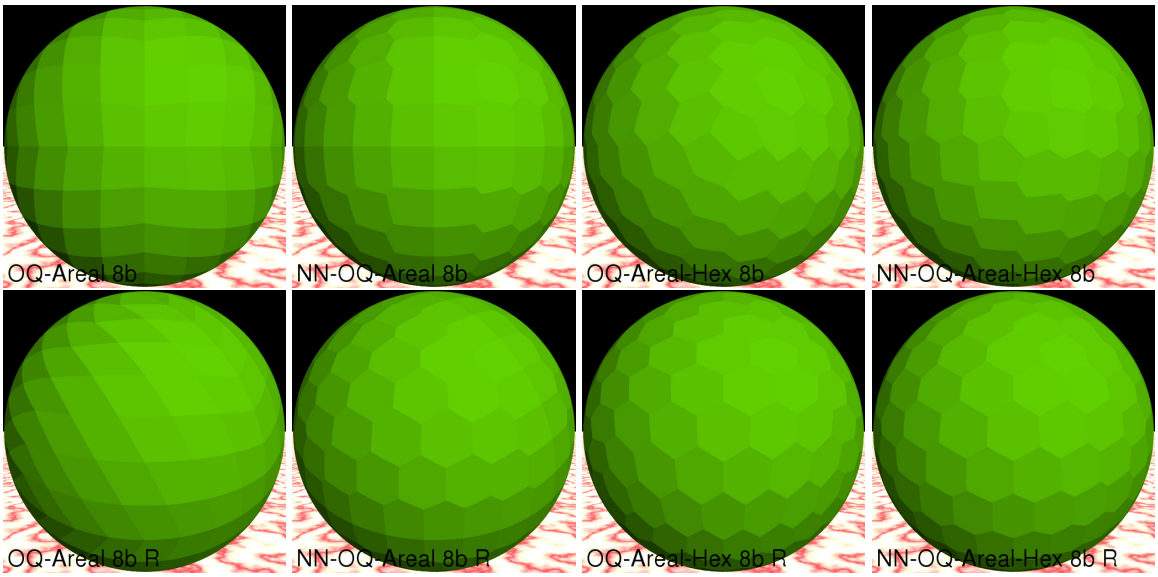
(a) OQ-Gnomonic (b) NN-OQ-Gnomonic (c) OQ-Tegmark (d) NN-OQ-Tegmark

Figure 3.22: Rendering a perfect sphere with normals quantized with 7 bits, using OQ-Gnomonic, NN-OQ-Gnomonic, OQ-Tegmark, and NN-OQ-Tegmark quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint.



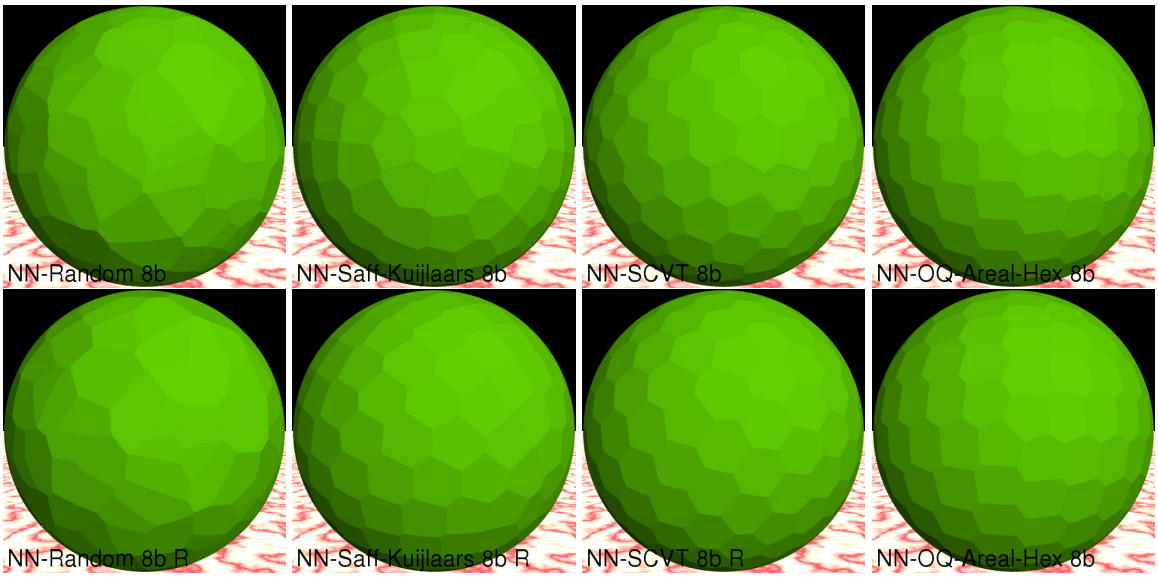
(a) OQ-Areal (b) NN-OQ-Areal (c) OQ-Buss-Fillmore (d) NN-OQ-Buss-Fillmore

Figure 3.23: Rendering a perfect sphere with normals quantized with 7 bits, using the OQ-Areal, the NN-OQ-Areal, the OQ-Buss-Fillmore, and the NN-OQ-Buss-Fillmore quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint.



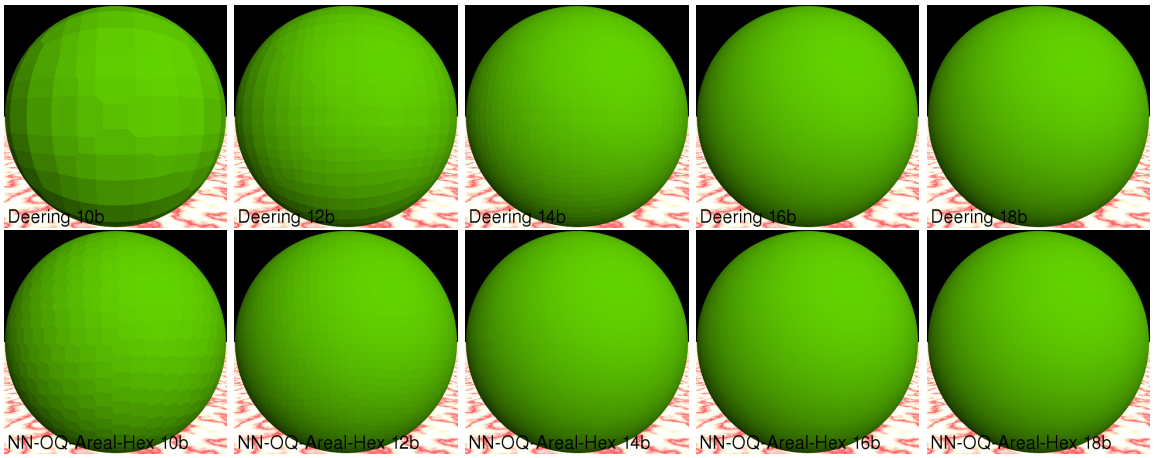
(a) OQ-Areal (b) NN-OQ-Areal (c) OQ-Areal-Hex (d) NN-OQ-Areal-Hex

Figure 3.24: Rendering a perfect sphere with quantized normals at 8 bits, using the OQ-Areal, the NN-OQ-Areal, the OQ-Areal-Hex, and the NN-OQ-Areal-Hex quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint.



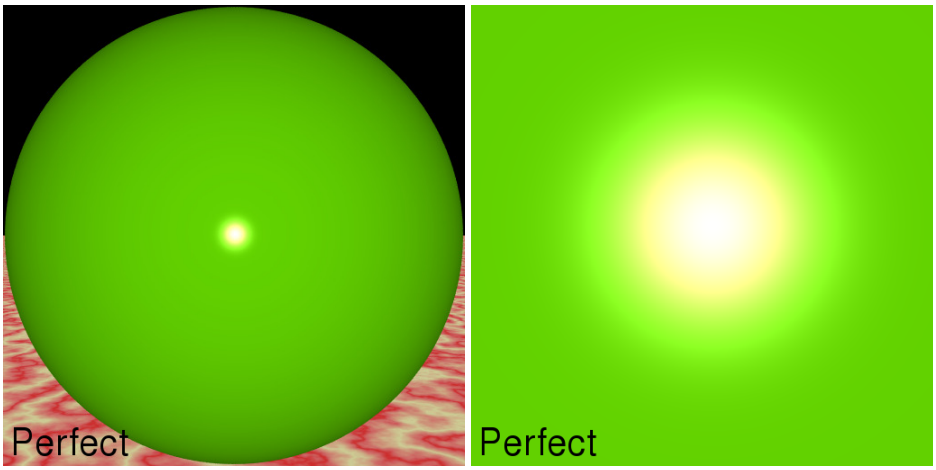
(a) NN-Random (b) NN-Saff-Kuijlaars (c) NN-SCVT (d) NN-OQ-Areal-Hex

Figure 3.25: Rendering a perfect sphere with quantized normals using the NN-Random, the NN-Saff-Kuijlaars, the NN-SCVT, and the NN-OQ-Areal-Hex quantization methods. The spheres in the top row have been rotated in the bottom row in order to show the tessellations from a different viewpoint.



(a) 10 bits (b) 12 bits (c) 14 bits (d) 16 bits (e) 18 bits

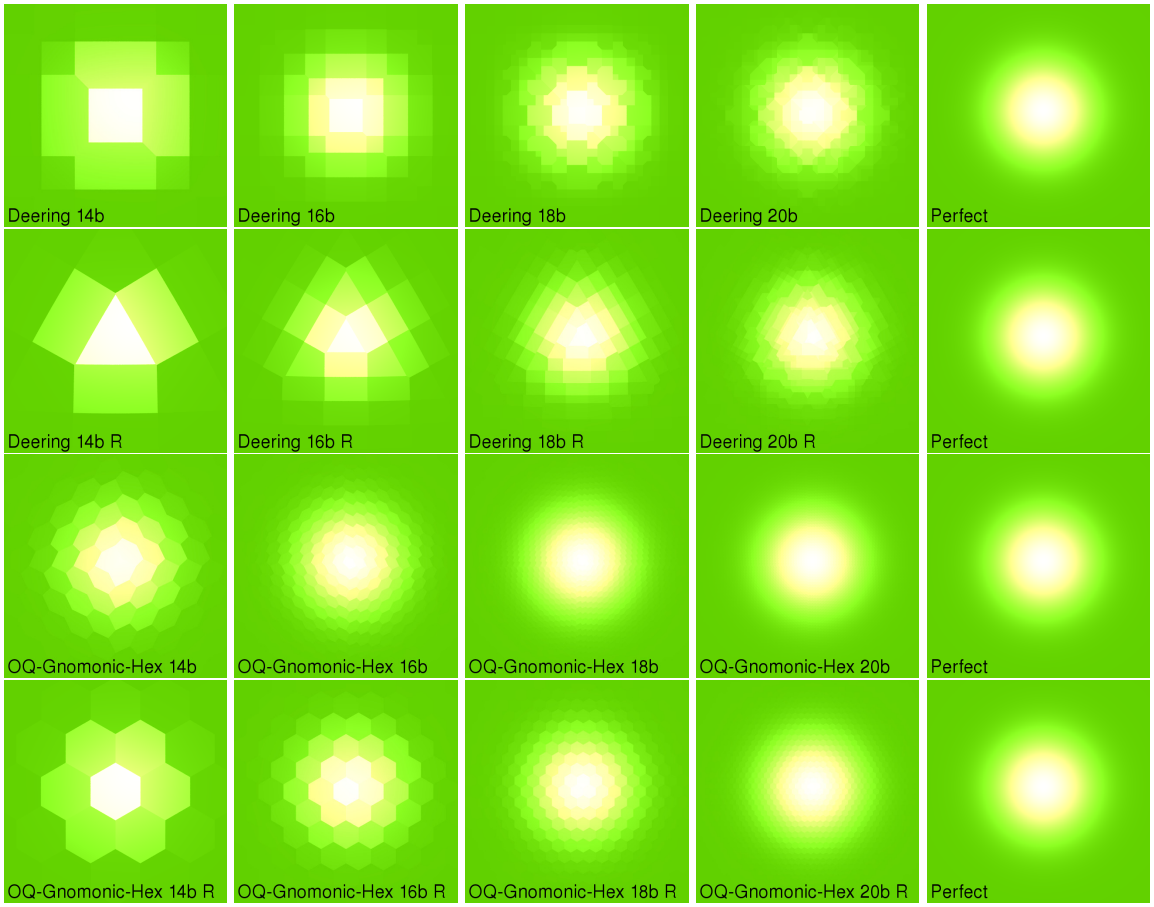
Figure 3.26: Rendering a perfect sphere with normals quantized with the Deering and the NN-OQ-Areal-Hex quantization methods with different bits of quantization. The spheres in the top row are quantized using the Deering method, and the spheres in the bottom row are quantized using the NN-OQ-Areal-Hex.



