CAR-TR-816 DEFG02-95ER25237
CS-TR-3609 IRI-92-16970
February 1996

Surface Modeling Using Quadtrees

Ron Sivan

Computer Vision Laboratory
Center for Automation Research
University of Maryland
College Park, MD 20742-3275

Abstract

Two quadtree variants effective in modeling $2\frac{1}{2}$-D surfaces are presented. The restricted quadtree can handle regularly sampled data. For irregular data, embedding a TIN inside a PMR quadtree is suggested. Together, these schemes facilitate the handling of most types of input within a single framework. Algorithms for the construction of both data structures from their respective data formats are described and analyzed. The possible application of each of the models to the problem of visibility determination is considered and its performance is theoretically evaluated.

The support of the Department of Energy under Grant DEFG02-95ER25237 and the National Science Foundation under Grant IRI-92-16970 is gratefully acknowledged, as is the help of Sandy German in preparing this paper.

Preface

This dissertation deals with modeling of surfaces, i.e., data structures that can facilitate the storage and manipulation of objects resembling topographical surfaces using computers.

Such surfaces, also known as $2\frac{1}{2}$-dimensional ($2\frac{1}{2}$-D) data, arise frequently in GIS applications (hence the term Digital Terrain Modeling (DTM) [Fowl79] is often applied to this process), but is not restricted to them. DTMs may be used wherever a continuous single-valued function of two variables adequately describes the objects of interest.

Where appropriate, computers may be used to manipulate the surface in ways that heretofore were done mostly manually. Graphical display, surface intersection, map production, visibility determination, path finding, locating basins and divides, are all examples of problems whose solution is facilitated using computers.

One common technique of modeling $2\frac{1}{2}$-D surfaces employs a polyhedral approximation: the curved, complex reality is approximated by a set of planar polygons connected in three-dimensional (3-D) space. The elevation at any point of the surface is then either explicitly stored or can be interpolated from the values stored for nearby points. The space requirements are thereby reduced. By adjusting the density (i.e., number and size) of the polygons, many applications can be adequately handled. The density need not be uniform across the surface but instead can be adaptive, calculated at each point to accommodate the variability that the surface exhibits there.

Quadtrees (e.g., [Same90a]) are such an adaptive data structure in which a planar shape is recursively subdivided according to some rule. By adjusting their decomposition rules, quadtrees have been found useful in many spatial applications. However, the classic region quadtree suffers from an inherent deficiency which makes it inappropriate for surface modeling. In this dissertation we will describes two quadtree variants which have been modified to overcome this deficiency and accommodate surface modeling: the restricted quadtree (RQT) [VonH89] for regularly-sampled surface data and the PMR quadtree [Nels86a] for irregularly-sampled data. Both these structures have been previously described, but have not been previously studied in the context of surface modeling. It is felt that quadtrees may be able to handle both regular and irregular data sufficiently well to allow their combined processing, a goal which so far seems to have eluded other techniques.

In order to verify the theoretical analysis of the RQT and PMR structures as surface models, both were implemented using a common quadtree engine, developed within the framework of the QUILT project [Shaf90b]. It is hoped that any deficiencies this engine may suffer from affect both models more or less equally. As a result, any differences in performance between the models can be attributed more to the intrinsics of their operations than to implementation details. To compare the models' performance, both were applied to

the problem of the horizon, which determines the visible portion of a given surface from a given point.

This dissertation is organized as follows: Chapter 1 briefly reviews some theoretical aspects of surface modeling and attempts to establish criteria for DTM classification based on the approach each takes towards this task. This is followed by a brief survey of DTMs in the scientific literature (Chapter 2), with special attention given to data structures similar to those discussed here. Chapter 3 describes the restricted quadtree approach in greater detail, including a comparison between two different ways of constructing such a model from raster data. Chapter 4 describes the other quadtree type, the PMR quadtree, and includes a discussion of several other operations, such as windowing and nearest neighbor finding, which make the PMR quadtree useful for other purposes as well. Chapter 5 focuses on Triangulated Irregular Networks (TINs) [Peuc75] stored inside PMR quadtrees as surface models suitable for irregular data. The case study of horizon extraction is described in Chapter 6, where the performance of both models is compared. Finally, conclusions are drawn and further research is suggested in Chapter 7.

Acknowledgements

I wish to thank the many people that have given me their help and support in the course of this undertaking.

First, thanks go to my advisor, Prof. Hanan Samet, whose understanding and judgment proved invaluable for this work. His diligence and uncompromising rigor could always be relied upon. He maintained an environment conducive to research and insisted that I made use of it. I do appreciate his putting up with me for all these years.

Thanks are due to the other members of my committee: Profs. Larry Davis, Azriel Rosenfeld, Samuel Goward (who made himself available on very short notice), and Dave Mount, from whom I have learned a great deal about computational geometry and even more about teaching.

I would like to thank Drs. Lee Schneyer and Sam Steppel for helping me through the more difficult stretches of this journey. Being familiar with the hardships involved in obtaining a graduate degree as well as many other aspects of life, their advice has been invaluable.

Students toiling under the demands of their own programs have often found time to help me with the many difficulties that can beset one during research: from finding a reference to understanding the bizarre workings of LaTeX. I wish to thank Drs. Mike Dillencourt, Walid Aref, Selene Bestul, Enrico Puppo and David Doermann. Just as helpful were soon-to-be PhDs Erik Hoel, Claudio Esperança, and Gisli Hjaltason. Special thanks go to Ehud Rivlin and, last but far from least, Aya Soffer, whose "just do it" attitude was often the thing that got me to do it.

All this would not have been possible without the unflagging support of my family: My parents, my daughters, and most of all, my wife, who put me on this path initially and saw me through it to the end.

Table of Contents

Section	Page
List of Tables	**viii**
List of Figures	**ix**
1 Theoretical Overview	**1**
1.1 Issues Raised by Surface Modeling	1
1.2 Classification Criteria	2
1.2.1 Area Subdivision	3
1.2.2 Function Set	5
1.2.3 Hierarchy	5
1.3 Triangulations	6
1.3.1 More about Area Subdivisions as Planar Graphs	7
1.3.2 Triangles Are Advantageous	8
1.3.3 Triangulated Irregular Networks	8
1.3.4 Triangular Decompositions	9
2 Previous Work	**11**
2.1 Regular Polyhedral Models	11
2.1.1 Data Compression	11
2.1.2 Triangular Bintrees	12
2.1.3 Planetary Relief	14
2.1.4 Semi-regular Model	16
2.2 Irregular Polyhedral Models	17
2.2.1 Triangulated Irregular Networks	17
2.2.2 Hierarchical TINs	18
2.2.3 Cartographic Coherence	20
2.3 Non-Polyhedral Models	21
2.3.1 Curved Surfaces	21
2.3.2 Fractals	22

3 The Restricted Quadtree **24**
3.1 Introduction . . . 24
3.2 Definitions . . . 24
3.2.1 Restricted Quadtree Definition . . . 24
3.2.2 Restricted Quadtree Variants . . . 25
3.2.3 Related Concepts . . . 26
3.3 Implementation . . . 28
3.3.1 Assumptions . . . 28
3.3.2 Atomic Nodes . . . 29
3.4 RQT Construction Algorithms . . . 31
3.4.1 The bottom-up Construction Algorithm . . . 32
3.4.2 The top-down Construction Algorithm . . . 35
3.4.3 Experimental Results . . . 45

4 The PMR Quadtree **50**
4.1 Introduction . . . 50
4.2 Definition . . . 50
4.3 Nearest Object . . . 51
4.3.1 Motivation . . . 51
4.3.2 The Principle . . . 51
4.3.3 The Expanded Block List . . . 52
4.3.4 The Algorithm . . . 54
4.3.5 Nearest Object in 3-D . . . 56
4.3.6 Nearest Object in Arbitrary Dimensions . . . 56
4.4 Windowing . . . 61
4.4.1 Previous Work . . . 61
4.4.2 The Algorithm . . . 62

5 Irregular Triangulations and Quadtrees **71**
5.1 Motivation . . . 71
5.2 Implementation . . . 71
5.2.1 Decomposition Rule . . . 72
5.2.2 Object-Block Intersection Rule . . . 72

6 Field of View: a Test Case **74**
6.1 Field Of View . . . 74
6.2 Priming . . . 75
6.3 A Data Structure For Horizon Modeling . . . 77
6.3.1 The Wings . . . 77

6.3.2 Processing a Surface Facet 78
6.4 Sorting the Facets 81
6.4.1 Common Issues 81
6.4.2 Sorting the Facets in the RQT Model 81
6.4.3 Sorting the Facets in the QTN Model 83
6.5 Experimental Results 86

7 Conclusions and Future Research **90**

Bibliography **91**

List of Tables

Number		Page
3.1	Costs associated with the different scenarios a node may be subject to in procedure restrict.	40
3.2	Timings of the RQT bottom-up and top-down constructions.	46
4.1	The volume of a d-dimensional hypersphere having unit diameter.	58
4.2	A comparison of the size of the core and expanded block lists for various dimensions.	60
6.1	Experimental results of field of view determination.	86

List of Figures

Number		Page
1.1	Surface discontinuity resulting from non-shared vertices.	4
1.2	Conditions under which cracks are likely.	5
1.3	Proof that maximal planar graphs have no non-shared vertices.	7
1.4	Example of the difficulty slivers present in interpolation.	9
1.5	Examples of triangle decomposition.	9
2.1	Triangular bintree: example of construction.	12
2.2	Cascading splits in a triangular bintree.	13
2.3	Triangular bintree correspondence with a binary tree.	13
2.4	Decomposition in the Geodesic Elevation Model.	15
2.5	Traditional quaternary decomposition.	16
2.6	Semi-regular decomposition derived from an irregular dataset.	17
2.7	A Delaunay triangulation of a superset of points is not necessarily a refinement of the triangulation of the original point set.	19
2.8	Triangulation with and without cartographic coherence.	20
2.9	The ways to split a triangle with cartographic coherence.	21
3.1	Restricted and non-restricted versions of a sample quadtree.	25
3.2	Restricted quadtree node relationships.	26
3.3	Mandatory vs. optional vertices in RQT nodes.	26
3.4	The 16 possible configurations of an RQT node.	27
3.5	Inherited vertices in RQT nodes.	27
3.6	Bad cases of arbitrary datasets embedded into an RQT square.	29
3.7	Refinement does not monotonically lead to better models.	30
3.8	The construction of atomic nodes from raw input data	31
3.9	Expected complexities of the bottom-up vs. top-down algorithms.	32
3.10	bottom-up construction algorithm in pseudo-code.	33
3.11	Required conditions for merging nodes in an RQT.	34
3.12	Merging RQT nodes	35