(a) A sphere rendered with specular lighting.

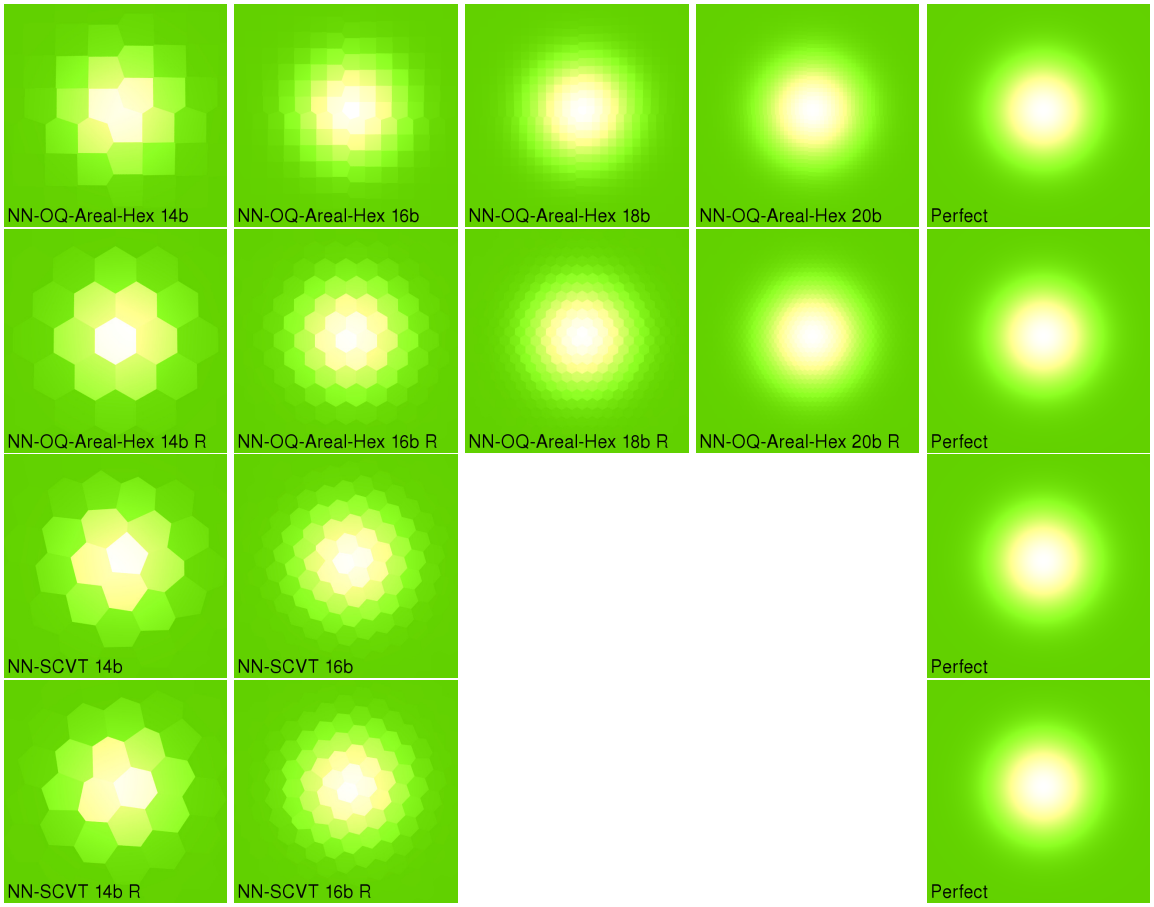
(b) The sphere in (a) magnified around its specular highlight.

Figure 3.27: Rendering of a sphere with specular highlight.



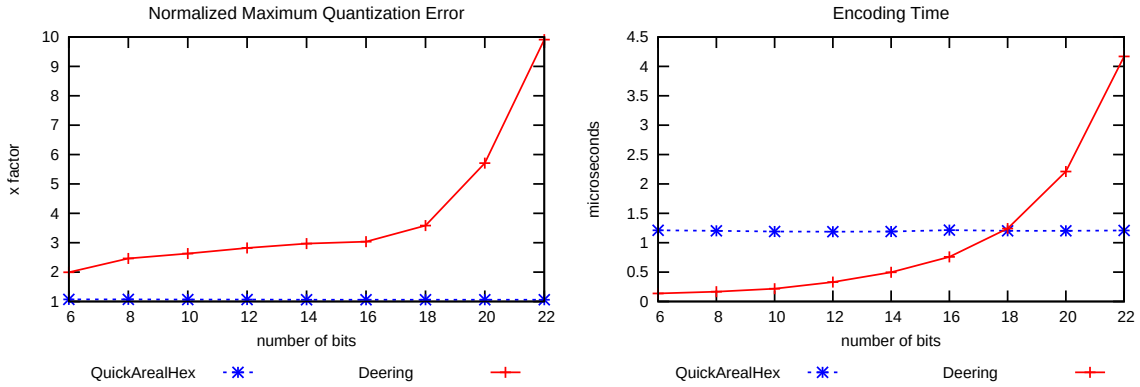
(a) 14 bits (b) 16 bits (c) 18 bits (d) 20 bits (e) no quantization

Figure 3.28: Rendering the sphere in Figure 3.27(a) with surface normals quantized with 14, 16, 18, and 20 bits using the Deering, the OQ-Gnomonic-Hex, the NN-OQ-Areal-Hex, and the NN-SCVT quantization methods.



(a) 14 bits (b) 16 bits (c) 18 bits (d) 20 bits (e) no quantization

Figure 3.28, continued: Rendering the sphere in Figure 3.27(a) with surface normals quantized with 14, 16, 18, and 20 bits using the Deering, the OQ-Gnomonic-Hex, the NN-OQ-Areal-Hex, and the NN-SCVT quantization methods.



(a) (b)

Figure 3.29: A comparison of the Deering and QuickArealHex methods. (a) Normalized Maximum Quantization Error. (b) Encoding time.

Chapter 4

Execution time analysis of a top-down R-tree construction algorithm

4.1 Introduction

R-trees [39] (see also [64] for a review of recent results) were developed as an index structure for the efficient management of multi-dimensional and spatial data such as points and regions, as well as spatial data with a temporal component (e.g., [10, 47, 56, 61]). Common operations performed on an R-tree include point location queries, range queries and nearest neighbor queries. Given a set of input data objects, an R-tree could be constructed by the repeated insertion of each data item. This approach does not take advantage of the fact that all the data items are known beforehand, as in this case it is preferable to insert all of the data items using a single operation. Such an operation is called *bulk loading*. An additional motivation for bulk loading is to enable the construction of an R-tree which can perform queries faster.

There have been a number of bulk loading techniques developed for R-trees (e.g., [1, 11, 13, 45, 49, 58, 77]). In this paper we present a formal analysis of the cost of building an R-tree using the *Top-down Greedy Split* (TGS) bulk loading technique that was origi-

nally proposed by García, López, and Leutenegger [31]. Our approach differs from theirs by providing a detailed implementation which enables a more precise analysis of the algorithm. In particular, the analysis given in [31] only considers the number of disk pages accessed for bulk loading of the data, while a formal analysis of the needed CPU time is missing. Given that memory is getting cheaper, many spatial databases fit into memory (e.g., in-car applications) and thus an analysis of the number of disk page accesses is not sufficient. This is especially true in the case of a bulk loading algorithm such as TGS which performs many sorting operations in order to obtain an R-tree that minimizes a particular cost function. The algorithm of García et al. in [31] uses a classical bottom-up packing approach. We also introduce a top-down packing approach, show how to incorporate it into the TGS algorithm, and discuss the tradeoffs in choosing one versus the other.

Our motivation for presenting the analysis and implementation of the TGS algorithm is to try to provide analytical support to the experimental results reported in [30] which showed that the R-tree built using the TGS bulk loading technique performs much better compared to those built using other bulk loading techniques, even though the bulk loading process is slightly slower for TGS. It is important to note that in this paper we do not analyze the performance of queries executed on an R-tree constructed by the TGS bulk loading operation; instead, we repeat, our contribution is to formally analyze the time required for performing the bulk loading operation.

The rest of this paper is organized as follows. Section 4.2 reviews R-trees and the bulk-loading process. Section 4.3 provides a description of the TGS bulk loading algorithm as well as a sample implementation. Section 4.4 describes the two approaches to packing that are used in bulk loading algorithms. Section 4.5 contains the formal analysis of the TGS algorithm, while Section 4.6 contains some concluding remarks.

4.2 Background

The most basic object that is stored in an R-tree is an *axis-aligned rectangle*, also called a *bounding box*. An R-tree data structure is a height balanced data structure similar to a B-tree [15] which facilitates storage of spatial data in secondary storage. Each leaf node of an R-tree holds two items for each data record. One is the bounding box of the record, and one is a pointer (or an identifier) to the data record itself. Similarly, each nonleaf node of an R-tree holds two items for each of its children: a bounding box of the child, and a pointer to the child. Naturally, the bounding box of a node is the smallest bounding box containing all the bounding boxes of the elements of that node. Furthermore, to ensure that an R-tree is height balanced, each node has between b and $M \geq 2b$ children, where M is called the *page capacity* of an R-tree node. In general, the page capacity of a leaf node is different from the page capacity of a nonleaf node. A node that has less than b children is termed *underpacked*. The root node of an R-tree is allowed to be underpacked.

The relationship between a node and its children is such that the boxes that are associated with the children of a node are all spatially contained in the box that is associated with the node itself. A common query on an R-tree is a *window query*, which given a query rectangle w, reports all the data records in the R-tree whose boxes intersect w. When w is a point, the query is called a *point query*. A window query is performed by examining the root of an R-tree and recursively searching all its children that intersect w.

The efficiency of operations on an R-tree depends on the geometric relation of the nodes with respect to each other as well as the height of the R-tree. For example, during a point query, all the nodes of the R-tree that cover the query point are visited. The performance of the query is thus proportional to the number of nodes visited. If the query point is inside the bounding box of two or more sibling R-tree nodes, then all such nodes must be visited. It is possible to perform queries faster if the sibling nodes of an R-tree have little or no overlap. Intuitively, reducing the overlap of sibling R-tree nodes also results in better performance. A *cost function* quantifies this notion by assigning a cost to the geometric relation of the

sibling nodes of an R-tree. Usually [4, 39, 45], the cost function of two bounding boxes is a function of their areas, perimeters, and their overlap area. For example, consider the collection of five bounding boxes in Figure 4.1(a). Suppose we are storing the five boxes in an R-tree with the parameters $b = 2, M = 4$. We need to partition the five boxes into two groups, one of size two, and one of size three. These partitions corresponds to the children of the root of the R-tree. We show two such partitions in Figures 4.1(b) and 4.1(c). The amount of overlap in Figure 4.1(b) is greater than in Figure 4.1(c), and thus the partition depicted in Figure 4.1(b) is preferable.

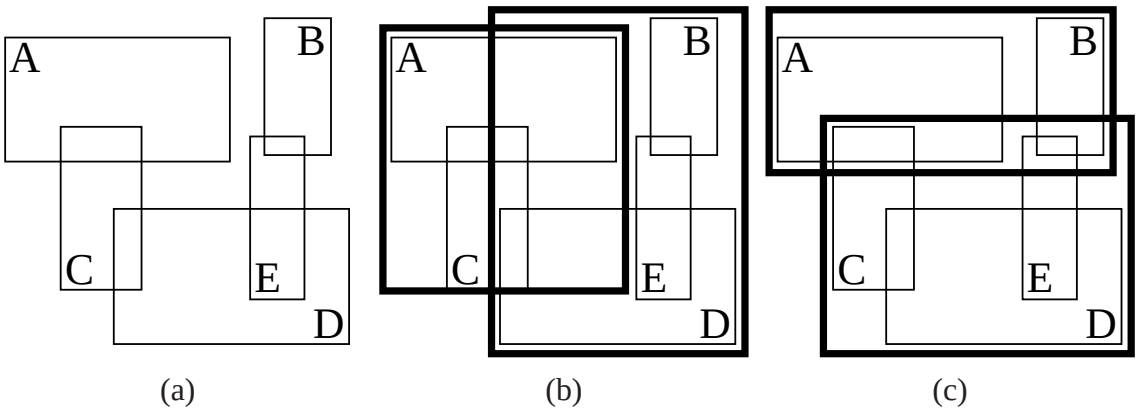


Figure 4.1: Arrangement of bounding boxes. (a) A set of five boxes. (b) One bounding box for boxes (A, C) and one for (B, D, E). (c) One bounding box for boxes (A, B) and one for (C, D, E).

Roussopoulos and Leifker [58] introduced the concept of *packed* R-trees. In a packed R-tree, all nodes of the R-tree, are as full as possible.This results in an R-tree with the lowest possible height, thereby possibly improving the performance of search queries. However, the search performance is still dependent on the amount of overlap between the nodes. Roussopoulos and Leifker's approach for building a packed R-tree is a *bottom-up* approach that builds an R-tree by placing spatially close rectangles together.

In general, a bottom-up approach for building packed R-trees is a two step process. It the first step, the n data rectangles are sorted according to a predetermined sort order. In the second step, groups of M data rectangles are placed into $\lceil \frac{n}{M} \rceil$ leaf nodes. After building the leaf nodes, the same process is applied to the bounding boxes of the leaf nodes to build

another level of the R-tree. This process is applied iteratively until the root node of the R-tree is obtained. Thus the sorting step is performed at each level of the tree although the number of elements that are sorted is successively smaller at the successively shallower levels of the tree. The time complexity of a bottom-up approach is

$$\sum_{i=0}^{\lfloor \log_M n \rfloor} \left\lceil \frac{n}{M^i} \right\rceil \log \left\lceil \frac{n}{M^i} \right\rceil = O(n \log n).$$

On the other hand, a *top-down* approach, builds the higher levels of the R-tree first. The data rectangles are sorted according to a predetermined sort order and then the groups of $\frac{n}{M}$ data rectangles are associated with the M children of the root. The process will be repeated for each of the M children of the root. In such an approach only one sort is needed for the first iteration, as the order of the boxes does not change during the subsequent iterations. The time complexity of a top-down approach is also $O(n \log n)$. Kamel and Faloutsos [45] use a Hilbert curve sort order to build packed R-trees by sorting the collection of data rectangles only once. Hence, their method while described as bottom-up approach is essentially a top-down approach.

The bulk loading approaches described so far do not take into account any notion of a cost function. Depending on the sort order chosen, these approaches may or may not produce a desirable R-tree. The TGS (Top-down Greedy Split) algorithm of García et al. [31] proposes to overcome this issue by taking into account a cost function and tries to find a partition with a low cost. The n data rectangles are first sorted using an appropriate sort key, and then inserted in order into M bins each holding $\frac{n}{M}$ rectangles. Moreover, the minimum bounding box of the rectangles in a bin is computed and kept as the bounding box of the bin. The bins are also numbered from 1 to M using each of the possible sort orders. At this point, we try to find an optimal partition of the M bins into two sets containing the first i bins and the next $M-i$ bins so that the value of the cost function on the minimum bounding rectangles of the bins that make up each of the two sets is minimized (e.g., their overlap).

The key to this step is that we try to find the optimal partition using all of the possible sort orders. It should be clear that in this initial step there are $M-1$ possible partitions and the TGS algorithm takes all of them and all of the possible sort orders into account when determining the optimal one at this step. This is a *greedy binary* split of the bins and the rectangles that they contain into two partitions. Each partition of the data rectangles is split again until all of the data rectangles are partitioned into M partitions of sizes less than or equal to $\frac{n}{M}$, at which time we have obtained the first level of the R-tree. The same algorithm is then applied recursively to the individual nodes of the R-tree until all nodes at a given level contain at most M data rectangles. Given S different sort orders, the TGS algorithm sorts the n rectangles in S different orders. While the S sort orders used in [31] are based on the $2d$ coordinates of the d dimensional rectangles, any sort order defined using a sort key, such as as the Hilbert order, could be used in the TGS algorithm. Section 4.3 contains a more detailed description of the algorithm.

4.3 TGS Bulk Loading Algorithm

In this section, we present a detailed implementation-level description of the TGS algorithm given in [30, 31]. The input to the TGS bulk loading algorithm is a list D of d-dimensional data rectangles. The algorithm builds an R-tree for these data rectangles. Each d dimensional rectangle r is defined by d pairs of scalars, where each pair $r_i = (r_i^-, r_i^+)$ denotes the range that r spans in the i^{th} dimension. We use the notation $p \boxplus q$ to denote the minimum bounding box of two rectangles p and q. For $r = p \boxplus q$, we have $r_i^- = \min(p_i^-, q_i^-)$ and $r_i^+ = \max(p_i^+, q_i^+)$ for $i = 1 \ldots d$.

We assume that there are S different sort keys associated with each rectangle r in D, where $\text{SORTKEY}(r,s)$ denotes the s^{th} sort key on r. For example, the sort keys of a two dimensional rectangle r could be chosen as its extents: $\text{SORTKEY}(r,1) = r_1^-$, $\text{SORTKEY}(r,2) = r_1^+$, $\text{SORTKEY}(r,3) = r_2^-$, and $\text{SORTKEY}(r,4) = r_2^+$. We assume further that each sort

key associated with a rectangle in are uniquely defined, and a mechanism for breaking the ties is in place. That is, for the s^{th} sort key and the distinct data rectangles r and p, $\text{SORTKEY}(r,s) \neq \text{SORTKEY}(p,s)$.

The algorithm is invoked by $\text{BULKLOAD}(D)$ (Algorithm 4.1), where D is the list of input rectangles. BULKLOAD proceeds by sorting the data in ascending order using S different sort keys, and storing the results in lists $D^{(1)},\ldots,D^{(S)}$. It then determines the height of the R-tree and invokes BULKLOADCHUNK, which generates an R-tree with the specified height using the sorted data. Note that the boldface symbol **D** denotes the sorted lists $D^{(1)},\ldots,D^{(S)}$.

Algorithm 4.1 $\text{BULKLOAD}(D)$

Input: $D = \{r_1,\ldots,r_n\}$ is a list of n rectangles.
(* *S is the number of sort keys defined on each rectangle.* *)
(* *N is the capacity of leaf nodes, and M is the capacity of nonleaf nodes.* *)
(* *Top-down-Greedy-Split bulk loading algorithm* *)
for $i = 1$ **to** S **do**
  $D^{(i)} \leftarrow \text{SORT}(D,i)$ (* *Sort D on the i^{th} sort key* *)
end for
$h \leftarrow \max(0, \left\lceil \log_M \frac{|D|}{N} \right\rceil)$ (* *Desired height of the R-tree.* *)
return $\text{BULKLOADCHUNK}(\mathbf{D},h)$

BULKLOADCHUNK (Algorithm 4.2) simply returns an R-tree leaf if the desired height of the R-tree is zero. Otherwise, it determines m, the desired number of data items that need to be placed under each node (Line 6). m is chosen so that all the nodes will have the maximum number of data rectangles under them. Next, it uses the PARTITION algorithm (Algorithm 4.3) to partition the data into sets of size m, and recursively builds an R-tree node for each partition, returning a nonleaf R-tree node as their parent.

The PARTITION algorithm partitions the input set **D** into partitions of size m using a greedy paradigm. It uses the BESTBINARYSPLIT algorithm (Algorithm 4.4) to find a desirable binary split of the input set **D** into two partitions **L** and **H**. Note again as in the case of **D**, that the boldface symbols **L** and **H** denote the sorted lists $L^{(1)},\ldots,L^{(S)}$ and $H^{(1)},\ldots,H^{(S)}$, respectively. It then recursively partitions **L** and **H** and builds a bigger

Algorithm 4.2 BULKLOADCHUNK$(\mathbf{D}, h)$

```
(* Bulk load data in D into an R-tree of height h. *)
(* M is the capacity of nonleaf nodes. *)
if h = 0 then
  return BUILDLEAFNODE(D^(1)) (* Note that any of the sorted lists could have been used. *)
else
  m ← N · M^(h−1) (* Desired number of data items under each child of this node. *)
  {D_1, ..., D_k} ← PARTITION(D, m) (* Partition of D into k ≤ M parts. *)
  for i = 1 to k do
    n_i ← BULKLOADCHUNK(D_i, h − 1) (* Recursively bulk load lower levels of the R-tree. *)
  end for
  return BUILDNONLEAFNODE(n_1, ..., n_k)
end if
```

partition by joining them.

Algorithm 4.3 PARTITION$(\mathbf{D}, m)$

```
(* Partition data into ⌈|D^(1)| / m⌉ parts of size m ≠ 0. *)
if |D^(1)| ≤ m then
  return D (* one partition *)
end if
L, H ← BESTBINARYSPLIT(D, m)
return Concatenation of PARTITION(L, m) and PARTITION(H, m)
```

The BESTBINARYSPLIT algorithm considers the S different orderings of the input set $\mathbf{D}$. It uses each ordering to group the data rectangles into groups of size m. That is, if there are $M \cdot m$ rectangles, then the first m rectangles are grouped together, then the second m rectangles are grouped together, and so forth. It then considers all possible splits of the groups into two parts. In particular, if there are M groups, it considers $S \cdot (M-1)$ possible ways of splitting the groups into two parts. The BESTBINARYSPLIT algorithm chooses the split with the lowest cost, and accordingly splits the input set $\mathbf{D}$ (i.e., the data and its S orderings) into two parts using the SPLITONKEY algorithm (Algorithm 4.6).

The COMPUTEBOUNDINGBOXES algorithm (Algorithm 4.5) determines the bounding boxes that are needed for determining the cost of each binary split considered in

Algorithm 4.4 BESTBINARYSPLIT$(\mathbf{D}, m)$

```
    (* Find the best binary split of D. *)
    (* m ≠ 0 is the size of each partition. *)
    M ← ⌈|D^(1)| / m⌉  (* Number of partitions *)
    c* ← ∞  (* Best cost found so far *)
 5: for s = 1 to S do
      F, B ← COMPUTEBOUNDINGBOXES(D^(s), m)
      for i = 1 to M − 1 do
        c ← cost(F_i, B_i)
        if c < c* then
10:       c* ← c  (* Best cost *)
          s* ← s  (* Best sort order *)
          key ← SORTKEY(D^(s)_{i·m}, s)  (* Sort key of split position *)
        end if
      end for
15: end for
    for s = 1 to S do
      L^(s), H^(s) ← SPLITONKEY(D^(s), s*, key)
    end for
    return L, H
```

BESTBINARYSPLIT. It first computes B, the bounding boxes for each group of m rectangles. It then computes lower bounding boxes (L) and the higher bounding boxes (H).

The SPLITONKEY algorithm will split a sorted list D, into two sorted lists L and H based on a threshold t, and the s^{th} sort key among the S sort keys defined on rectangles. At the end of SPLITONKEY, L will hold all elements of I such that their s^{th} key is less than t, and H will hold the rest.