3.13 top-down construction algorithm: location of uninherited vertices 36
3.14 Example of node insertion when using the top-down algorithm 37
3.15 Steps in the execution of procedure incorporate. 38
3.16 Procedure incorporate which incorporates a node into an RQT. 39
3.17 Procedure top-down which constructs a RQT from raster data. 40
3.18 Procedure load which loads input data into an empty RQT. 41
3.19 Procedure restrict: convert a non-restricted model into an RQT. 42
3.20 Procedure backtrack: recursively handles large marked nodes. 43
3.21 Procedure split-file: splits the input into subfiles small enough to fit in RAM. 44
3.22 top-down construction algorithm: out-of-core version. 45
3.23 Construction times of map "Salisbury east 2,0". 47
3.24 Construction times of map "data". 47
3.25 Construction times of map "Reno west 0,0". 48
3.26 Perspective display of the 513×513 surface "Salisbury east 2,0". 48
3.27 Perspective display of the 513×513 surface "data". 49
3.28 Perspective display of the 513×513 surface "Reno west 0,0". 49

4.1 Scope of search for a nearest object in a PMR quadtree. 52
4.2 Different cases of search spheres in a 2-d PMR quadtree. 53
4.3 Blocks that should be searched in a 2-D PMR quadtree. 54
4.4 Algorithm to find the object nearest to a point in a PMR quadtree. 55
4.5 Blocks that should be searched in a 3-D PMR quadtree. 56
4.6 Algorithm to produce expanded block list in 3-D. 57
4.7 The list of blocks to traverse to find nearest objects in 3-D. 59
4.8 Various ways of interpreting a windowing operation. 61
4.9 Algorithm window to find the objects intersecting a window. 64
4.10 Algorithm window-block: extract windowed objects from a block. 65
4.11 A walk through the operation of window at levels 0 and 1. 66
4.12 A walk through the operation of window: level 2. 67
4.13 A walk through the operation of window: level 3. 68
4.14 A walk through the operation of window: level 4. 69
4.15 A walk through the operation of window: conclusion. 70

5.1 Touching objects in a bucket PMR quadtree 73

6.1 Algorithm field-of-view: determine the visible part of a surface. 75
6.2 Possible configurations of the viewpoint and its incident facets. 76
6.3 Possible combinations of surface facet distance and size. 77
6.4 The four wings onto which surface facets are projected. 78

6.5 Algorithm horizon: (1) Initial state. 79
6.6 Algorithm horizon: (2) Projecting a facet onto the "wing". 79
6.7 Algorithm horizon: (3) The impact a facet has on the horizon. 80
6.8 Algorithm horizon: (4) projecting the visible portion onto the facet. 80
6.9 An example of a set of triangles defying spatial sorting. 82
6.10 The zones induced by a viewpoint for RQT block sorting. 82
6.11 The eight possible orientations of a block w/r to the viewpoint. 83
6.12 Algorithm rqt-sort to sort the facets of an RQT. 84
6.13 Algorithm qtn-sort: stages in the development of the active border. 85
6.14 Algorithm qtn-sort incremental step: possible configurations. 85
6.15 Algorithm qtn-sort to sort the facets of a QTN. 88
6.16 field-of-view algorithm execution times vs. model size. 89
6.17 field-of-view algorithm execution times vs. model tolerance. 89

Chapter 1

Theoretical Overview

The modeling of surfaces for use in digital computers is known as Digital Terrain Modeling (DTM) [Fowl79]. The term "terrain" is used due to the frequent application of DTMs to problems of topography. However, DTMs may be used wherever a continuous single-valued function of two variables adequately describes the objects of interest. Where appropriate, computers may be used to manipulate the surface in ways that heretofore were done mostly manually. Graphical display, surface intersection, map production, visibility determination, path finding, locating basins and divides, are all examples of problems whose solution is facilitated using computers.

Many different approaches to surface modeling have been put forward in the literature in the last two decades. It is useful to classify them and find how the present research relates to other work in the field. Surface modeling raises several issues that every solution must confront. Section 1.1 focuses on those issues first so that they only need to be hinted at when discussing the ways the various DTMs address them. Section 1.2 lists the classification criteria and some of the choices DTMs make in that regard. Due to the important role of triangles in DTMs, Section 1.3 provides an overview of, and a justification for, triangulations in general and TINs in particular.

1.1 Issues Raised by Surface Modeling

A model of a surface should, as a minimum, be able to predict the elevation[1] of the surface at any (x, y) location. However, it is impossible to completely describe a continuous entity such as a surface using the discrete-mathematical capabilities of computers. Only surfaces conforming (at least piecewise) to some concisely-expressible analytical function can even be defined completely, and natural surfaces are seldom, if ever, manifestations of such functions.

Surface modeling, then, is essentially a process of approximation. In principle, we are looking for a function or a finite set of functions which collectively provide a sufficiently accurate description of the surface for the task at hand. Every DTM, therefore, is associated

[1] In keeping with the metaphor of terrain, the values of the surface function will be referred to as "elevations", although elevation may not be the only variable for which a DTM is used even in applications to topography. Also, we will assume a Cartesian coordinate system, and refer to elevation sometimes as a z value, although the ideas presented here do not depend on the coordinate system used.

with a tolerance value which indicates the maximum allowable deviation between the actual surface and the model.

If, as is usually the case, the surface cannot be characterized precisely in mathematical terms, then the only way to define it is by specifying its elevation at certain locations, information normally obtained as a result of some sampling process applied to the surface. The sampling may be done manually or be produced by an automatic acquisition system. Elevations may be specified at regular intervals or only at some irregularly distributed locations. The choice between the two signifies more than a method for selecting sample points. An irregular sampling process is usually driven by the shape of the surface being modeled, measuring its elevation at points where its trend changes, such as at peaks, pits, ridges, valleys, and saddle points. A regular scheme, on the other hand, samples at points tied to the space the surface occupies. As a result, irregular descriptions are invariant to rotations and translations of the surface, while regular descriptions are not.

The distribution of the available raw data, therefore, has an impact on the types of DTMs which may be successfully constructed from it. As will be demonstrated below, some DTM schemes, such as the restricted quadtree, are amenable to a regular grid of samples, whereas others, such as the TIN, are satisfied with randomly distributed samples. Once a DTM of the surface is created, the model itself can be queried to produce one sampling set given the other, but the accuracy of this procedure is obviously limited by the accuracy of the model itself.

In general, the more sample points provided, the better the surface description. By sampling the surface at a high enough spatial density, any level of accuracy can be supported. The problem is that such a representation can prove too voluminous to be of any use. One of the main goals of our study of DTMs is to find ways to minimize the space required to model a surface to a given tolerance.

1.2 Classification Criteria

As hinted above, mathematical functions can provide the elevation at any point on a surface given only a handful of parameters, providing a very concise description for many types of surfaces. DTMs almost invariably harness this capability to achieve data compression by splitting the surface area into a set of sub-areas, each of which is small enough to be adequately described by such mathematical forms. Each sub-area is then covered with a patch whose 3-d shape can be made to conform, within the DTM tolerance, to the piece of the surface being modeled. Often, the smaller the sub-areas, the smaller the tolerances which can be met, at the obvious expense of an increase in the number of sub-areas which need to be maintained.

Many applications require access to the same surface at several accuracy levels within a single task. Searches, for example, can often be better accommodated by accessing a low-resolution representation first (where the search space may be smaller) and moving to higher resolutions only after a general location has been verified and the search space pruned. Almost any scheme which can describe a surface to a given tolerance can be expanded to stack several representations of the surface using a variety of tolerances. The added time invested in constructing such multi-resolution data structures can pay off when, at

run time, access to the various representations is readily available. However, to make the vertical movement among descriptions of different resolutions effective, it is often essential that high-resolution descriptions be refinements of lower-resolution ones, i.e., every patch in a high-resolution description is spatially covered by exactly one patch in every lower resolution. A multi-resolution stack having this relationship among its layers is known as a hierarchical structure.

The multitude of data structures proposed for digitally representing surfaces may be classified by their approaches to the tasks identified above:

1. Area subdivision: the choice of scheme for subdividing the surface into sub areas.
2. Function set: the family of functions from which patch descriptions are chosen.
3. Hierarchy: its presence or absence.

The choices made by a modeling scheme in these categories are not independent of each other. Planar patches are compatible with triangles, for instance, whereas 2-D splines work better with quadrilaterals. These choices also determine the suitability of the DTM to regular or irregular sample data.

1.2.1 Area Subdivision

Area subdivision is a scheme for generating a finite set of mutually exclusive sub-areas which collectively cover the surface being modeled. To simplify their description, such schemes often decompose the relevant part of the x-y plane and the resulting subdivision is then projected onto the curved surface. The term "subdivision" refers to both the planar subdivision and the induced subdivision of the surface when no ambiguity arises.

This induced subdivision of the x-y plane has the property that its edges do not intersect (other than at vertices). This is a result of the fact that the surface being modeled is $2\frac{1}{2}$-dimensional, i.e. that every parallel to the z-axis intersects the surface at most once. This is also true of any polyhedral model of the surface, if its tolerance is sufficiently small, and of the model's edges in particular. If the projections onto the x-y plane of two of these edges were to intersect, the z-axis parallel passing through the intersection point would be violating the posited $2\frac{1}{2}$-dimensionality of the model. In graph theory, graphs whose edges do not intersect are known as planar [Hara69]. The fact that area subdivisions are planar is relevant to the discussion of triangulations (Section 1.3).

One of the most important features of a subdivision scheme is its adaptability. An adaptive subdivision scheme will generate smaller sub-areas to cover the more rugged regions of the surface, while using larger patches to describe the parts of the surface where deviation from a linear fit is more moderate.[2] Adaptive subdivision schemes exhibit improved storage utilization when compared with non-adaptive schemes, whose sampling density is unaffected by the variability of surface elevation, and therefore may oversample in areas of slow change.

[2]This support of variable resolution is distinct from that of a hierarchy, in which descriptions of the same surface region with different resolutions are accommodated. For more details see Section 1.2.3.

Moreover, when a non-adaptive scheme is employed, a cap on the resolution must be fixed before construction can begin, while adaptive schemes allow resolution to increase locally during the construction process if variation in surface elevation mandates it.

Unfortunately, subdivision schemes successful in decomposing the plane for other purposes, e.g. area maps, encounter difficulties when applied to $2\frac{1}{2}$-D surfaces. In particular, schemes often achieve adaptability by allowing sub-areas of different sizes to freely border each other. For example, consider Figure 1.1 which shows a part of a quadtree (a scheme in which each square can be subdivided into four squares of half its side length [Same90a, Same90b]). On the boundary between two nodes of different size, at least one vertex, say B, of the smaller node is not a vertex of its larger neighbor. Since elevation data is stored only in vertices, the elevation of B in the larger node is assumed to be interpolated from the elevations of the other vertices in that node, namely A and C. On the other hand, B is a vertex in the smaller node, where elevation is stored. These two elevation values are associated with the same location but need not coincide, thereby causing a "crack", as shown in the figure.