4.4 Bottom-up Packing Versus Top-Down Packing Algorithms

Figures 4.2 shows a set of 30 randomly generated rectangles that are bulk loaded using the TGS algorithm into an R-tree with page capacity of 8 (i.e., $N = M = 8$). Each R-tree in the figure consists of a root and four leaf nodes under the root. The inner rectangles

Algorithm 4.5 COMPUTEBOUNDINGBOXES(D,m)

Output: $L_i = D_1 \boxplus \cdots \boxplus D_{i\cdot m}$ for $1 \leq i < M$.
Output: $H_i = D_{i\cdot m+1} \boxplus \cdots \boxplus D_n$ for $1 \leq i < M, n = |D|$.
(* *Compute the lower and higher bounding boxes of of possible binary splits of D list of n rectangles into groups of size m* *)
(* $m \neq 0$ *is the size of each group.* *)
$M \leftarrow \left\lceil \frac{|D|}{m} \right\rceil$ (* *Number of groups* *)
B is a list of M rectangles.
L, H are each a list of $M-1$ rectangles.
for $i = 1$ **to** M **do**
  $B_i \leftarrow D_{(i-1)\cdot m+1} \boxplus D_{(i-1)\cdot m+2} \boxplus \cdots \boxplus D_{\min(|D|,i\cdot m)}$
end for
$L_1 \leftarrow B_1$
$H_{M-1} \leftarrow B_M$
for $i = 2$ **to** $M-1$ **do**
  $L_i \leftarrow L_{i-1} \boxplus B_i$
  $H_{M-i} \leftarrow B_{M-i+1} \boxplus H_{M-i+1}$
end for
return L, H

correspond to the data rectangles, and the outer rectangles correspond to the bounding boxes of each leaf node. Four sort keys are used in the generation of the figure. In particular, the four sort keys of a rectangle are its two extents in each of the two dimensions. The cost functions used to generate Figure 4.2 involved minimizing the overlap area of two rectangles (Figure 4.2(a)) and minimizing the total area of two rectangles (Figure 4.2(b)).

Traditionally, packing methods work by filling the leaf nodes as much as possible and then proceed to apply the same filling criteria to the nonleaf nodes. We characterize such

Algorithm 4.6 SPLITONKEY(D,s,t)

L and H are two empty lists.
for all r **in** D **do**
  if SORTKEY$(r,s) < t$ **then**
    append r to the end of list L.
  else
    append r to the end of list H.
  end if
end for
return L, H

an approach as *bottom-up packing* and is the one used in the implementation of the TGS algorithm described in Section 4.3 and illustrated in Figure 4.2. We could also proceed by starting at the root and packing the nonleaf nodes as much as possible. Such an approach can be characterized as *top-down packing*. The TGS algorithm whose implementation we described could also be converted to use top-down packing by modifying line 6 of BULKLOADCHUNK (Algorithm 4.2) to be:

$m \leftarrow \left\lceil \frac{|D^{(1)}|}{M} \right\rceil$ (* *Desired number of data items under each child of this node.* *)

Figure 4.3 was obtained using this modification. They use the same dataset as in Figure 4.2, and again the trees are differentiated on the basis of the cost function that is minimized. Notice that the number of leaf nodes in this example is eight, and that four rectangles are placed in each leaf node with the exception of one leaf node which has just two rectangles. Given N and M as the capacities of the leaf and nonleaf nodes, respectively, the bottom-up packing and top-down packing yield identical results when n, the total number of data objects, equals $N \cdot M^h$ for some integer value $h > 0$. However, when n is not equal to $N \cdot M^h$, the top-down packing yields a different result as can be seen by comparing Figures 4.2(a) and 4.2(b) with Figures 4.3(a) and 4.3(b). It is interesting to observe that top-down packing can potentially allow the queries to be performed faster as there are more children under each nonleaf node thereby permitting more effective pruning when answering queries. Notice also that regardless of whether top-down or bottom-up packing is used with the TGS algorithm, the resulting R-trees have the same height. However, an R-tree constructed with the bottom-up packing TGS algorithm has fewer nodes than an R-tree constructed with the top-down TGS algorithm. Therefore, we can can identify a tradeoff between the two packing approaches. In particular, the top-down packing TGS algorithm builds R-trees that can potentially be used to answer queries faster than R-trees built by the bottom-up packing TGS algorithm at the expense of requiring more storage space. We point out that the relative merit of the two packing strategies depends on the query model. For example, to answer the window query Q shown in Figure 4.2(a), one leaf node needs to

be read from disk. However, to answer the same query shown in Figure 4.3(a), only the root node needs to be examined. On the other hand, to answer a window query that intersects all the leaf nodes, the top-down approach is faster than the bottom-up approach. Moreover, future insert operations to an R-tree built by the bottom-up packing TGS algorithm are more likely to increase the height of the R-tree than insert operations to an R-tree built by the top-down packing TGS algorithm. Finally, we observe that one of the consequences of using top-down packing is that some of the leaf nodes of the R-tree may be underpacked (e.g., recall that one leaf node in Figure 4.3(a) has just two data rectangles.) Of course, when using bottom-up packing at most one node at each level is underpacked. However, this does not affect the correctness of the results returned by queries on these R-trees.

Figure 4.2: Result of applying the TGS bottom-up packing bulk loading algorithm to bulk load a packed R-tree using a cost function that minimizes (a) the overlap area, and (b) the total area.

4.5 Analysis

In this section we analyze the running time of the TGS bulk loading algorithm. As we pointed out in Section 4.1, the analysis provided in [31] was only in terms of the number

(a) (b)

Figure 4.3: Result of applying the top-down packing TGS bulk loading algorithm to bulk load an R-tree using a cost function that minimizes (a) the overlap area, and (b) the total area.

of disk page accesses, whereas here we focus on the CPU time in light of the repeated invocation of the sorting steps by the algorithm in the process of minimizing the particular cost function. To simplify the analysis, we assume that the number of input data rectangles results in a fully packed R-tree, i.e., there are $n = NM^h$ data rectangles, where h denotes the height of the resulting R-tree.

Let $T(n)$ denote the time complexity of the BULKLOAD algorithm. The BULKLOAD algorithm performs S sorts. We have,

$$T(n) = O(Sn\log n) + B(n,h) \tag{4.1}$$

where $B(n,h)$ denotes the time complexity of the BULKLOADCHUNK algorithm.

Notice that as the initial number of the data rectangles results in a fully packed R-tree,

it suffices to derive $B(N \cdot M^h, h)$. Letting $C(h)$ denote $B(N \cdot M^h, h)$. We have,

$$C(h) = \begin{cases} O(N) & h = 0 \\ P(N \cdot M^h, N \cdot M^{h-1}) + M \cdot C(h-1) + O(M) & h > 0 \end{cases} \tag{4.2}$$

where $P(n,m)$ denotes the time complexity of the PARTITION algorithm, and $O(M)$ corresponds to the cost of invoking BUILDNONLEAFNODE.

We now proceed to derive $P(n,m)$. We first notice that the PARTITION algorithm consists of a call to the BESTBINARYSPLIT algorithm and two recursive calls to itself. The worst-case scenario arises when each call to BESTBINARYSPLIT results in a minimum partition. That is, BESTBINARYSPLIT(D,m) yields two sets, such that one of them is of size m. We have the following recurrence relation:

$$P(n,m) = \begin{cases} O(1) & n \leq m \\ E(n,m) + P(m,m) + P(n-m,m) & n > m, \end{cases} \tag{4.3}$$

where $E(n,m)$ denotes the time complexity of the BESTBINARYSPLIT algorithm.

Observe that the execution times of the COMPUTEBOUNDINGBOXES algorithm and the SPLITONKEY algorithm are linear in their input size. Moreover, as the BESTBINARYSPLIT algorithm invokes the COMPUTEBOUNDINGBOXES algorithm S times, its execution time, $E(n,m)$, is $O(S \cdot n)$, where n is the number of input rectangles.

Therefore, we can rewrite equation 4.3 as:

$$P(n,m) = \begin{cases} O(1) & n \leq m \\ O(S \cdot n) + P(m,m) + P(n-m,m) & n > m \end{cases} \tag{4.4}$$

To further simplify the analysis, we assume that $n = L \cdot m$, where L is the number of groups that the PARTITION algorithm considers. Notice that for each initial call of PARTITION from BULKLOADCHUNK, we have $L = M$. Therefore, we can rewrite equa-

tion 4.4 in closed form:

$$P(L \cdot m, m) = \sum_{i=2}^{L} O(S \cdot i \cdot m) = O(S \cdot L^2 \cdot m) \tag{4.5}$$

By substituting $N \cdot M^{h-1}$ for m and M for L in equation 4.5 we get $P(N \cdot M^h, N \cdot M^{h-1}) = O(S \cdot (M^2)M^{h-1}) = O(S \cdot M^{h+1})$, and we can rewrite equation 4.2 as:

$$C(h) = \begin{cases} O(N) & h = 0 \\ O(S \cdot M^{h+1}) + M \cdot C(h-1) + O(M) & h > 0. \end{cases}$$

The recurrence relation for $C(h)$ can be solved to yield

$$C(h) = O(M^h \cdot (S \cdot h \cdot M + N)) = O(n \cdot (S \cdot h \cdot \frac{M}{N} + 1)),$$

where we have used $n = N \cdot M^h$.

Recalling that $C(h) = B(NM^h, h)$ and that $h = \log_M \frac{n}{N}$, we obtain from equation 4.1 that

$$T(n) = O(Sn \log n + n \cdot (S \frac{M}{N} \log_M \frac{n}{N})).$$

In particular, for $M = N = O(1)$, we have $T(n) = O(Sn \log n)$, which demonstrates that the observed improved performance of the TGS algorithm by García et al. [31] comes at a cost of a factor of S over that resulting from the use of a bottom up bulk loading approach. Given that S is relatively low for low dimensional data, the improvement seems worth the extra effort. However, in the case of high dimensional data, this may not be the case.

4.6 Concluding Remarks

We have provided a formal analysis of the TGS R-tree bulk loading algorithm of García et al. [31]. Our approach differs from theirs by providing a detailed implementation which

enabled a more precise analysis of the algorithm. In particular, we focused on the CPU time requirements rather than the number of disk page accesses, which is what was done in [31]. We also discussed the tradeoffs of using a classical bottom-up packing approach and a top-down packing approach, and showed how to incorporate the top-down packing approach in the TGS algorithm.

Chapter 5

BV-trees, axis aligned rectangles, and binary space partitioning

5.1 Introduction

The BV-tree [26, 27] is an abstract spatial indexing technique that is based on decoupling (e.g., [63]) the partitioning and grouping processes that form the basis of most spatial indexing methods that use tree directories of buckets. In the case of the BV-tree, the decoupling is designed to overcome the following drawbacks of traditional solutions:

1. Multiple postings in disjoint space decomposition methods that lead to balanced trees such as the hB-tree [22, 50] where a node split in the event of node overflow may be such that one of the children of the node that was split becomes a child of both of the nodes resulting from the split.

2. Multiple coverage and non-disjointness of methods based on object hierarchies such as the R-tree [39] which lead to non-unique search paths.

Note that the principle of decoupling the grouping and partitioning processes has also been used in the PK-tree [78, 83] although the motivation was different (i.e., to overcome the

presence of directory nodes with similarly-shaped hyper-rectangle bounding boxes that have very minimal occupancy in disjoint space decomposition methods such as those based on quadtrees (e.g., [24]) and k-d trees [5] that make use of regular decomposition). Spatial indexes are useful in applications in spatial databases (e.g., [57]) as well as spatio-temporal databases (e.g., [56]).

The BV-tree improves on its predecessor, the BANG file [25, 28], by introducing the concept of guards which guarantee that the height of the BV-tree is always logarithmic in the number of input data points. In addition, the execution time of point insertions and point queries are also guaranteed to be logarithmic. The BANG file employs a regular binary space decomposition that — similar to the k-d tree [5] — cycles the splitting hyperplanes through the axes at each level of the decomposition. In contrast to the BANG file, the BV-tree imposes no restriction on the space decomposition scheme. Instead, the space decomposition scheme is replaced by regions which may have arbitrary shapes. We term the original description of the BV-tree in [26] as an *abstract* BV-tree. However, in this paper we show that the BV-tree can only be implemented when the shape of the regions are precisely defined. We use the term *concrete* BV-tree to refer to the BV-tree such that the shapes of the regions are precisely defined. Moreover, we show that only a binary space partitioning scheme would guarantee the satisfaction of the design assumptions of the BV-tree. This implies that the concrete BV-tree could be decoupled from the binary space partitioning scheme, and is suitable for handling non-spatial data such as metric and non-metric data, as long as a suitable binary partitioning scheme can be defined for the underlying data domain.

The rest of this paper is organized as follows. Section 5.2 provides a brief description of the BV-tree data structure and the implicit assumptions used in the design of the BV-tree; Section 5.3 describes the issues arising when using axis-aligned rectangles for the BV-tree; and Section 5.4 shows that the BV-tree is only applicable to binary space partitioning schemes.