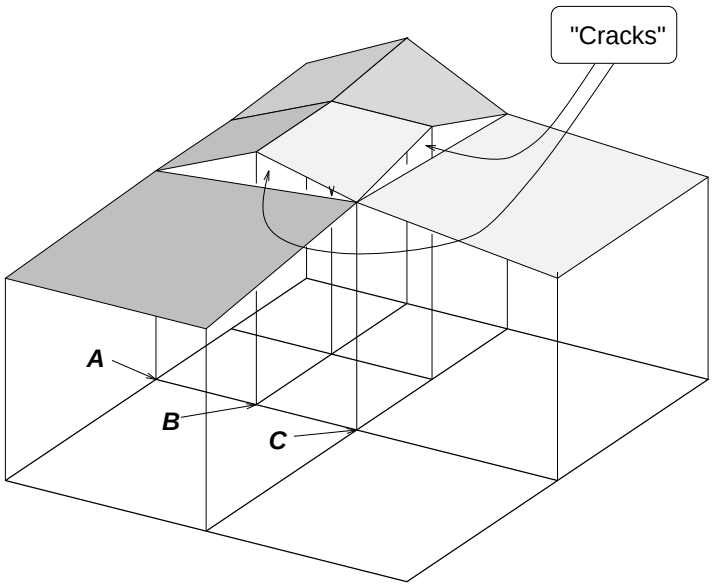

Figure 1.1: Surface discontinuity resulting from edges not meeting at their vertices.

Such cracks may form in places where polygons of the subdivision which share an edge do not share a vertex of that edge. We call such vertices non-shared vertices. Non-shared vertices are common in adaptive decompositions, in which the sizes of the subdivision polygons are determined locally by the ruggedness of the surface being modeled (Section 1.2.1). In the course of constructing a model for a given surface, there could clearly be a case where refinement is mandated for one polygon but not for its neighbor. This may introduce a non-shared vertex on their common edge, one which belongs only to the decomposed polygon, as in Figure 1.2.

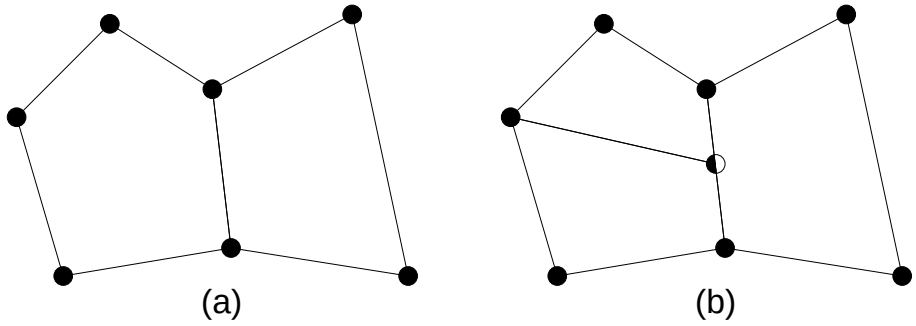

Figure 1.2: Conditions under which cracks are likely. (a) Adjacent polygons in a subdivision. (b) Only the left polygon is decomposed, introducing a non-shared vertex on the boundary between the polygons. A crack of the kind depicted in Figure 1.1 can now form.

1.2.2 Function Set

The simplest patch approximation is a plane: each patch is covered by a polygon which can be embedded in a plane. The polygon's shape, 3-D position, and orientation are designed to approximate the actual surface as known through its sampling. The resulting model is a polyhedron if care is taken to insure continuity along the boundaries between polygons. Most of the work done on DTMs, including this dissertation, deal with continuous polyhedral approximations.

Non-planar patch approximation functions are more versatile in that they can follow the curves of a meandering surface to a certain extent. Such patches, called splines, are described by a mathematical function, often a polynomial. The parameters of a particular patch are set so that the values it takes at the sample points agree with the measured elevations, and its derivatives at these points are equal to the derivatives of all other neighboring patches. Thus both the surface and its derivatives are continuous throughout the whole described area, yielding a smooth representation. Such models may be more realistic in some applications where the creases formed between the planar faces of a polyhedral representation are unacceptable. It is also conceivable that a non-planar patch can, on average, explain a larger piece of the terrain, thereby reducing the total number of patches required for a complete model, and possibly offset the added complexity such patches introduce.

A third category of functions attempts to simulate the surface rather than to faithfully describe it. Fractals have been used to generate pseudo-panoramas of terrain with stunning realism [Four82, Herb84]. By studying a surface and extracting its fractal parameters, one could fill the gaps between sampled points with generated data which may have no relation to the actual surface but nonetheless retains its texture and appearance, which for some applications, such as display, may be more important than the accuracy of the elevation values.

1.2.3 Hierarchy

Many applications using DTMs may require elevation information for a single region with different tolerances or resolutions at different stages of processing. For example, while rendering scenery in real time, a flight simulator needs good resolution for nearby objects but

can do with coarse descriptions of those farther away. Another example is surface intersection: judging by rough renditions of the surfaces, intersection can be precluded in all but a reduced number of areas, where a final determination is then made based on the higher resolution descriptions.

The desirability of a multiple-resolution surface model is prompted by the observation that most DTMs exhibit resolution vs. access time tradeoff: a low resolution surface description is often smaller in size and quicker in access than a high resolution one. Therefore, when an algorithm can either make do with a low resolution, or else requires the information quickly, the low resolution capability can be used effectively, resorting to the time-consuming high resolution only when necessary.

A DTM is said to be hierarchical if it can support multiple resolution descriptions of a surface. This amounts to more than having several replicas of the complete data structure, each built with a different tolerance. From the examples above it can be seen that an important feature of a hierarchy is the ability to navigate through it, quickly moving among the different-resolution descriptions of the same part of the surface. Approaching a location on the flight simulator's screen, increasingly higher resolutions of a decreasing portion of the surface are called for in rapid succession. The ability to cut vertically through the different resolutions is therefore central to the hierarchical DTM.

Such vertical navigation is simplified when the relationship between the different-resolution descriptions can be described by a tree, i.e., each subarea at any level of resolution (other than the lowest) is completely contained in single subarea of its predecessor, the next lower resolution description. In this case, there is a simple fanout of subareas when moving from lower to higher resolutions, which can be directly encoded into the data structure. There are hierarchies, such as the Delaunay Pyramid [DeFl89], where this is not the case and their suggested implementation is correspondingly more complex.

Some algorithms, when calling for lower-resolution information, are not as demanding as is the flight simulator. For surface intersection, for instance, bounding-box information is sufficient to preclude intersection when the boxes are disjoint. To produce the horizon as seen by an observer on the surface, after closer areas have been scanned and an initial skyline formed, maximum and minimum elevations in a subarea are sufficient to decide whether that subarea can possibly be seen (see Chapter 6). Thus, the hierarchical requirements of these and other algorithms can be accommodated with a less than complete low-resolution description of the surface. As will be seen later, many DTMs can naturally store such summary information, and will be labeled "partially hierarchical".

1.3 Triangulations

An area subdivision whose internal faces are all triangles is known as a triangu- lation. Due to reasons listed below, triangles and triangulations have a special role in surface modeling. Section 1.3.1 explains how triangulations appear naturally in polyhedral models. Section 1.3.2 provides additional incentives for using triangles. Section 1.3.3 defines the Triangulated Irregular Network (TIN), which is a common structure used with irregular data, including one presented here. Section 1.3.4 describes the major schemes by which adaptability and hierarchy are introduced in triangle-based models.

1.3.1 More about Area Subdivisions as Planar Graphs

As was shown above, area subdivisions induced by models of $2\frac{1}{2}$-dimensional surfaces, when regarded as graphs, are planar (Section 1.2.1). Of particular interest are maximal planar graphs, which are planar graphs to which no edge may be added without violating their planarity. It is easy to see that all the faces[3] of a maximal planar graphs are triangular. If this were not so, and a maximal graph had a non-triangular face, that face would have at least one diagonal. That diagonal, although clearly not an edge of the graph, could be added without violating the graph's planarity, in contradiction to the assumption that the graph is maximal.

Recall that one of the pitfalls area subdivisions should avoid is non-shared vertices (Section 1.2.1), which are locations where cracks in the model may form. Area subdivisions that are also maximal planar graphs are remarkable in that they have no non-shared vertices. To see that, assume the contrary: let vertex V of a maximal planar graph (see Figure 1.3) be non-shared, incident on polygon P of the figure but not belonging to it. V must be somewhere in the middle of an edge of P, or else it would itself be a vertex of P. The endpoints of the edge V is on constitute two of P's vertices. However, P must have at least three vertices (since a triangle is the simplest polygon), so there exists a vertex of P, say W, that does not share an edge with V. The edge $\overline{VW}$ can be added to the graph without violating its planarity, again in contradiction to the assumption the the graph was maximal.

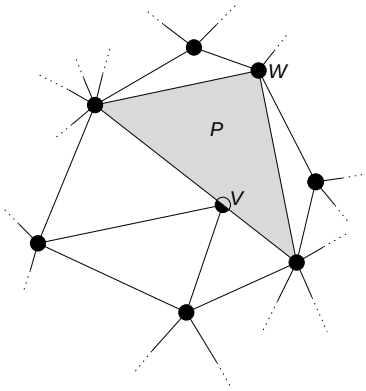

Figure 1.3: Proof that maximal planar graphs have no non-shared vertices. Each vertex is marked by a circle; the portion of the circle included in polygons that contain the vertex is filled. A non shared vertex, then, is denoted by a circle that is not totally solid.

It is clear, then, that one way to avoid non-shared vertices in a subdivision is to use only those that are also maximal planar graphs, which are, in particular, triangulations. This is how triangles arise naturally in surface modeling.

[3]To be precise, the statement is true of all internal faces. Sometimes the face made up of the outermost edges of the graph (and including the infinitely distant boundary of the plane) is also considered a face of the graph, called the external face. A graph has only one external face, and it is not necessarily triangular even in a maximal planar graph.

1.3.2 Triangles Are Advantageous

Triangulations are appealing also for other reasons. Any three distinct non-collinear points define a plane, and also define a non-degenerate triangle. There is a correspondence, therefore, between planes and non-degenerate triangles that more complex polygons do not enjoy. If the faces of a polyhedral DTM, in which each surface patch is approximated by a flat (planar) face, are constrained to be triangular, face planarity is automatic. Polyhedral DTMs allowing non-triangular faces may require a test for each face in order to guarantee its flatness.

Triangles are also always convex, an attribute which simplifies the task of finding the face of a DTM which contains a given query point, for example. Moreover, when the DTM is used for display purposes, many shading algorithms (e.g., Gouraud shading [Gour71]) exhibit discontinuities when applied to non-convex polygons.

As a result, most polyhedral DTMs, including both schemes presented here, use triangles as their basic building blocks.

1.3.3 Triangulated Irregular Networks

When irregular sample data is triangulated as described in Section 1.3.2, the result is termed a Triangulated Irregular Network or TIN [Peuc75]. Most irregular models either consist of a TIN implementation or use a TIN as part of a larger structure, including the irregular model described in this dissertation. TINs are appealing because they have the potential of minimizing the number of vertices required to represent a given surface with a given accuracy. This is because there are no external constraints on the placement of vertices in a TIN, so they can be placed where their informational content is maximized.

The issue of choosing the vertices for the triangulation in a TIN is, therefore, a central one. It can determine the surface locations where elevation should be sampled. More frequently, though, the process involves selecting a subset of the sample points in a given dataset such that the elevations at the remaining points may be regenerated with sufficient accuracy. As it turns out, points coinciding with dominant surface features, such as local extrema and flexing points, tend to be picked as representative of the surface by many algorithms.