5.2 Description of the BV-tree data structure

The BV-tree is a height-balanced data structure similar to the B-tree [15], that facilitates storage of spatial data in secondary storage. The data objects or pointers to them are stored at the lowest level of the BV-tree, called the *leaf nodes*. In our treatment of the BV-tree, we uniquely identify each node of a BV-tree with two attributes, a *level* and a *region*. The leaf nodes are at level zero, and level k nodes are aggregates of level $k-1$ nodes. The region of a node refers to a subset of the domain space that it spans. Two regions are said to be *cordial* in our terminology, if and only if, they are either disjoint or one is contained in the other one, *i.e.*, no two cordial regions partially overlap each other. For a more complete description of the BV-tree, we refer the interested reader to [26, 27, 63].

The following assumptions – implicit in [26] – describe the regions of a BV-tree:

- **Representation of Regions**: Regions are represented using a common scheme, such as axis-aligned rectangles, Morton blocks [32], *etc*.
- **Cordiality**: The regions of any two nodes in a BV-tree are either disjoint or one is completely contained in the other one.
- **Constant Splits**: Given n cordial regions, it is possible to find a region that is cordial to all the given regions and that contains from $\frac{n}{3}$ to $\frac{2n}{3}$ of the regions.

In the following sections we show the implications of these assumptions on the space decomposition scheme.

5.3 BV-trees and axis-aligned rectangles

The abstract BV-tree does not specify the shape of regions, nor how to represent the regions. We claim that not every region representation scheme is suitable, thereby requiring a more precise definition. In particular, using an example, we show that it is not possible

to represent the regions of a BV-tree with arbitrary axis-aligned rectangles. Axis-aligned rectangles are especially interesting because they are used as the basis for aggregation of objects in in many spatial data structures such as the R-tree [39] and its variants [4, 68].

For example, Figure 5.1 that shows a given set of 24 axis-aligned rectangles that are drawn in solid lines. Consider an additional axis-aligned rectangle placed in the same figure, such as the one shown by dotted lines. It is evident from the figure that this additional rectangle either (i) partially intersects one of the given rectangles, or (ii) contains only one of the given rectangles, or (iii) contains all the given rectangles. Hence we have shown that it is not possible to find a cordial axis-aligned rectangle that contains at least two of the given rectangles, but not all of the given rectangles. Therefore, the *Constant Split* assumption cannot be satisfied in this case. In other words, we have just shown that representing regions of a BV-tree with arbitrary axis-aligned rectangles is not possible because it is not possible to always satisfy the BV-tree design assumptions.

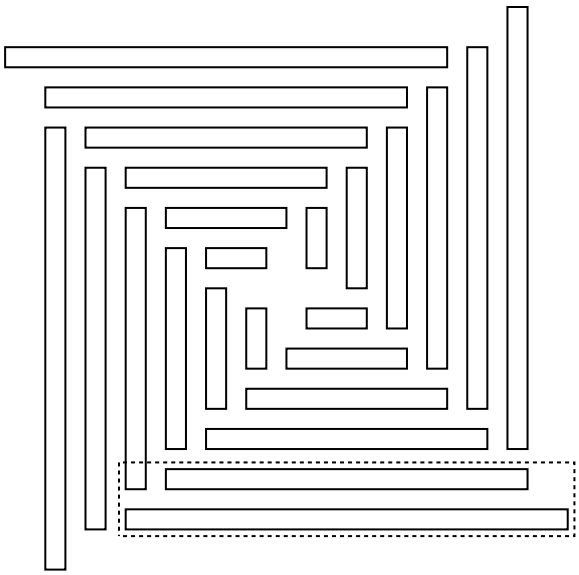


Figure 5.1: A pathological example of axis-aligned rectangles that leads to violation of the BV-tree design assumptions.

The above example assumes a data space of two or more dimensions. We now give an example that shows that even for the one-dimensional case, it is not possible to represent the regions of the BV-tree with arbitrary intervals (*i.e.*, one-dimensional axis-aligned rect-

angles). Consider two nodes of the BV-tree which span the same region, but are at different levels. The BV-tree should be able to split each of these nodes into regions that are cordial with all other regions in the BV-tree. However, this may require information about the regions of the other nodes, and unless other restrictions are in place, the splitting of a node may create a region that is not cordial to the region of another node.

Figure 5.2 is an example of such a case. Figures 5.2(a)-(h) show the successive insertion of 12 one-dimensional data points into a BV-tree with page capacity of three. The regions of the BV-tree are labeled with capital boldface letters. Notice that the node **A0** is promoted in Figure 5.2(f) as it is a guard[1] of the node **E1**. Figure 5.2(g) shows the BV-tree before the insertion of point l, but after the insertion of point k. Insertion of point l requires that the leaf node **A0** containing data points g, i, k, and l to be split. A possible split of **A0** may result in the leaf node **F0** containing data points i and k. It is evident from Figure 5.2(h) that the interval **F** is not cordial to the interval **E**. Observe that there would be no problem if the split of node **A0** would result in leaf node **F0** containing the new data point l.

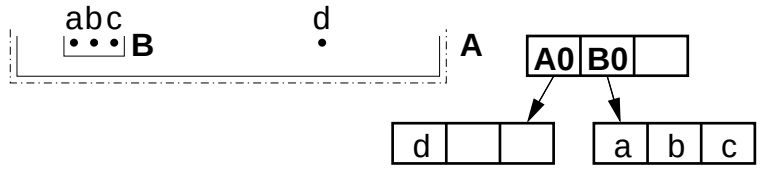
The above example showed that even for one-dimensional data, we cannot use arbitrary intervals for the representation of the BV-tree regions. An insight that we gain from this example is that the local information about the children of a node may not be sufficient for the proper splitting of the node.

5.4 Cordial regions and binary space partitioning

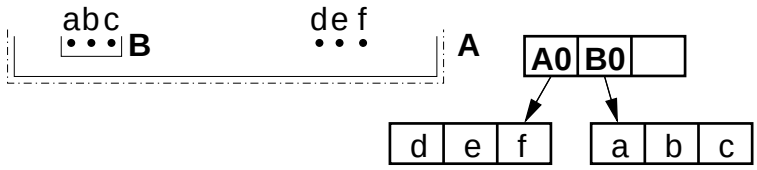
We first start by introducing a few definitions. We use the symbol $\sqsubseteq$ to show the containment relationship between regions, that is $R_1 \sqsubseteq R_2$, if and only if, the region R_2 contains the region R_1. Similarly, $R_1 \sqsubset R_2$, if and only if, R_2 properly contains R_1. Notice that the containment relationship imposes a partial order on regions.

[1] A node *a* is a *guard* of another node *b* when the region of *a* contains the region of *b* and the level of *a* is deeper than the level of *b*. Guards serve to ensure that the search paths for a point query are unique thereby overcoming drawback of spatial indexes based on a non-disjoint decomposition of the underlying space as is the case for object hierarchies such as the R-tree.

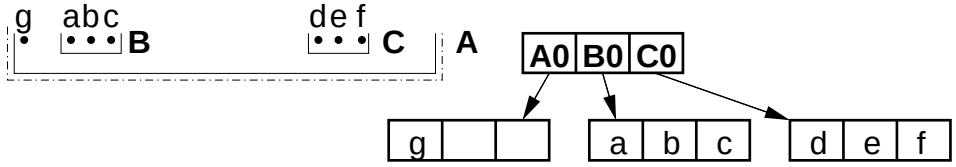
(a) After insertion of data points a, b, c.



(b) After insertion of data point d.



(c) After insertion of data points e, f.



(d) After insertion of data point g.

Figure 5.2: Example of a BV-tree with intervals as regions. The BV-tree, shown on the right, has a page capacity of three. Data points and the regions are shown on the left. The regions corresponding to level 0 nodes, level 1 nodes, and level 2 nodes are drawn in solid lines, dash-dot lines, and dash-dot-dot lines, respectively.

A *binary space partitioning scheme* hierarchically partitions a space *S* into two subsets. Each subset of the space defined by a binary space partitioning scheme can be described using a binary string. Note that in such a representation, the empty string denotes the unpartitioned space *S*. Moreover, if a binary string *a* is a prefix of a binary string *b*, then the region corresponding to *a* contains the region corresponding to *b*.

In this section, we show that the design assumptions of the BV-tree result in a binary space partitioning scheme. This is in contrast to the impression created in the original presentation of the BV-tree that any arbitrary space partitioning scheme can be used. Further,

we claim that any binary space partitioning scheme is suitable for implementing a concrete BV-tree. In other words, each region in a BV-tree corresponds to a subset of the space resulting from the binary space partitioning scheme. Therefore, any such region can be represented using a binary string.

For a concrete BV-tree and a region S, let $\mathscr{R}_S$ denote the set of all possible regions that can result from splitting a node whose region is S using rules that satisfy the BV-tree design assumptions given in Section 5.2. We also define $\mathscr{R}_S^0$,

$$\mathscr{R}_S^0 = \{r \in \mathscr{R}_S : \neg\exists v \in \mathscr{R}_S : r \sqsubset v \sqsubset S\}.$$

That is, $\mathscr{R}_S^0$ is the set of regions such that no other region in $\mathscr{R}_S$ contains them. Notice that $\mathscr{R}_S^0$ covers S, that is each element in S is in at least one member of $\mathscr{R}_S^0$. For example, let $\mathscr{R}_S$ consist of the set of rectangles **A**–**L** as in Figure 5.3(a), not necessarily resulting from a BV-tree decomposition. The rectangles **B**, **G**, **K**, and **I** are contained by the rectangle **E**. The rectangles **L** and **J** are contained by the rectangle **C**, the rectangle **G** and **H** is contained by the rectangle **A**, the rectangle **H** is contained by the rectangle **F**, while the rectangles **A**, **C**, **D**, **E**, and **F** are not contained by any other rectangle. Hence, we have $\mathscr{R}_S^0 = \{\mathbf{A}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}\}$ for this example.

Lemma 5.1 *For a given region S of a concrete BV-tree, $\mathscr{R}_S^0$ is a partition of S.*

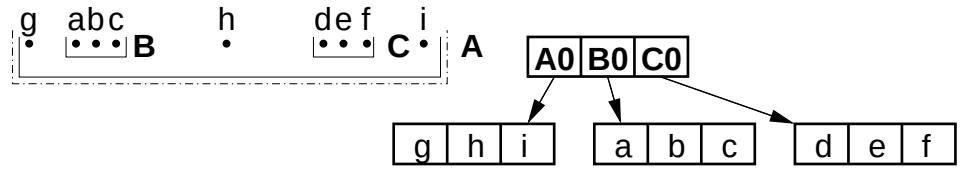
Proof For a given S, construct a BV-tree such that S is the region corresponding to two nodes of the BV-tree at different levels, such as $S0$ and $S1$. Consider distinct arbitrary regions $t, r \in \mathscr{R}_S^0$. Notice that by definition of $\mathscr{R}_S^0$, we have, $t \not\sqsubseteq r$ and $r \not\sqsubseteq t$. Moreover, by definition of $\mathscr{R}_S$, a sequence of BV-tree insertion operations can result in a split of the node $S1$ resulting in region r. Similarly, a sequence of BV-tree insertion operations can result in a split of the node $S0$ resulting in region t. As r and t are cordial by the definition of the BV-tree, we have $r \sqsubseteq t$ or $t \sqsubseteq r$ or $r \cap t = \varnothing$. However, as t and r are two distinct members of $\mathscr{R}_S^0$, $t \not\sqsubseteq r$ and $r \not\sqsubseteq t$, therefore we have $r \cap t = \varnothing$. Hence, we showed that no two elements

of $\mathscr{R}_S^0$ intersect. Therefore, $\mathscr{R}_S^0$ is a partition of S. □

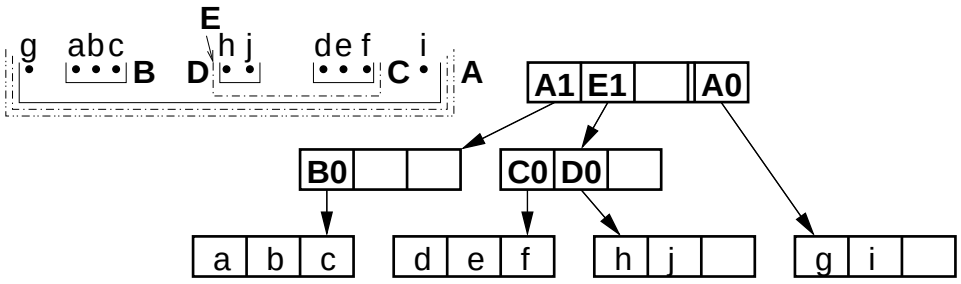
Notice that the above lemma is correct for any space S as well as the entire data space. Moreover, the partition must also satisfy the *constant splits* assumption of the BV-tree which assumes cordial regions and that given n cordial regions, it is always possible to find a region that is cordial to all of the give regions and that contains from $n/3$ to $2n/3$ of the regions. This assumption is based on a proof constructed for the hB-tree [50] which requires a binary partition, and thus in the case of the BV-tree, we can further tighten the Lemma to only be true when the partitioning scheme is binary. Hence, we have the following theorem.