Another issue is how to triangulate the vertices, once they have been chosen. Arbitrary triangulations are in general undesirable because they tend to result in triangles which are long and narrow, sometimes called slivers. When interpolating the elevation at a point in the interior of a sliver triangle, the data stored at its vertices is used, when there are likely to be closer vertices that are nonetheless ignored, as illustrated in Figure 1.4.

To avoid the formation of slivers as much as possible, a triangulation known as a Delaunay Triangulation is often used [Prep85]. This extensively studied geometrical construction has been shown to produce triangulations whose smallest angle is maximized, thus making sure that all its triangles are as equilateral (as opposed to long and thin) as possible. Delaunay triangulations have other attractive properties, such as that their triangles can be sorted for visibility from any point, a fact which is essential for some field-of-view algorithms (Section 6.4).

One problem with the Delaunay triangulation is that it is a 2-D structure, dealing not with the triangular facets of a polyhedral surface approximation but with the projections of these

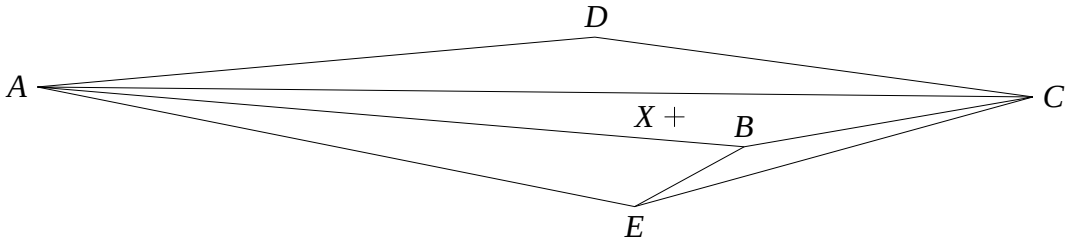

Figure 1.4: Example of the difficulty slivers present in interpolation. The elevation at point X is estimated based on the values known at points A, B and C, while points D and E, which are much closer to X, are ignored.

facets on the xy plane. It is becoming apparent that equiangularity of those projections does not guarantee optimality of the surface approximation, because of edges of the triangulation which may be drawn contrary to trends on the surface being modeled. As mentioned above, TINs are often based on points which represent locations where the surface trends change direction, such as peaks, pits and saddles. The edges of the subsequent triangulation can then be made to follow the linear features of the surface, such as valleys and ridges, a property shown to be advantageous [Scar92, Poli92]. These considerations are anchored in the 3-D data and are lost when projected on the plane to be Delaunay triangulated.

1.3.4 Triangular Decompositions

As was shown in Sections 1.2.1 and 1.2.3, the ability of a DTM to have its faces decompose in a regular fashion is helpful in supporting the adaptability and hierarchical properties of the model. Triangles can conveniently be decomposed in one of two ways, known as ternary and quaternary. Ternary decomposition splits a triangle into three smaller ones by connecting an interior point with the vertices of the original triangle, as in Figure 1.5a. Quaternary decomposition produces four descendants by connecting points on each edge, as seen in Figure 1.5b.

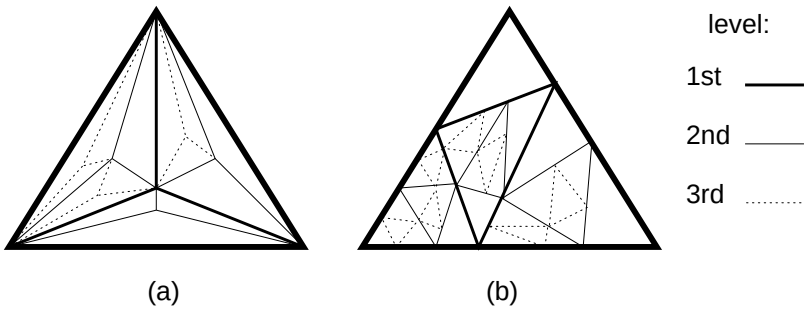

Figure 1.5: Examples of triangle decomposition: (a) ternary decomposition; (b) quaternary decomposition.

In the ternary scheme, the angles of the original triangle are subdivided at each decomposition step, very quickly producing triangles with very acute angles, the hallmark of the undesired slivers. The quaternary approach does not exhibit this problem. In fact, if points on the edges are chosen so that they split the edges in a fixed ratio (such as the edge

midpoints), all triangles will be similar to the original one, differing only in size. However, quaternary decomposition can cause edges to split unilaterally, i.e. triangles on both sides of an edge are not synchronized to split together, thus creating non-shared vertices which may cause surface discontinuities or "cracks" (see Section 1.2.1).

Chapter 2

Previous Work

Most of the work surveyed here deals, as expected, with polyhedral surface models. Section 2.1 describes regular polyhedral models, while Section 2.2 deals with irregular models. Some attention is given to models which are non-polyhedral in Section 2.3.

2.1 Regular Polyhedral Models

2.1.1 Data Compression — [Barr87]

A simple quadtree-based DTM whose main purpose in data compression is presented in [Barr87]. It assumes regular sampling of a square plot having $2^m + 1$ samples on a side for some integer m. A method of scanning the data and choosing values for storage in a quadtree is described. Care is taken to insure that values selected to be skipped can still be retrieved (within a given tolerance) from the data structure. Compression ratios depend on the shape of the terrain modeled. On their test data, the authors were able to demonstrate over 50% reduction even with a tight tolerance.

The input is scanned several times, with each pass taking a larger subsample of the data. In the first pass, only nine elevations are considered: the extreme corners, the side midpoints, and the center of the area being modeled. The resolution is doubled with each subsequent iteration, until, after m steps, all data values have been considered.

Each input elevation in turn is compared with the average of its ancestors, which are some of the locations closest to it in the previous, sparser iteration. The current value is stored in the quadtree being constructed only if it differs significantly (by more than the predetermined tolerance) from that average. The authors suggest several ancestry schemes which differ in the numbers of points that are considered ancestors of a given value and in the way their average is calculated. Empirically, the choice of scheme seems to have only a small impact on the final outcome.

As implemented, the elevation values are stored sequentially in a flat file, similar to the way the input is organized. However, since some of the elevation values have been eliminated, it is no longer possible to tell the (x, y) location of a value from its placement in the file. A quadtree is employed to keep track of these locations. Implemented with pointers, the quadtree gray nodes are available for storing summary information such as the number of non-discarded points and the maximum and minimum elevations within the area spanned by

the node. Also, since the cost of storing a point in the file as well as the overhead incurred if it is removed are predictable, a data point is physically discarded only when eliminating it will result in an overall reduction of storage space. Otherwise, even interpolatable points are retained.

During construction, the input data is accessed in an order which is unlikely to correspond to the way in which the data is stored. Since this DTM is geared mostly towards data compression, construction time may be of lesser importance if the resulting compressed data is used many times. However, little is provided in the form of a mechanism for efficient retrieval.

2.1.2 Triangular Bintrees — [VonH89]

The triangular bintree ([VonH89]) is a compact data structure for triangulation of regular sample data. The structure can be implemented as binary tree, hence its name.

Initially, the square plot is divided by one of its diagonals into a pair of right triangles. The diagonal serves as the hypotenuse of both triangles. Triangle pairs can be subdivided by splitting their shared hypotenuse, forming two pairs of triangles which are similar to the original ones, i.e. they are isosceles right triangles. Note that with the exception of triangles on the exterior boundary of the area of interest, the triangles of the new pairs share their hypotenuses as well, so the process may continue recursively. Boundary triangles may be split individually. See Figure 2.1.

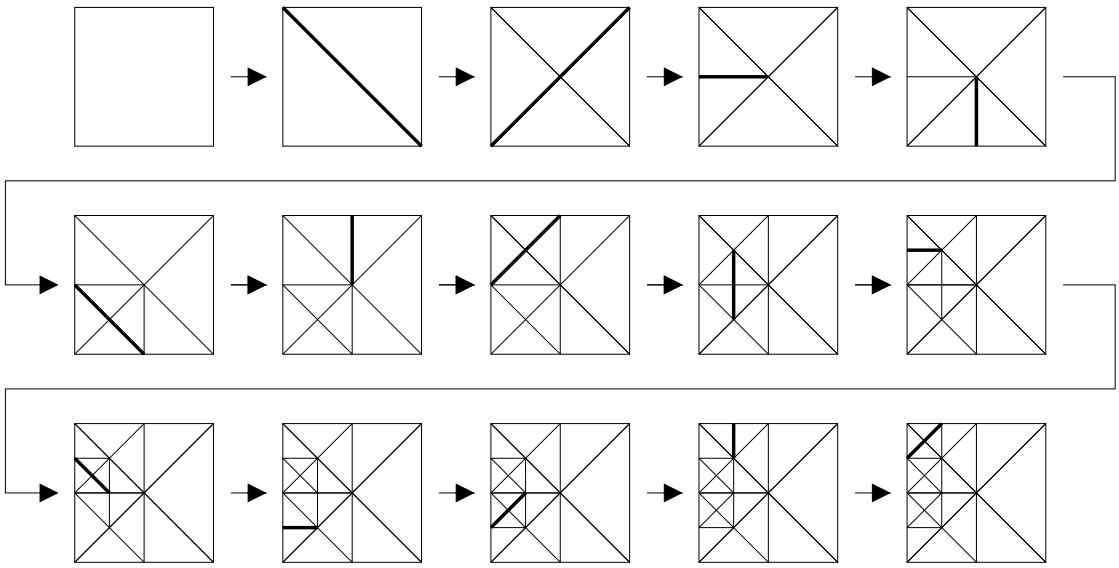

Figure 2.1: Triangular bintree: example of construction. At each step, the last edge to be added is highlighted.

The restriction that only pairs of triangles having their hypotenuses in common may split gives the splitting of a triangle the potential to cascade, bringing about further splits. If the hypotenuse of the triangle targeted for splitting coincides with another triangle's leg, the smaller triangle cannot be split before the larger triangle is split. The same may be true for

the larger triangle as well, so the split may cascade recursively. Termination is guaranteed, however, since the triangles involved increase in size by $\sqrt{2}$ at each step, requiring only a finite number to reach the size of one of the two initial triangles, which can always be split unconditionally. For an example, see Figure 2.2.

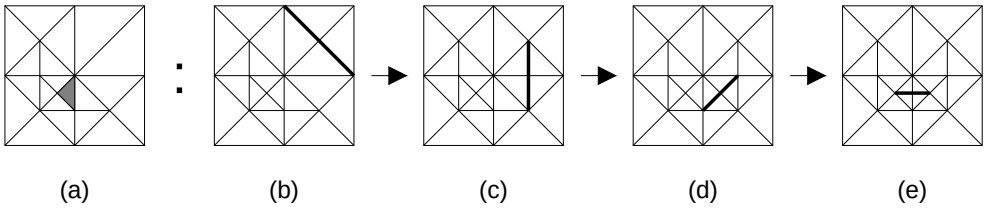

Figure 2.2: Cascading splits in a triangular bintree. (a) A triangular bintree with the next triangle targeted for splitting marked (solid gray). (b)–(d) Steps in preparing the desired split. (e) Finally, splitting the desired triangle.

The above scheme tiles the xy plane with isosceles right triangles, which are about as good as equilateral triangles in terms of the absence of slivers. Also, each triangle is in one of two possible orientations: either its legs or its hypotenuse are parallel to the axes. This property in turn simplifies the procedure for finding which triangle contains a given point. Splitting triangles in pairs also eliminates the discontinuities that plague other schemes which allow edges to meet at points other than their vertices.

To construct a surface model, the elevations at the vertices of the first triangle pair are determined, assigning an interpolated elevation value to each point in the area described. If any of these values deviates significantly from the elevation specified in the input, the triangle in which this deviation occurs is split. This process continues until the interpolated values at all locations are sufficiently close to those indicated in the input.

When implemented as a binary tree, every node corresponds to a triangle. If and when that triangle is split, the leaf node representing it is given a pair of children, each corresponding to one of the smaller triangles the original one was split into, as is Figure 2.3. If information such as the minimum and maximum elevation in each triangle is maintained, the internal nodes can retain their previous values when they are split, providing a hierarchical quality to the data structure.

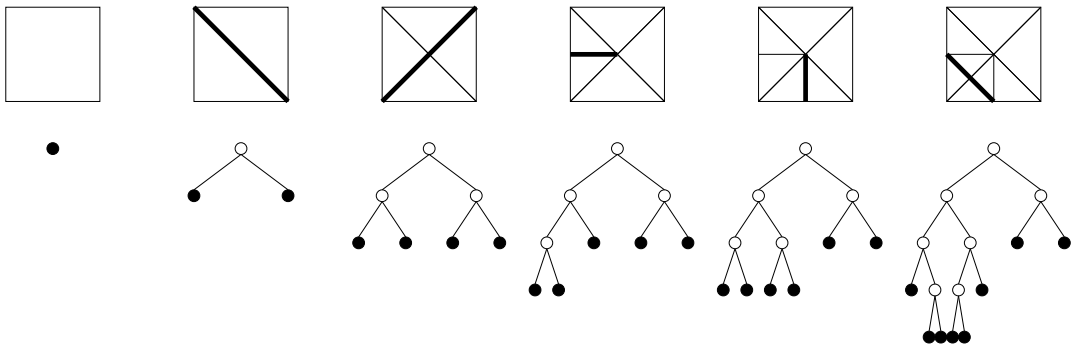

Figure 2.3: Triangular bintree correspondence with a binary tree.

2.1.3 Planetary Relief — [Dutt84, Feke90]

So far, only DTMs for essentially flat terrain have been discussed. A scheme ambitious enough to encompass the whole globe must be able to model a sphere rather than a plane, where elevations are measured along the sphere's radius rather than along the z axis. Several such systems have been proposed, permitting the accumulation and integration of elevation data for many areas on the globe in a single data structure.

One such model, called the Geodesic Elevation Model or GEM for short, is presented in [Dutt84]. The model starts out with an octahedron, with two of its vertices coinciding with the poles of the globe and the other four vertices located on the globe's equator. The triangular faces are then subdivided into smaller triangles in a regular fashion.

At each level, the triangular faces are decomposed into smaller triangles whose areas are one third of that of the triangle being decomposed. Statistically, then, the number of triangles increases threefold at each decomposition step. The details are a bit involved, however, because first-generation descendants extend into neighboring faces. Instead of decomposing a face into three triangles, as would be the case in a strict ternary hierarchy, the area of a face is shared among six child triangles. The face covers only half the area of each child triangle, the other half being covered by a neighbor of the original face. See Figure 2.4b.

However, after two decomposition steps, the grandchild triangles obey a strict hierarchy relationship with their grandparents: nine triangles cover their grandparent exactly. This relationship is represented in two nonary trees, one for the odd levels of the hierarchy and one for the even ones. To track the hierarchy, the two trees need to be visited alternatingly.

The subdivision provides a natural way to encode the faces: assigning a number to each descendant (between 1 and 9, for instance), the sequence of faces visited on the way to the desired location constitutes a geocode which uniquely identifies it. On the face of the Earth, eight or nine digits are sufficient to specify areas smaller than most postal zones and census tracts.

Elevation is encoded in this model by using only one bit per face. If this bit is clear it is an indication that the associated face has the same elevation as its parent. Conversely, setting the bit denotes that the elevation of the face deviates from that of its parent. Whether it is higher or lower depends on the tree it is in. If the face is part of the odd-levels tree (the one having the initial octahedron at its root) it is assumed to be higher than its parent when its elevation bit is set. Faces in the other tree, the one with the even levels, are assumed to be lower than their parents. The amount by which they are higher or lower depends on their level within the hierarchy: it decreases (by $\sqrt{3}$) from level to level. Such a scheme has a limit on the maximum deviation from sphericity which can be expressed, but it is certainly adequate for the Earth: while the highest mountain peak is less than 9 kilometers above sea level, whereas elevations as great as 500 kilometers can be expressed using this scheme.

The scheme has the advantage shared by scientific number notation: The depth of the tree (like the number of digits in a decimal fraction) indicates both the value conveyed and its accuracy. A datum inserted into a GEM database will trickle down the trees only to a depth commensurate with its accuracy. If the database is initialized with possibly coarse but accurate data covering the complete sphere, subsequent attempts to load erroneous data

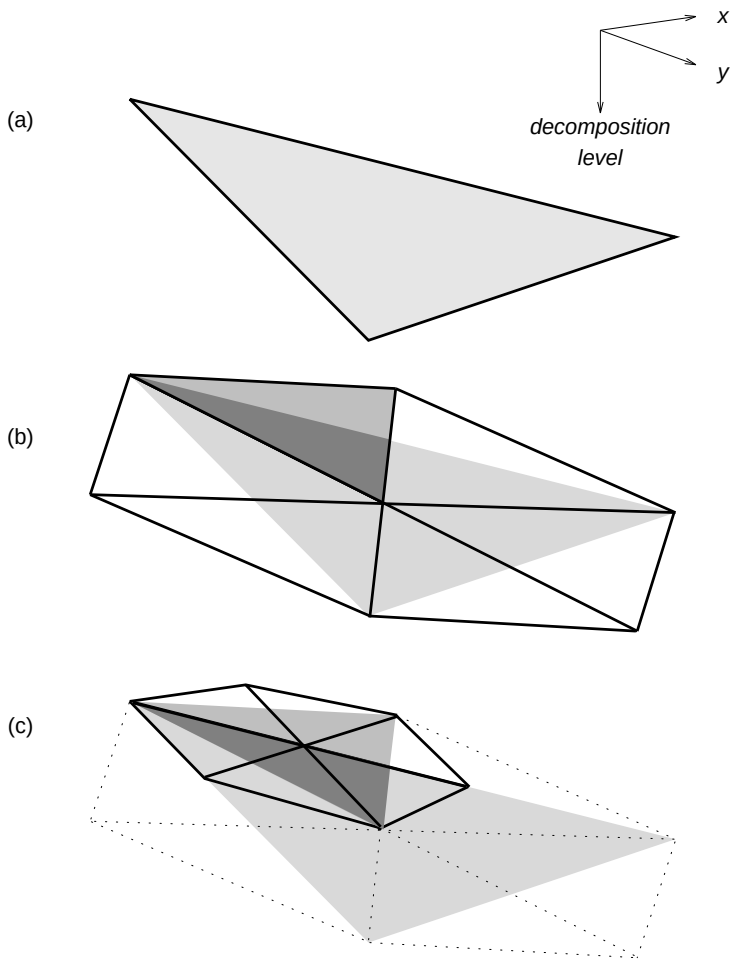

Figure 2.4: Decomposition in the Geodesic Elevation Model. (a) A triangular face. (b) First-generation descendants: six triangles, each shared equally with one of the neighbors of the original triangle. (c) Second-generation descendants: the children of only one face (the one highlighted in (b)) are shown. Three of these are completely contained inside their grandparent. Other faces in (b) contribute additional triangles, totaling 9 which cover the total area of the original face.

can be automatically flagged as suspicious by the database.

This model has a few problems as well. The shapes of faces projected onto the sphere are not fixed, but depend on their location relative to the original octahedron. Another problem has to do with systematic deviations from sphericity that planets, Earth included, often have. [Dutt84] suggests that a global framework, representing the first few levels of a hierarchy, be agreed upon by all users. This framework can represent the deviation of the globe from perfect sphericity sufficiently well. However, such a scheme depends on an uncommon level of cooperation among software developers, agencies and users.

Another attempt to describe a sphere, this time with a quadtree, is made in [Feke90]. The initial shape is an icosahedron, the platonic solid with the greatest number of faces. Each of the icosahedron's twenty triangular faces is subdivided into four triangular subfaces

by connecting the edges' midpoints (quaternary subdivision).

Having four descendants makes quadtrees appropriate for the representation of this structure. A complete representation consists of a forest of twenty quadtrees, each stemming from one face of the original icosahedron. Faces, or trixels, are labeled within a quadtree in a similar way to that described in [Dutt84], but only four possible values are needed rather than nine. Naming vertices is a bit more involved, since vertices are shared among six trixels (except for the twelve vertices of the original icosahedron, which are shared only by five). For uniqueness, the lexicographically smallest among the five or six synonyms is selected.

For the quadtree representation, a host of established algorithms can be adopted to find neighbors, connected components, and other useful properties. The naming conventions make it possible to determine adjacency by examining only the names of candidate trixels. Although a quadtree describes only a single icosahedron face, extending the algorithms to the whole sphere is not very difficult.

2.1.4 Semi-regular Model — [Gome79]

In one of the early attempts to formulate a DTM [Gome79], the hope was expressed that if an irregular dataset is sufficiently dense, it should support a regular decomposition. The attempt was based on recursive quaternary decomposition of triangles.

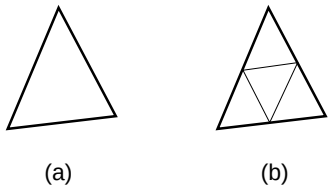

Figure 2.5: Traditional quaternary decomposition. (a) A triangle. (b) The same triangle decomposed into four smaller triangles formed by connecting the midpoints of the edges.

Traditionally, quaternary decomposition of a triangle is achieved by connecting the midpoints of its three edges, as shown in Figure 2.5. The new vertices introduced in the process (the edges' midpoints) are not likely to be related to the input data. [Gome79] suggests using actual data points which fall close to the desired edge midpoints instead (see Figure 2.6). Thus the model is topologically isomorphic to a quaternary model but is not quite identical with it. As a result, there could be parts on the outskirts of the surface being modeled that may remain unexpressed due to errors resulting from this deviations in vertex positioning. Solutions to this and other problems raised by the model are offered, but notably no solution is offered to the issue of crack formation, which is assumed not to be serious [Gome79]. Another difficulty is the fact that no two of the triangular faces are alike. In applications where these problems are not important, this approach has the combined appeal of both regular and irregular models.