Theorem 5.1 *For a given region S of a concrete BV-tree, $\mathscr{R}_S^0$ is a binary partition of S.*

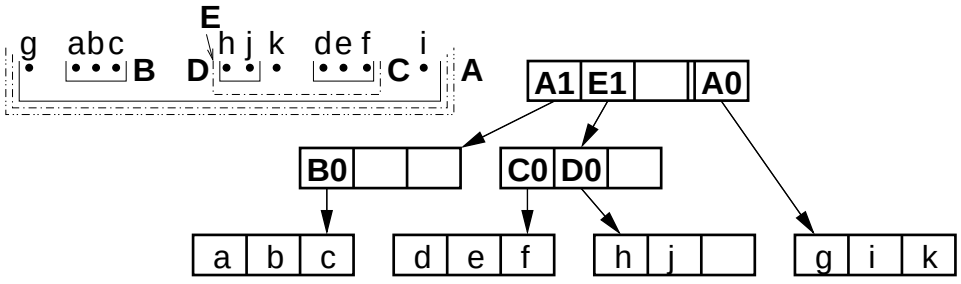
We can conclude from Theorem 5.1 that the BV-tree design assumptions require a hierarchical binary partitioning of the underlying data space. However, any binary partitioning scheme is acceptable. Therefore, it could be possible, for example, to adapt the BV-tree to metric spaces in conjunction with a *ball partitioning* scheme or a *generalized hyperplane partitioning* scheme [76].



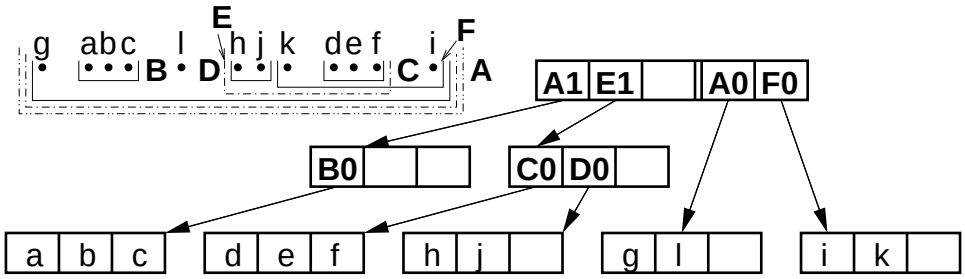
(e) After insertion of data point h, i.



(f) After insertion of data point j.

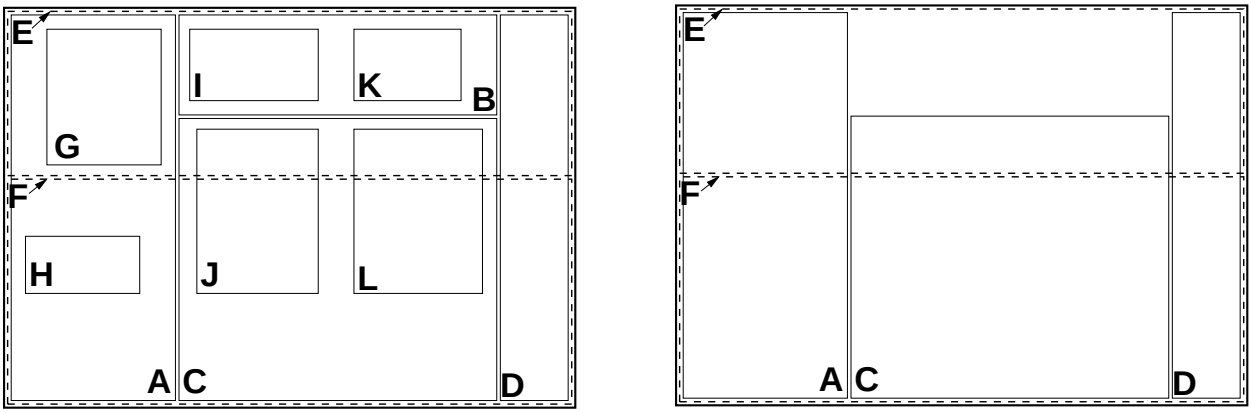


(g) After insertion of data point k.



(h) After insertion of data point l.

Figure 5.2, continued: Example of a BV-tree with intervals as regions.



(a) $\mathscr{R}_S$ (b) $\mathscr{R}_S^0$

Figure 5.3: Example illustrating the definition of $\mathscr{R}_S^0$. The region S is the outer rectangle, and the sets $\mathscr{R}_S$ and $\mathscr{R}_S^0$ consist of the inner rectangles.

Chapter 6

Conclusion and future work

This dissertation discussed several problems that deal with spatial data. In Chapter 2 we discussed our experience in extending a spatial database such it could handle spherical data. In particular, we discussed how we adapted the PMR quadtree and the R-tree to handle spherical data, such as spherical points, lines, and polygons. We further provided geometric algorithms for handling spherical data. In particular, we gave detailed implementations of algorithms for calculating the distance between spherical data. We also provided algorithms for determining the intersection of spherical data.

Our work on spherical data lead us to investigate techniques for quantizing surface normal vectors in Chapter 3. We designed the QuickArealHex algorithm which is a fast normal vector quantization algorithm with low quantization error and low memory requirements. We showed that the QuickArealHex algorithm provides (1) lower quantization error, (2) better rendering quality, and (3) better computation efficiency than the current most widely used normal vector quantization method proposed by Deering [16].

In Chapter 4, we provided a detailed CPU execution-time analysis and implementation for the top-down greedy split R-tree bulk loading algorithm of García, López, and Leutenegger [31]. The TGS algorithm makes use of a classical bottom-up packing approach. In addition, we introduced an alternative packing approach termed top-down pack-

ing which may lead to improved query performance. We also discussed a few the tradeoffs of using the bottom-up and top-down packing approaches.

The BV-tree is an abstract spatial indexing technique that is based on decoupling the hierarchy inherent in the tree structure of the directory from the containment hierarchy associated with the recursive partitioning process of the underlying space from which the data is drawn. The BV-tree is an improvement over its predecessor, the BANG file, which achieves guaranteed logarithmic search time for point queries. The BANG file decomposes the underlying space using a regular space decomposition process. In Chapter 5 we discussed a number of issues that arise in implementing a BV-tree without requiring a regular decomposition of the underlying space. In particular, we pointed out the limitations of a space decomposition where objects are aggregated using axis-aligned rectangles similar to what might be used in an object hierarchy such as an R-tree. In addition, we showed that BV-trees were only suitable for hierarchical binary space partitioning schemes.

6.1 Directions for future work

Many computer graphics applications deal with directional data. An example is surface normal vector data which was discussed at length in Chapter 3. In particular, in Chapter 3, we

showed how to project the unit sphere onto to a square. This projection allowed us to construct the Octahedral Quantization method which is a low distortion normal vector quantizer. We propose to investigate other applications of the Octahedral Quantization method in computer graphics. An important example is BRDF data [51]. The Bidirectional Reflectance Distribution Function (BRDF) models the light reflection properties of a surface as a function of two parameters, (i) the incoming direction of light and (ii) the outgoing direction of light. More complex models such as the Spatial BRDF (SBRDF) [52] and the Time and Space Varying BRDF (TSVBRDF) [38] augment the BRDF function with the

addition of other parameters. In particular, SBRDF is a function of the incoming and outgoing light directions as well the the surface position. TSV-BRDF adds a time-varying component to SBRDF in order to model the change of the reflection properties over time. For example, the SBRDF of a wet piece of wood changes over time as it dries. In recent years we have seen numerous efforts to build databases of BRDF data [38, 51]. Usually, these models are sparsely sampled. Gu et. al [38] report that interpolation the BRDF data resulted in rendered images with poor quality. Instead, they suggested building an analytic model from the BRDF data and rendering using the analytic model.

Traditionally, the directional parameters of BRDF data are parameterized using the Geographic coordinates of the incoming and outgoing light. We plan to investigate the measurement and representation of the BRDF data using techniques that are similar to the Octahedral Quantization method. We believe that the low quantization error of such techniques are also effective in improving the quality of rendering using sampled BRDF. In particular, representing each direction as a two-dimensional point allows us to easily apply quadtree-based multi-resolution techniques to BRDF data. For example, BRDF data can be represented with a scalar field over a four-dimensional hyper-cube. Hence, if the hyper-cube is subdivided using a regular decomposition, then the cells of subdivision correspond to well distributed samples of the BRDF data.

Appendix A

A.1 Derivation of the Areal Projection

In this section, we derive the equations for the Areal projection. Consider the spherical triangle $\triangle XYZ$ with vertices $X = (1,0,0)$, $Y = (0,1,0)$, and $Z = (0,0,1)$, as shown in Figure 3.9. The spherical point $N = (x,y,z) : x,y,z \geq 0$ is inside the triangle $\triangle XYZ$, and partitions the triangle into three spherical triangles $\triangle NYZ$, $\triangle XNZ$, and $\triangle XYN$. We only need to derive $\Gamma = A_S(N,Y,Z)$, the area of the spherical triangle $\triangle NYZ$, as the area of the two other triangles can be obtained similarly.

Let O denote the center of the sphere, and let $\overrightarrow{u}$, $\overrightarrow{v}$, and $\overrightarrow{w}$ denote the normal vectors of the planes passing through (O, Y, Z), (O, Z, N), and (O, N, Y) respectively. Let a, b, and c denote the side lengths of the triangle $\triangle NYZ$. We have

$$\begin{aligned} \cos d &= \overrightarrow{OZ} \cdot \overrightarrow{OY} = 0, \\ \cos e &= \overrightarrow{ON} \cdot \overrightarrow{OZ} = z, \\ \cos f &= \overrightarrow{OY} \cdot \overrightarrow{ON} = y. \end{aligned}$$

We have [79]

$$\begin{aligned}
\cos\alpha &= \frac{\cos d - \cos e\cos f}{\sin e\sin f} = -\frac{yz}{\sqrt{1-y^2}\sqrt{1-z^2}}, \\
\cos\beta &= \frac{\cos e - \cos f\cos d}{\sin f\sin d} = \frac{z}{\sqrt{1-y^2}}, \\
\cos\gamma &= \frac{\cos f - \cos d\cos e}{\sin d\sin e} = \frac{y}{\sqrt{1-z^2}}.
\end{aligned}$$

Hence, we can obtain

$$\begin{aligned}
\sin\alpha &= \sqrt{1-\cos^2\alpha} = \frac{x}{\sqrt{1-y^2}\sqrt{1-z^2}}, \\
\sin\beta &= \sqrt{1-\cos^2\beta} = \frac{x}{\sqrt{1-y^2}}, \\
\sin\gamma &= \sqrt{1-\cos^2\gamma} = \frac{x}{\sqrt{1-z^2}}.
\end{aligned}$$

Therefore,

$$\begin{aligned}
\tan\alpha &= -\frac{x}{yz}, \\
\tan\beta &= \frac{x}{z}, \\
\tan\gamma &= \frac{x}{y}.
\end{aligned}$$

Girard's spherical excess formula [79] expresses the area of a spherical triangle in terms of its internal angles. We have,

$$\Gamma = \alpha+\beta+\gamma-\pi.$$

We further simplify Γ.