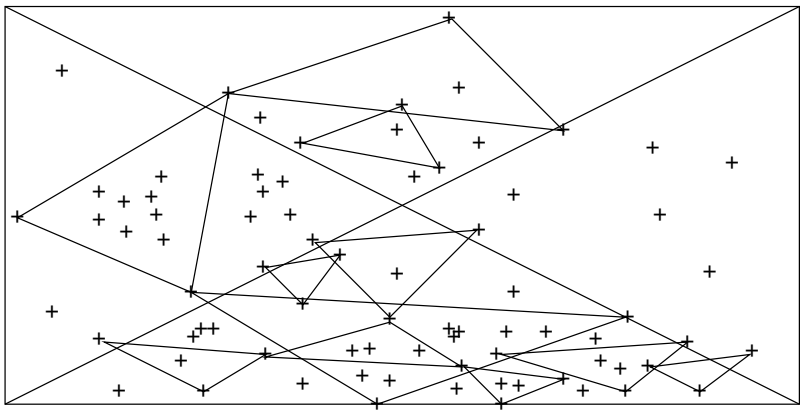

Figure 2.6: Semi-regular quaternary decomposition derived from an irregular dataset.

2.2 Irregular Polyhedral Models

2.2.1 Triangulated Irregular Networks (TINs) — [DeFl84, DeFl85]

A ternary adaptable TIN is described in [DeFl84, DeFl85]. The construction and a few representative applications are studied and the results are compared with Delaunay triangulation of the same dataset.

The input to this model is assumed to be an irregular sample of the surface to be modeled. The purpose of the model is to select points from this dataset and triangulate them so that the error associated with the induced polyhedron is smaller than some predefined value. The error is measured as the maximum distance between a face of the polyhedron and the actual surface. Since the surface is known only through the dataset, the error can be measured only at locations for which data is provided in the input.

The algorithm presented assumes that the given dataset is triangulated. Any triangulation of the convex hull of the dataset is acceptable. The triangles of that triangulation are then sorted according to their error value, and the one with the highest error is processed first. The point in its interior with the greatest error is found and a ternary decomposition is induced by connecting that point to the triangle's vertices. The new triangle set is resorted and the above step is repeated. The process iterates until all triangles have errors within a given tolerance. Several ways to minimize the cost of the sorting phase, either by means of a heap or by threading the triangles in order of decreasing error values, are presented. It is shown that if the initial set contains n points, of which m ($m \leq n$) are ultimately selected to represent the rest, then the construction of the model takes $O(mn)$ time in the worst case and $O(m \log n)$ time on the average.

Three deficiencies of this model are pointed out by the authors:

1. There is no guarantee that the smallest number of points which can support the given tolerance will actually be selected. In the course of refining the initial triangulation, the algorithm only adds points, and never deletes any. It is possible that the initial triangulation already contains inefficient points that would not be included in an optimal triangulation. By refraining from removing any points, it is impossible for the

algorithm to transform such initial triangulations into optimal ones. It is conjectured that finding the optimal model is NP-complete, since it seems that all $\binom{n}{m}$ selections for all $1 \leq m \leq n$ must be examined in order to find it.

2. A small change in the underlying data does not necessarily translate into a minor update of the model stemming from this data. A change in a single input value may necessitate reconstruction of the model from scratch, making incremental changes difficult to introduce. This is not too serious a handicap where terrain is considered, since terrain seldom changes, but it could be a problem for other applications of DTMs.

3. There is no guarantee that the tree describing the model is balanced; it is possible, in the worst case, that only one descendant at any level ever gets subdivided. As a result, search time can increase to $O(m)$. In the average case, however, it is expected to be $O(\log m)$.

The difficulties caused by the many long, thin triangles this model is likely to contain are only hinted at, when it is mentioned that this triangulation is inferior to Delaunay triangulation if contour lines are desired. It is suggested that the selected point set be retriangulated by a Delaunay algorithm for that purpose when the need arises [DeFl84].

Most of the tests of this model have been based on data generated by randomly sampling surfaces representing analytical functions, such as a sphere. Only one actual terrain sample is used in [DeFl85].

2.2.2 Hierarchical TINs — [DeFl89, DeFl92b]

To alleviate some of the problems raised in the ternary TIN described in Section 2.2.1, the authors suggest a TIN based on Delaunay triangulations, which they term a Delaunay pyramid.

Delaunay triangulations do not lend themselves to hierarchical structures. This is because the Delaunay triangulation of a given set of points is not necessarily a refinement of the Delaunay triangulation of any subset of that set. If a given point set is Delaunay triangulated, then as additional points are added and the augmented set is retriangulated, some edges will be added but some may also be removed, as seen in Figure 2.7. Consequently, finer meshes which represent higher accuracies can no longer be produced from coarser ones just by decomposing triangles individually, as was possible in the scheme described in Section 2.2.1.

However, the fact that the Delaunay triangulation of a refined point set is not necessarily a refinement of the original point set does not mean that the two triangulations are unrelated. Since the edges of a Delaunay triangulation are determined based on local criteria, the effect of adding a point to a point set and retriangulating it is confined to a vicinity of the new point. It is possible to identify a polygon of influence of the added point, a region to which the changes induced by the added point are confined. On average, a polygon of influence consists of only six triangles [Sibs78]. The sets of triangles covered by this polygon in the original and new triangulations have a many-to-many relationship, but these "many"s are bounded and in fact are quite small.

It is practical, then, as suggested in [DeFl89], to maintain bidirectional pointers between triangles in different triangulations which have a non-empty intersection. Through these

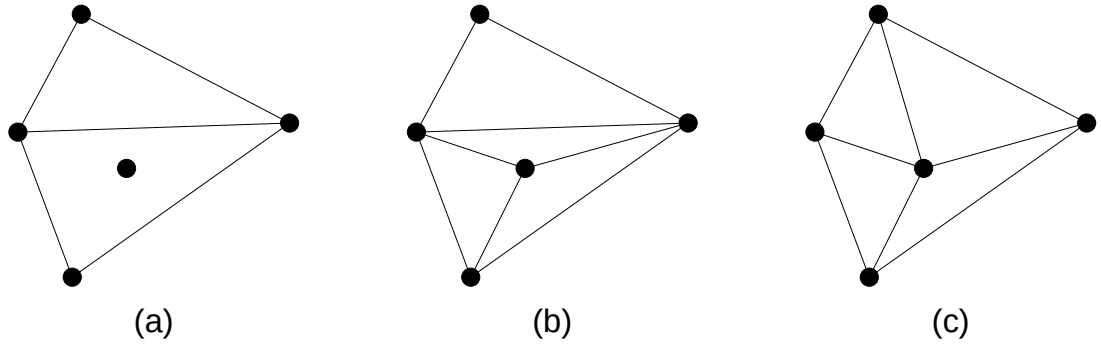

Figure 2.7: A Delaunay triangulation of a superset of points is not necessarily a refinement of the triangulation of the original point set. (a) A Delaunay triangulation of a four-point set to which a fifth point is added. (b) A refinement of the triangulation in (a) which incorporates the extra point. (c) A Delaunay triangulation of the five-point set. Note that it is not a refinement of the triangulation in (a).

pointers, algorithms can move between triangulations that represent different resolutions of the same part of the surface. As a result, the complete structure becomes a true hierarchy in the sense of Section 1.2.3. Regarding these pointers as directed, leading from the root to the leaves, the tree of the ternary TIN can be seen as being replaced here by a directed acyclic graph (DAG).

A construction algorithm very similar to the one suggested for the ternary TIN can now be employed. Starting with some initial choice of points from the input, their Delaunay triangulation forms the root of the hierarchy. Of the points not yet selected, the one associated with the largest error is found and added to the existing point set, which is then retriangulated. Normally, most triangles are unaffected by this addition, so their occurrences in the old and new triangulations are simply linked. Multiple links are established between the affected triangles in the old triangulation and those replacing them in the new one. This step is repeated until all points are used or the error of the triangulation as a whole drops below a predefined tolerance value.

The most basic function of a DTM is to predict the elevation of the modeled surface at a location for which no input value is specified. In the present model, however, this operation is not as straightforward. As with TINs (Section 2.2.1), the hierarchy needs to be traversed until a triangle containing the query point with a small enough error is found, where the elevation at the point can be interpolated. However, since now the trace is of a DAG, there could be levels between which there is no fanout, so no progress is made. Moreover, even when a triangle on one level is linked with more than one triangle on the next lower level, there is no guarantee that these lower-level triangles are any smaller, again limiting the possibility of refinement. In the worst case, it is possible for a hierarchy to be $O(n)$ deep, where n is the number of points. This must be compared with the somewhat similar K-Structure [Kirk83] which is guaranteed to be no deeper than $O(\log n)$.

On the other hand, the Delaunay pyramid is a hierarchical surface model in the sense described in Section 1.2.3: it accommodates several descriptions of the same surface with different degrees of accuracy. Moreover, each of these descriptions is a Delaunay triangulation, which has many advantageous properties (Section 1.3.3). It also allows an application

to store in higher levels summary data about lower levels, often allowing for faster searches.

An improvement on the Delaunay pyramid, the Hierarchical Delaunay TIN, is presented in [DeFl92b]. A strict hierarchy is imposed on the Delaunay triangulations by shifting the focus from vertices to triangles. Here, at each construction step, a triangle is selected for decomposition, rather than a point from the input being chosen for insertion. The edges of the selected triangle are decomposed first by adding vertices at positions where the differences between the measured and interpolated elevations are greatest. Once the boundary of a selected triangle is described with a better accuracy, its interior can be Delaunay triangulated with that accuracy without affecting any of the neighboring triangles. To avoid surface discontinuities, all the triangles whose edges are affected must be retriangulated. The fanout in this structure is indeterminate, and depends on the variability of the surface. The important aspect is, though, that edges are never removed when moving from a coarse triangulation to a refined one, so this hierarchy is again a tree rather than a DAG.

2.2.3 Cartographic Coherence — [Scar92]

As discussed in Section 1.3.3, there is no agreement on whether equiangularity of the faces is an overriding property of a good triangulation. In [Scar92] it is argued that for certain surface formations, triangles with very acute angles may be natural. Triangulations in which the edges do not disregard the linear features of the terrain, called cartographically coherent, may outperform those in which equiangularity is the sole consideration (e.g. Delaunay triangulations).

For example, consider Figure 2.8. If a ridge happens to be crossing the area covered by the triangle, elevation measurements are likely to be made along the summits, producing a concentration of points as seen in Figure 2.8a. If each of these points is added in turn using a ternary decomposition rule, as in Figure 2.8b, many more faces are formed than if the cartographic coherence approach is adopted, as in Figure 2.8c.

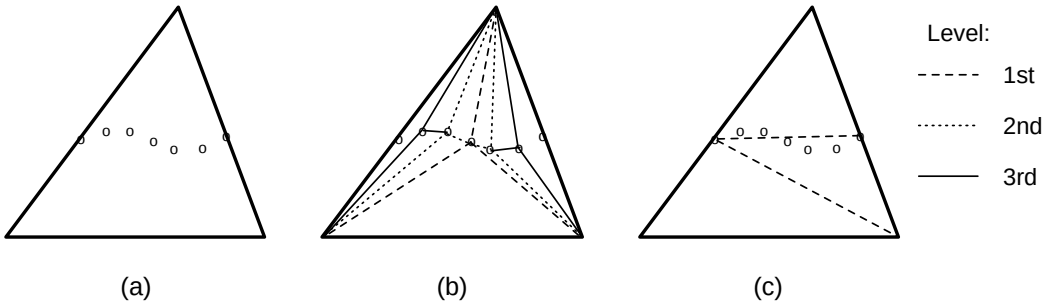

Figure 2.8: Comparison of triangulations with and without cartographic coherence (after [Scar92]). (a) A triangle and the data points in its interior; (b) a likely decomposition without cartographic coherence; (c) the proposed way to decompose the triangle.