$$
\begin{aligned}
\tan\Gamma &= \tan(\alpha+\beta+\gamma-\pi) \\
&= \tan(\alpha+\beta+\gamma) \\
&= \frac{\tan\alpha+\tan\beta+\tan\gamma-\tan\alpha\tan\beta\tan\gamma}{1-\tan\alpha\tan\beta-\tan\alpha\tan\gamma-\tan\beta\tan\gamma} \\
&= \frac{\frac{-x}{yz}+\frac{x}{z}+\frac{x}{y}-\frac{-x^3}{y^2z^2}}{1-\frac{-x^2}{yz^2}-\frac{-x^2}{y^2z}-\frac{x^2}{yz}} \\
&= \frac{-xyz+xy^2z+xyz^2+x^3}{y^2z^2+x^2y+x^2z-x^2yz} \\
&= \frac{x(-yz+y^2z+yz^2+x^2)}{y^2z^2+x^2(y+z-yz)} \\
&= \frac{x(-yz+y^2z+yz^2+1-y^2-z^2)}{y^2z^2+(1-y^2-z^2)(y+z-yz)} \\
&= \frac{x(yz(y-1)+(y-1)z^2-(y-1)(y+1))}{y^2z^2+(1-y^2)(y+z-yz)-z^2(y+z-yz)} \\
&= \frac{x(y-1)(yz+z^2-(y+1)}{(y^2-y-z+yz)z^2+(1-y^2)(z+y-yz)} \\
&= \frac{x(y-1)((z-1)(z+1)+y(z-1))}{(y-1)(y+z)z^2-(y-1)(y+1))(z+y(1-z))} \\
&= \frac{x(y-1)(z-1)(y+z+1)}{(y-1)((y+z)z^2-(y+1)(z-y(z-1)))} \\
&= \frac{x(y-1)((z-1)(z+1)+y(z-1))}{(y-1)((yz^2+z^3-yz-z+(y+1)y(z-1)))} \\
&= \frac{x(y-1)(z-1)(y+z+1)}{(y-1)((yz(z-1)+z(z-1)(z+1)+(y+1)y(z-1)))} \\
&= \frac{x(y-1)(z-1)(y+z+1)}{(y-1)(z-1)((yz+z(z+1)+(y+1)y))} \\
&= \frac{x(y-1)(z-1)(y+z+1)}{(y-1)(z-1)(y^2+z^2+y+z+yz)} \\
&= \frac{x(y+z+1)}{(y^2+z^2+y+z+yz)} \\
&= \frac{2x(y+z+1)}{2(y^2+z^2+y+z+yz)} \\
&= \frac{2x(y+z+1)}{2(y^2+z^2)+2(y+z+yz)}
\end{aligned}
$$

$$\begin{aligned}
\tan\Gamma &= \frac{2x(y+z+1)}{y^2+z^2+1-x^2+2(y+z+yz)} \\
&= \frac{2x(y+z+1)}{(y+z+1)^2-x^2} \\
&= \frac{\frac{2x}{y+z+1}}{1-\left(\frac{x}{y+z+1}\right)^2} \\
&= \tan\left(2\arctan\frac{x}{y+z+1}\right).
\end{aligned}$$

Hence,

$$A_S(N,Y,Z) = \Gamma = 2\arctan\frac{x}{y+z+1}.$$

We can similarly obtain,

$$A_S(X,N,Z) = 2\arctan\frac{y}{x+z+1},$$

and

$$A_S(X,Y,N) = 2\arctan\frac{z}{x+y+1}.$$

Hence, if we define the Areal projection of a point $N=(x,y,z)$ to be the point $P=(a,b,c)$ such that

$$\begin{aligned}
a &= \frac{A_S(N,Y,Z)}{\frac{\pi}{2}}, \\
b &= \frac{A_S(X,N,Z)}{\frac{\pi}{2}}, \\
c &= \frac{A_S(X,Y,N)}{\frac{\pi}{2}};
\end{aligned}$$

then we have,

$$\begin{aligned} a &= \frac{4}{\pi}\arctan\frac{x}{y+z+1}, \\ b &= \frac{4}{\pi}\arctan\frac{y}{x+z+1}, \\ c &= \frac{4}{\pi}\arctan\frac{z}{x+y+1}. \end{aligned}$$

Moreover, we have

$$\begin{aligned} \tan\frac{\pi}{4}a &= \frac{x}{y+z+1}, \\ \tan\frac{\pi}{4}b &= \frac{y}{x+z+1}, \\ \tan\frac{\pi}{4}c &= \frac{z}{x+y+1}. \end{aligned}$$

Hence,

$$\begin{aligned} \frac{\tan\frac{\pi}{4}a}{\tan\frac{\pi}{4}a+1} &= \frac{x}{x+y+z+1}, \\ \frac{\tan\frac{\pi}{4}b}{\tan\frac{\pi}{4}b+1} &= \frac{y}{x+y+z+1}, \\ \frac{\tan\frac{\pi}{4}c}{\tan\frac{\pi}{4}c+1} &= \frac{z}{x+y+z+1}. \end{aligned}$$

Defining $s(\cdot)$ such that

$$s(u) = \frac{\tan\frac{\pi}{4}u}{\tan\frac{\pi}{4}u+1},$$

We get,

$$\begin{aligned} s(a) &= \frac{x}{x+y+z+1}, \\ s(b) &= \frac{y}{x+y+z+1}, \\ s(c) &= \frac{z}{x+y+z+1}. \end{aligned}$$

Therefore,

$$1 - s(a) - s(b) - s(c) = \frac{1}{x+y+z+1}.$$

We have,

$$\begin{aligned} s(a) &= \frac{x}{1-s(a)-s(b)-s(c)}, \\ s(b) &= \frac{y}{1-s(a)-s(b)-s(c)}, \\ s(c) &= \frac{z}{1-s(a)-s(b)-s(c)}. \end{aligned}$$

And finally,

$$\begin{aligned} x &= \frac{s(a)}{1-s(a)-s(b)-s(c)}, \\ y &= \frac{s(b)}{1-s(a)-s(b)-s(c)}, \\ z &= \frac{s(c)}{1-s(a)-s(b)-s(c)}. \end{aligned}$$

Bibliography

[1] ARGE, L., HINRICHS, K. H., VAHRENHOLD, J., AND VITTER, J. S. Efficient bulk operations on dynamic R-trees. In *ALENEX '99: Proc. of the 1st Workshop on Algorithm Engineering and Experimentation* (Jan. 1999), vol. 1619 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 328–348.

[2] ARVO, J., AND KIRK, D. Fast ray tracing by ray classification. *SIGGRAPH Computer Graphics 21*, 4 (1987), 55–64.

[3] AURENHAMMER, F. Voronoi diagrams — a survey of a fundamental geometric data structure. *ACM Comput. Surv. 23*, 3 (Sept. 1991), 345–405.

[4] BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., AND SEEGER, B. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD '90: Proc. of the Intl. Conf. on Management of Data* (New York, NY, May 1990), H. Garcia-Molina and H. V. Jagadish, Eds., ACM Press, pp. 322–331.

[5] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept. 1975), 509–517.

[6] BLINN, J. F. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proc. of the 5th Annual Conf. on Computer Graphics and Interactive Techniques* (New York, NY, Aug. 1978), ACM Press, pp. 286–292.

[7] BLINN, J. F., AND NEWELL, M. E. Texture and reflection in computer generated images. *Commun. ACM 19* (1976), 542–546.

[8] BOTSCH, M., WIRATANAYA, A., AND KOBBELT, L. Efficient high quality rendering of point sampled geometry. In *EGRW '02: Proc. of the 13th Eurographics Workshop on Rendering* (Pisa, Italy, June 2002), pp. 53–64.

[9] BUSS, S. R., AND FILLMORE, J. P. Spherical averages and applications to spherical splines and interpolation. *ACM Trans. Gr. 20*, 2 (Apr. 2001), 95–126.

[10] CAI, M., KESHWANI, D., AND REVESZ, P. Z. Parametric rectangles: A model for querying and animating spatiotemporal databases. In *EDBT '00: Proc. of the 7th Conf. on Extending Database Technology* (Mar. 2000), C. Zaniolo, P. C. Lockemann, M. H. Scholl, and T. Grust, Eds., vol. 1777 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 430–444.

[11] CHEN, L., CHOUBEY, R., AND RUNDENSTEINER, E. A. Bulk-insertions into R-trees using the Small-Tree-Large-Tree approach. In *GIS '98: Proc. of the 6th ACM Intl. Symp. on Advances in Geographic Information Systems* (New York, NY, 1998), R. Laurini, K. Makki, and N. Pissinou, Eds., ACM Press, pp. 161–162.

[12] CHEN, Z. T., AND TOBLER, W. R. Quadtree representations of digital terrain. In *Proc. of Auto-Carto London* (London, United Kingdom, Sept. 1986), vol. 1, pp. 475–484.

[13] CHOUBEY, R., CHEN, L., AND RUNDENSTEINER, E. A. GBI: A generalized R-tree bulk-insertion strategy. In *SSD '99: Proc. of the 6th Intl. Symp. on Advances in Spatial Databases* (July 1999), R. H. Güting, D. Papadias, and F. H. Lochovsky, Eds., vol. 1651 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 91–108.

[14] CIGNONI, P., DE FLORIANI, L., MAGILLO, P., PUPPO, E., AND SCOPIGNO, R. Selective refinement queries for volume visualization of unstructured tetrahe-

dral meshes. *IEEE Transactions on Visualization and Computer Graphics 10*, 1 (Jan. 2004), 29–45.

[15] COMER, D. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June 1979), 121–137.

[16] DEERING, M. Geometry compression. In *SIGGRAPH '95: Proc. of the 22nd Annual Conf. on Computer Graphics and Interactive Techniques* (New York, NY, Aug. 1995), ACM Press, pp. 13–20.

[17] DEVILLERS, O., AND GANDOIN, P.-M. Geometric compression for interactive transmission. In *VIS '00: Proc. of the 11th Conf. on Visualization* (2000), pp. 319–326.

[18] DU, Q., GUNZBURGER, M. D., AND JU, L. Constrained centroidal Voronoi tessellations for surfaces. *SIAM Journal on Scientific Computing 24*, 5 (2003), 1488–1506.

[19] DUTTON, G. Zenithial orthotriangular projection. In *Proc. of the Tenth Intl. Conf. on Computer-Assisted Cartography (Auto-Carto 10)* (Baltimore, MD, Mar. 1991), pp. 77–95.

[20] ESPERANÇA, C., AND SAMET, H. Spatial database programming using SAND. In *Proc. of the 7th Intl. Symp. on Spatial Data Handling* (Delft, The Netherlands, Aug. 1996), M. J. Kraak and M. Molenaar, Eds., vol. 2, Intl. Geographical Union Commission on Geographic Information Systems, Association for Geographical Information, pp. A29–A42.

[21] ESPERANÇA, C., AND SAMET, H. Experience with SAND/Tcl: a scripting tool for spatial databases. *Journal of Visual Languages and Computing 13*, 2 (Apr. 2002), 229–255.

[22] EVANGELIDIS, G., LOMET, D., AND SALZBERG, B. The hB$^{\Pi}$-tree: a multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal 6*, 1 (Jan. 1997), 1–25.

[23] FEKETE, G., AND TREINISH, L. Sphere quadtrees: a new data structure to support the visualization of spherically distributed data. In *Extracting Meaning from Complex Data: Processing, Display, Interaction* (Aug. 1990), E. J. Farrell, Ed., vol. 1259 of *Proc. of SPIE/SPSE Symp. on Electronic Imaging Science and Technology*, pp. 242–253.

[24] FINKEL, R. A., AND BENTLEY, J. L. Quad trees: a data structure for retrieval on composite keys. *Acta Inf. 4*, 1 (Mar. 1974), 1–9.

[25] FREESTON, M. The BANG file: A new kind of grid file. In *SIGMOD '87: Proc. of the Intl. Conf. on Management of Data* (New York, NY, May 1987), U. Dayal and I. L. Traiger, Eds., ACM Press, pp. 260–269.