As in the TIN models discussed earlier, a face in the triangulation is selected for decomposition based on the size of the error associated with it. Here, however, the decomposition is sensitive to the location of the point (or locations of the points) with large errors within the triangle. For each triangle, four points with maximal error are found: one with overall

maximum error and one on each edge. Depending on whether there is a peak within the triangle, and on how many edges have significant errors on them, one of five basic ways in which a triangle may be decomposed is selected. The rules and the conditions for choosing them are shown in Figure 2.9.

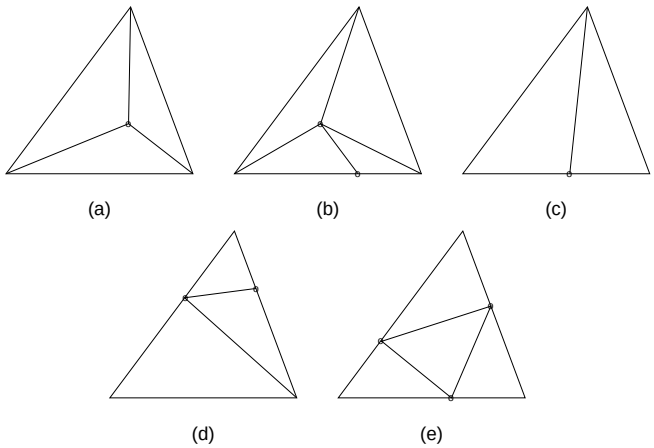

Figure 2.9: The five ways in which a triangle is split with cartographic coherence (after [Scar92]). (a) Split in center; (b) split on one edge with a significant center; (c) split on one edge without a significant center; (d) split on two edges; (e) split on all three edges.

In experiments using several small datasets, the performance of this approach was equal to and sometimes better than that of some Delaunay triangulation approaches in terms of number of faces, data compression ratio, and even absence of slivers.

2.3 Non-Polyhedral Models

2.3.1 Curved Surfaces — [Schm86]

The technique of adaptive subdivision employed for polyhedral surface approximation can also be applied when the patches of the proposed model are not planar. In [Schm86] a quadtree-like structure is described using curved patches to model a surface. Since such patches are not constrained to be planar, different-size patches may freely border each other with no danger of forming cracks. Consequently, quadtrees can be used freely in this environment.

This technique was developed to support an automatic data acquisition system which can sample the surface of an object placed in it at close to 30000 readings per second. Faced with such an abundance of data, the need for a data compression facility was great. However, a polyhedral approximation was rejected as inappropriate for modeling the smooth-surfaced, man-made objects the system was designed for.

The method uses bicubic splines patched together while keeping the surface continuous and differentiable (G^1 continuity). In the representation used, each patch is controlled by a 4×4 array of points whose position affects the shape of the patch. The patch interpolates (actually passes through) only the four points at the corners. Continuity between two patches

is assured if the positions of the control points on the boundary of one patch match those on the other. A set of additional constraints (which include the non-boundary points as well) is developed to guarantee the continuity also of the tangents at patch boundaries.

Like many other methods, processing starts out with a coarse approximation of the modeled surface; there is nothing to preclude the use of a single patch as an initial state. The algorithm iterates through all patches in the current model, accepts those which can be made to fit the data sufficiently well, and subdivides the rest. The task is complete when the distance between model and data points is within tolerance throughout the model, or when the resolution limits of the input data are reached.

To subdivide a patch, its area is split into four smaller patches, and the additional control points required to describe their 3-D shapes are computed. Imitating planar quadtrees, in which a square is subdivided into four squares by connecting the centers of its opposite sides, a patch is subdivided along lines passing through its center. The data point closest to the new vertex formed at the center of the patch is found and is used, along with the points at the four corners of the patch, as a basis for the control point arrays for the new patches. The data in the vicinity of the new vertex is tested to obtain the likely inclination of the tangent plane there, determining the directions the tangents along the new patch boundaries are supposed to follow. Once the new patches are established, they are compared to the input data and accepted if close enough, or subdivided if not.

The mathematics involved in the formulation of splines and their Bernstein-Bézier representation is very different from that of the other models discussed here, but the surface modeling principles are the same.

2.3.2 Fractals — [Four82]

As we have seen, one of the central motivations for DTMs is data compression. This is achieved by utilizing succinct mathematical forms to express shapes that when described numerically require a lot of space. So far, we have discussed deterministic models, in which the elevation at each point is either the result of a measurement made on the actual surface or an interpolation obtained from such measurements.

A completely different approach to surface modeling is presented in [Four82], based on the work of Mandelbrot [Mand68]. Instead of attempting to deterministically capture the exact shape of a surface, the surface is simulated by a stochastic process. Like planar polygons and curved splines, such processes can be completely described by a handful of parameters, and yet produce the elevations on a complete, continuous surface patch. In fact, just as the use of deterministic patches promises only an approximation of the true surface, so do stochastic renditions of it. However, the latter have several advantages that, for some applications, may be very attractive.

First, since elevation values between sample points are computed by a random process, new surface details can be generated at every magnification level, allowing indefinite zooms onto the surface. Since these details are fabricated, the simulation can be run up to the level required for display but no further. This means that when the surface is viewed from a distance, computation can be limited to producing sufficient detail to make the surface recognizable. For close-up views, more detail can be generated, but only in the area being

viewed (which due to the screen's physical size, cannot be large), again limiting the amount of computation needed. This situation contrasts with that of a deterministic model, which on the one hand cannot provide meaningful detail when a zoom-in exceeds its resolution, and on the other hand still has to sift through all the data, even when a zoom-out makes most of it irrelevant.

Second, such surface simulations have been shown to be capable of producing pleasing or even striking images of terrain [Herb84]. The visual test is not an unimportant one, since in many applications (e.g., flight simulators) the realism of the graphic display is of prime importance. Moreover, deterministic models are empirical, and do not represent theoretical understanding of terrain formation; they are therefore not necessarily more profound.

The basic algorithm is quite simple: the space between locations for which elevations are specified in the input is subdivided, and new elevation values are inserted. However, instead of using the average of the known elevations (as would be the case in a polyhedral model) or some polynomial function of them (as when splines are used), the value obtained by any of these methods is further perturbed by a random amount, controlled by a so-called fractional Brownian motion (fBm).

FBm is related to ordinary Brownian motion, which is the continuous counterpart of a random walk, by taking a moving average of the latter, weighted by a factor. The factor is dependent on a parameter, called the self-similarity of the function, which controls the width of the aperture through which the moving average is taken. Different surfaces have different self-similarities, and the best value for an application must be determined empirically.

Two properties crucial for a patch-generating function to be useful in piecewise surface modeling are the so-called internal consistency and external consistency [Four82]. Internal consistency assures that a patch can be regenerated consistently at different positions, orientations and magnifications (i.e., it is independent of the coordinate system and resolution). External consistency guarantees that adjacent patches agree on the elevations of points on their common boundary. In deterministic models, internal consistency is an automatic by-product of the patch-generating functions. External consistency is just a generalization of the need to avoid cracks (Section 1.2.2). Nondeterministic algorithms, however, must pay special attention to both these issues. No generally applicable approach is offered; each random generating function (such as the fBm) must be studied and a scheme to maintain internal and external consistencies must be individually tailored for it.

Fractals are by no means limited to modeling of surfaces; they are applicable to any objects, and even to time-varying phenomena, which are too complex to predict or describe numerically. Coastlines (in one dimension) and smoke (in three dimensions) are examples of non-surface objects to which fractals can be applied. Fractals can also be used to determine aspects other than surface elevation, such as color or motion. Fractals are probably the technique of choice in applications where realism and speed outweigh fidelity to some predefined shape.

Chapter 3

The Restricted Quadtree

3.1 Introduction

Quadtrees have been shown to be useful in many spatial applications [Same90b]. However, quadtree-like decompositions are not immune to non-shared vertices. (A non-shared vertex is one that does not belong to all the polygons it is incident on; see Section 1.2.1.) When subdividing an area for use in surface modeling, non-shared vertices are sites where cracks in the surface model may form.

A quadtree variant called the restricted quadtree or RQT, presented in [VonH89], adapts quadtree decomposition to surface modeling by restricting the number of potential non-shared vertices per edge. Each quadtree node is then triangulated, forming between four and eight triangles, depending on the number of non-shared vertices it actually possesses. The final result is a quadtree-like decomposition which employs only shared vertices. The RQT is formally defined in Section 3.2.

Various issues regarding the implementation of the RQT are dealt with in Section 3.3. Special attention is given to the construction of RQTs from raster data. Two algorithms are presented using the classic bottom-up and top-down paradigms. The details of their implementation, analysis of their performance, and experimental resulls are given in Section 3.4.

3.2 Definitions

3.2.1 Restricted Quadtree Definition

The two-dimensional restricted quadtree [VonH89] (also known as a 1-irregular mesh [Bank83] or balanced quadtree [Bern90]) is one in which the lengths of the sides of neighboring squares differ at most a factor of two. As an example, the decomposition depicted in Figure 3.1b represents the RQT corresponding to the non-restricted quadtree shown in part (a) of the figure. While the number of non-shared vertices per edge (denoted by half-solid circles) in a non-restricted quadtree is unlimited (as seen at point X in the figure), in an RQT there is at most one.

When implemented as a tree, this restricts neighboring leaf nodes to be at most one level apart, thereby moderating the depth variation in the tree. As a result, while a node in a non-restricted quadtree may be decomposed independently of all other nodes, splitting a

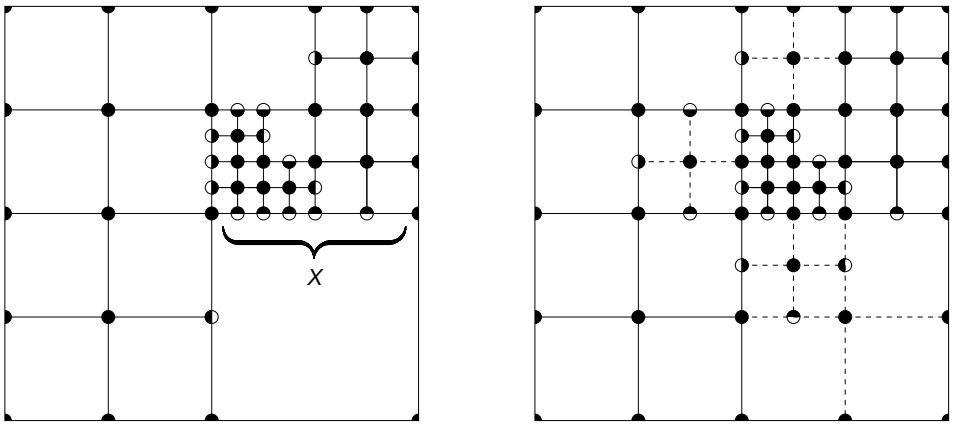

Figure 3.1: Comparing the restricted and non-restricted versions of a sample quadtree. Vertices are marked by circles; the portion of the circle included in nodes that contain the vertex is filled. A non-shared vertex, then, is denoted by a circle that is not totally solid. (a) A non-restricted quadtree. Note that the number of non-shared vertices (half filled circles) per edge is unbounded (e.g., at X). (b) the restricted version of the quadtree. Edges that were added to make the quadtree restricted are marked by dashed lines. There is at most one non-shared vertex per edge in an RQT.

node in an RQT can cascade, affecting neighboring nodes as well. However, it is shown in [Moor92, pp. 39–40] that an RQT contains no more than eight times as many nodes as its non-restricted counterpart. In big O notation the space complexity of the restricted version of a quadtree is equivalent to that of the non-restricted one.