[26] FREESTON, M. A general solution of the n-dimensional B-tree problem. In *SIGMOD '95: Proc. of the Intl. Conf. on Management of Data* (New York, NY, May 1995), M. J. Carey and D. A. Schneider, Eds., ACM Press, pp. 80–91.

[27] FREESTON, M. On the complexity of BV-tree updates. In *CDB '97: Proc. of the 2nd Intl. Workshop on Constraint Database Systems* (Jan. 1997), V. Gaede, A. Brodsky, O. Günther, D. Srivastava, V. Vianu, and M. Wallace, Eds., vol. 1191 of *Lecture Notes in Computer Science*, pp. 282–293.

[28] FREESTON, M. W. A well-behaved file structure for the storage of spatial objects. In *SSD '89: Proc. of the 1st Symp. on Advances in Spatial Databases* (July 1989), A. P. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, Eds., vol. 409 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 287–300.

[29] GANDOIN, P.-M., AND DEVILLERS, O. Progressive lossless compression of arbitrary simplical complexes. *ACM Trans. Gr. 21*, 3 (July 2002), 372–379.

[30] GARCÍA, Y. J., LÓPEZ, M. A., AND LEUTENEGGER, S. T. A greedy algorithm for bulk loading R-trees. Computer Science Technical Report 97-02, University of Denver, Denver, CO, 1997.

[31] GARCÍA R., Y. J., LÓPEZ, M. A., AND LEUTENEGGER, S. T. A greedy algorithm for bulk loading R-trees. In *GIS '98: Proc. of the 6th ACM Intl. Symp. on Advances in Geographic Information Systems* (New York, NY, 1998), R. Laurini, K. Makki, and N. Pissinou, Eds., ACM Press, pp. 163–164.

[32] GARGANTINI, I. An effective way to represent quadtrees. *Commun. ACM 25*, 12 (Dec. 1982), 905–910.

[33] GOODCHILD, M. F., AND SHIREN, Y. A hierarchical data structure for global geographic information systems. In *SDH '90: Proc. of the 4th Intl. Symp. on Spatial Data Handling* (July 1992), vol. 1, pp. 911–917.

[34] GRABNER, M. Compression of arbitrary triangle meshes with attributes for selective refinement. *Journal of WSCG 11*, 1 (2003).

[35] GRAY, J., SZALAY, A. S., FEKETE, G., NIETO-SANTISTEBAN, M. A., O'MULLANE, W., THAKAR, A. R., HEBER, G., AND ROTS, A. H. There goes the neighborhood: Relational algebra for spatial data search. Tech. Rep. MSR-TR-2004-32, Microsoft Research, Redmond, WA, Apr. 2004.

[36] GREENE, N. Environment mapping nd other applications of world projections. *IEEE CG&A* (Nov. 1986), 21–29.

[37] GROSSMAN, J. P. Point sample rendering. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug. 1998.

[38] GU, J., TU, C.-I., RAMAMOORTHI, R., BELHUMEUR, P., MATUSIK, W., AND NAYAR, S. Time-varying surface appearance: Acquisition, modeling and rendering. *ACM Trans. Gr. 25*, 3 (July 2006), 762–771.

[39] GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. In *SIGMOD '84: Proc. of the Intl. Conf. on Management of Data* (New York, NY, June 1984), B. Yormark, Ed., ACM Press, pp. 47–57.

[40] HJALTASON, G. R., AND SAMET, H. Incremental distance join algorithms for spatial databases. In *SIGMOD '98: Proc. of the Intl. Conf. on Management of Data* (New York, NY, June 1998), L. M. Haas and A. Tiwary, Eds., ACM Press, pp. 237–248.

[41] HJALTASON, G. R., AND SAMET, H. Distance browsing in spatial databases. *ACM Trans. Database Syst. 24*, 2 (June 1999), 265–318.

[42] HJALTASON, G. R., AND SAMET, H. Improved bulk-loading algorithms for quadtrees. In *GIS '98: Proc. of the 7th ACM Intl. Symp. on Advances in Geographic Information Systems* (New York, NY, 1999), R. Laurini, K. Makki, and N. Pissinou, Eds., ACM Press, pp. 110–115.

[43] HUNTER, G. M., AND STEIGLITZ, K. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2 (Apr. 1979), 145–153.

[44] KALAIAH, A., AND VARSHNEY, A. Statistical geometry representation for efficient transmission and rendering. *ACM Trans. Gr. 24*, 2 (2005), 348–373.

[45] KAMEL, I., AND FALOUTSOS, C. On packing R-trees. In *CIKM '93: Proc. of the 2nd Intl. Conf. on Information and Knowledge Management* (New York, NY, Nov. 1993), B. Bhargava, T. Finin, and Y. Yesha, Eds., ACM Press, pp. 490–499.

[46] KLINGER, A. Patterns and search statistics. In *Optimizing Methods in Statistics*, J. S. Rustagi, Ed. Academic Press, New York, NY, 1971, pp. 303–337.

[47] KRIEGEL, H.-P., PÖTKE, M., AND SEIDL, T. Managing intervals efficiently in object-relational databases. In *VLDB '00, Proc. of 26th Intl. Conf. on Very Large Data Bases* (Sept. 2000), A. El Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds., Morgan Kaufmann, pp. 407–418.

[48] KUGLER, A. IMEM: An intelligent memory for bump- and reflection-mapping. In *HWWS '98: Proc. of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, Aug. 1998), Eurographics Association, pp. 113–122.

[49] LI, J., ROTEM, D., AND SRIVASTAVA, J. Algorithms for loading parallel grid files. In *SIGMOD '93: Proc. of the Intl. Conf. on Management of Data* (New York, NY, May 1993), P. Buneman and S. Jajodia, Eds., ACM Press, pp. 347–356.

[50] LOMET, D. B., AND SALZBERG, B. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst. 15*, 4 (Dec. 1990), 625–658.

[51] MATUSIK, W., PFISTER, H., BRAND, M., AND MCMILLAN, L. Efficient isotropic BRDF measurement. In *EGRW '03: Proc. of the 14th Eurographics Workshop on Rendering* (Leuven, Belgium, June 2003), pp. 241–247.

[52] MCALLISTER, D. K., LASTRA, A., AND HEIDRICH, W. Efficient rendering of spatial bi-directional reflectance distribution functions. In *HWWS '02: Proc. of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, Aug. 2002), Eurographics Association, pp. 79–88.

[53] MOUNT, D. M., AND ARYA, S. ANN: A library for approximate nearest neighbor searching, May 2005. `http://www.cs.umd.edu/users/mount/ANN/`.

[54] NELSON, R. C., AND SAMET, H. A consistent hierarchical representation for vector data. *SIGGRAPH Computer Graphics 20*, 4 (Aug. 1986), 197–206.

[55] PRAUN, E., AND HOPPE, H. Spherical parametrization and remeshing. *ACM Trans. Gr. 22*, 3 (July 2002), 340–349.

[56] REVESZ, P. *Introduction to Constraint Databases*. Springer, New York, NY, 2002.

[57] RIGAUX, P., SCHOLL, M., AND VOISARD, A. *Spatial Databases: with Applications to GIS*. Morgan-Kaufmann, San Francisco, 2001.

[58] ROUSSOPOULOS, N., AND LEIFKER, D. Direct spatial search on pictorial databases using packed R-trees. In *SIGMOD '85: Proc. of the Intl. Conf. on Management of Data* (New York, NY, May 1985), S. B. Navathe, Ed., ACM Press, pp. 17–31.

[59] RUSINKIEWICZ, S., AND LEVOY, M. QSplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proc. of the 27th Annual Conf. on Computer Graphics and Interactive Techniques* (New York, NY, July 2000), ACM Press, pp. 343–352.

[60] SAFF, E. B., AND KUIJLAARS, A. B. J. Distributing many points on a sphere. *Mathematical Intelligencer 19*, 1 (1997), 5–11.

[61] ŠALTENIS, S., JENSEN, C. S., LEUTENEGGER, S. T., AND LÓPEZ, M. A. Indexing the positions of continuously moving objects. In *SIGMOD '00: Proc. of the 2000 Intl. Conf. on Management of Data* (New York, NY, May 2000), ACM Press, pp. 331–342.

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

[63] SAMET, H. Decoupling partitioning and grouping: Overcoming shortcomings of spatial indexing with bucketing. *ACM Trans. Database Syst. 29*, 4 (Dec. 2004), 789–830.

[64] SAMET, H. Object-based and image-based object representations. *ACM Comput. Surv. 36*, 2 (June 2004), 159–217.

[65] SAMET, H. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, 2006.

[66] SAMET, H., ALBORZI, H., BRABEC, F., ESPERANÇA, C., HJALTASON, G. R., MORGAN, F., AND TANIN, E. Use of the SAND spatial browser for digital government applications. *Commun. ACM 46*, 1 (Jan. 2003), 63–66.

[67] SCOTT, G. M. The cubic quadtree: a spatial data structure for spherical surfaces. scholarly paper CSC 1024, University of Maryland, College Park, MD, Dec. 1996.

[68] SELLIS, T., ROUSSOPOULOS, N., AND FALOUTSOS, C. The R^+-tree: a dynamic index for multi-dimensional objects. In *VLDB '87: Proc. of the 13th Intl. Conf. on Very Large Databases* (Sept. 1987), Morgan Kaufmann, pp. 507–518.

[69] SNYDER, J. P. *Map Projections - A Working Manual*. United States Government Printing Office, Washington, DC, 1987.

[70] SONG, L., KIMERLING, A. J., AND SAHR, K. Developing an equal area global grid by small circle subdivision. In *Discrete Global Grids* (Santa Barbara, CA, 2002), M. Goodchild and A. J. Kimerling, Eds., National Center for Geographic Information and Analysis.

[71] TAUBIN, G., HORN, W. P., LAZARUS, F., AND ROSSIGNAC, J. Geometry coding and VRML. *Proc. of the IEEE 86*, 6 (June 1998), 1228–1243.

[72] TAUBIN, G., AND ROSSIGNAC, J. Geometric compression through topological surgery. *ACM Trans. Gr. 17*, 2 (Apr. 1998), 84 – 115.

[73] TEGMARK, M. An icosahedron-based method for pixelizing the celestial sphere. *Astrophysical Journal Letters 470* (Oct. 1996), 81–84.

[74] TOBLER, W., AND CHEN, Z. T. A quadtree for global information storage. *Geographical Analysis 18*, 4 (Oct. 1986), 360–371.

[75] TOBOR, I., SCHLICK, C., AND GRISONI, L. Rendering by surfels. In *GRAPHICON '00: Proc. of the 10th Intl. Conf. on Computer Graphics & Vision* (Aug. 2000), pp. 193–204.

[76] UHLMANN, J. K. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett. 40,* 4 (Nov. 1991), 175–179.

[77] VAN DEN BERCKEN, J., SEEGER, B., AND WIDMAYER, P. A generic approach to bulk loading multidimensional index structures. In *VLDB '97: Proc. of the 23rd Intl. Conf. on Very Large Databases* (Aug. 1997), M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds., Morgan Kaufmann, pp. 406–415.

[78] WANG, W., YANG, J., AND MUNTZ, R. R. PK-tree: A spatial index structure for high dimensional point data. In *Proc. of the 5th Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)* (Nov. 1998), K. Tanaka and S. Ghandeharizadeh, Eds., pp. 27–36.

[79] WEISSTEIN, E. W. *The CRC Concise Encyclopedia of Mathematics*. CRC Press, Boca Raton, FL, 1998.

[80] WHITE, D., KIMERLING, A. J., SAHR, K., AND SONG, L. Comparing area and shape distortion on polyhedral-based recursive partitions of the sphere. *Intl. Journal of Geographical Information Science 12,* 8 (Dec. 1998), 805–827.

[81] WILLMOTT, A. J. *Hierarchical Radiosity with Multiresolution Meshes*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Dec. 2000.

[82] WITTEN, I. H., NEAL, R. M., AND CLEARY, J. G. Arithmetic coding for data compression. *Commun. ACM 30,* 6 (June 1987), 520–540.

[83] YANG, J., WANG, W., AND MUNTZ, R. Yet another spatial indexing structure. Computer Science Technical Report 97040, University of California at Los Angeles, Los Angeles, Nov. 1997.