To deal with the non-shared vertices that do occur in an RQT, each node is triangulated. The triangulation is done first by means of its two diagonals, forming four triangles. If any of these triangles faces a smaller node, it is incident on a non-shared vertex. That vertex is incorporated into the triangle by splitting the latter into two triangles. Since this is done on each of the four sides independently, the original node may end up with anywhere from four triangles, if no splits occur, to eight, if splits occur on all sides. Figure 3.2 depicts a node in a RQT along with neighbors of all possible sizes: equal to the node (along sides A and C), half as large (along side B) and twice as large (along side D).

3.2.2 Restricted Quadtree Variants

The boundaries between equal-sized nodes may or may not be split; the decomposition rules described so far neither require nor preclude such splits. Deciding one way or the other provides for two variants of the RQT, known as the 4-triangle and 8-triangle rules. The names are derived from the number of triangles that each node would have if all nodes had the same size. The 4-triangle rule, which mandates that the boundary between equal-sized nodes be kept in one piece, provides for a more concise tree with fewer triangles. Conversely, the 8-triangle rule, which splits nodes into as many triangles as possible without violating the other rules, is likely to produce a better-fitting surface because it incorporates more

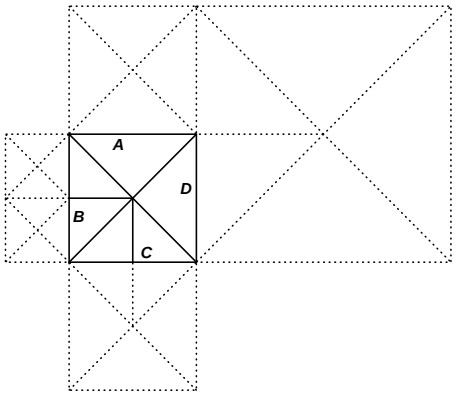

Figure 3.2: Restricted quadtree node relationships. The edge neighboring smaller nodes (B) is split; the edge facing a larger node (D) is not. On the sides where equal-sized neighbors are found, the node is split only if the 8-triangle rule is in effect (C), and it is not split if the 4-triangle rule is used (A).

data points. Figure 3.2 shows the result of applying the 4-triangle rule on side A and the 8-triangle rule on side C.

3.2.3 Related Concepts

Below we describe some concepts relevant to the way RQTs operate. They are described here in detail so that later they can be referred to concisely.

Mandatory and optional vertices: As mentioned above, an RQT node can consist of as few as four or as many as eight triangles, depending on the sizes of the nodes surrounding it. A node with the minimal configuration of four triangles has five vertices: four at the corners and one at the center (Figure 3.3a). All RQT nodes have these vertices as a minimum, so we shall refer to them as mandatory. A node with a maximal configuration of eight triangles (Figure 3.3b) has, in addition to the five mandatory ones, four optional vertices: one at the center of each edge.

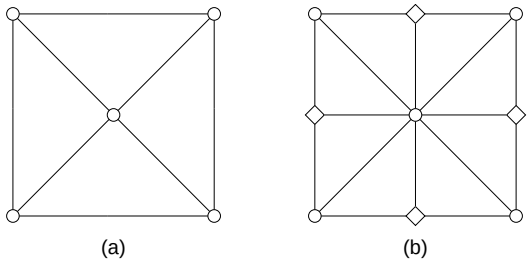

Figure 3.3: Mandatory vs. optional vertices in RQT nodes. Circles denote mandatory vertices; diamonds denote optional vertices. (a) Minimal configuration of four triangles, using only mandatory vertices. (b) Maximal configuration of eight triangles, using all possible vertices, mandatory and optional.

The different selections of the four optional vertices give rise to $2^4 = 16$ configurations an RQT node can have, as shown in Figure 3.4.

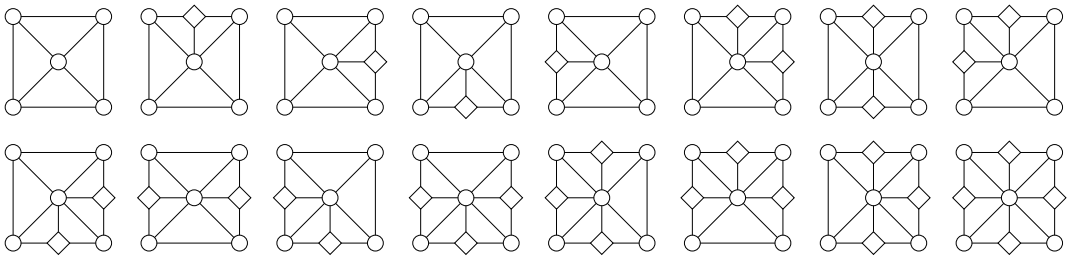

Figure 3.4: The 16 possible configurations of an RQT node.

Inherited and uninherited vertices: Every non-root node in an RQT shares some vertices with its immediate parent. Such vertices are denoted as inherited. The vertices at the four corners of a node q are inherited, while the vertex at q's center and those at the midpoints of q's edges are not (See Figure 3.5). Vertices that are not inherited are called uninherited. Note that the vertices at the corners of the node are both mandatory and inherited, and that the midpoints of the edges are both uninherited and optional. Only the node at the center of the node is uninherited yet mandatory.

The definition above pertains to non-root nodes only. For completeness we categorize the vertices of the root node as well. Since the root itself has no parents and therefore cannot inherit anything, we define all its vertices as uninherited.

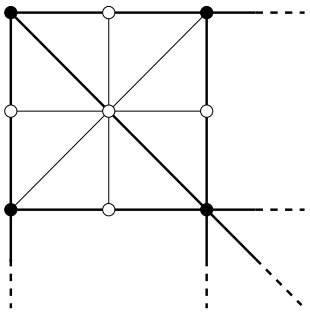

Figure 3.5: Inherited vertices in RQT nodes are denoted by solid circles, while uninherited vertices are denoted by empty circles. Thin lines correspond to the edges of the node, while the heavy lines correspond to the edges of the node's parent. The uninherited vertices of the node are positioned where its parent cannot have any vertices.

Stored elevations are the elevations at the vertices of an RQT node. The stored values determine the inclinations of all the triangular faces of the node. Stored values normally correspond to values read from the input representing actual measurements made on the real surface. The implementation guarantees access to these values at retrieval time, hence the label "stored". Note that only elevations stored in the inherited vertices of a node q are

also stored values in q's parent; the elevation at an uninherited vertex of q is necessarily a computed one (see next definition) in the parent of q.

Computed elevations are those produced by the surface model for points other than vertices by interpolation of the relevant stored elevations. These are therefore not input values and may differ from the actual elevation of the surface at the same location. However, it is assumed that the data points in the input are sufficiently dense so that the chances of such a discrepancy being significant are slim. This assumption is not specific to RQT models but is true in general for all sampled models.

Any non-vertex point within a node occurs in some triangle of the node (perhaps more than one, if it is on an edge). The computed elevation is obtained by interpolating the stored values found at the three vertices of that triangle. For a point common to two or more triangles, the elevation can be computed from any of the triangles the point is incident on; the result is guaranteed to be independent of this choice.

3.3 Implementation

3.3.1 Assumptions

Embedding arbitrary datasets in RQT: Like the region quadtree, the RQT describes a two-dimensional square area whose side is a power of 2. Arbitrary datasets can be embedded in an appropriate square (one whose side is the smallest power of 2 to exceed both dimensions of the dataset) by zero padding. This transformation allows us to consider only datasets of size $(2^m + 1) \times (2^m + 1)$ for some positive integer m with no loss of generality.

This transformation for arbitrary datasets may, in the worst case, induce considerable overhead in terms of empty, zero-filled RQT nodes whose presence is nevertheless mandated by the RQT decomposition rules. For example, the dataset depicted in Figure 3.6a covers an area of $2S$ pixels but incurs an overhead of $6S + 3 \log S - 6$ empty blocks. This would indeed be inefficient, since there are three empty blocks for each pixel in the area of interest, regardless of size.

However, DTMs usually span an area having the shape of an upright rectangle. The worst-case behavior of such an area is depicted in Figure 3.6b. Here the dataset dimensions exceed a power of 2 by a small amount, thereby requiring a padding area almost three times as large as the dataset. The overhead in this case is only $4S + 3 \log S - 4$ compared to the dataset area of $(S + 1)^2$ pixels. The ratio between the two is inversely proportional to S, so it improves with the size of the dataset.

Monotonicity of refinement: As explained in Section 1.1, associated with any surface model is a tolerance, which bounds the difference between the elevation values reported in the input and those computed by the model for any location. It may be assumed that since small nodes can express terrain variation with greater detail and fidelity than larger ones, decomposing a large node into its smaller descendants can only improve the tolerance of the resulting model. Although this is the general trend, it is not universally true. In particular,

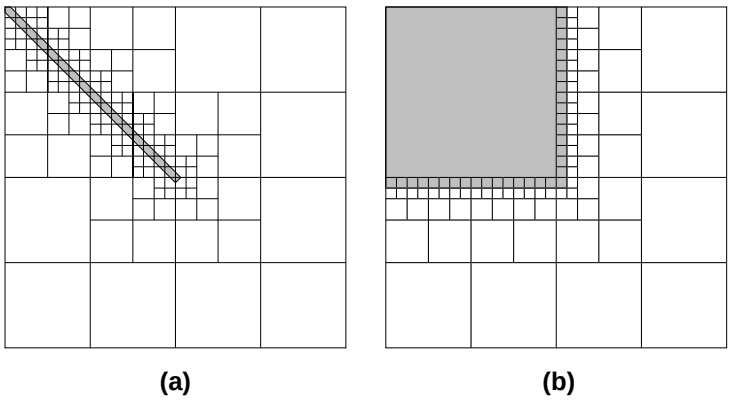

Figure 3.6: Bad cases of arbitrary datasets embedded into an RQT square. (a) Absolutely worst case. (b) Worst case involving upright rectangular datasets.

in certain cases a node split may in fact degrade the model's accuracy. It should be noted that this phenomenon is related to the way the model's accuracy is measured. If, for example, average deviation were used in Figure 3.7 instead of maximum deviation, monotonicity would have been retained. However, one could devise a case where monotonicity breaks down for that method as well.

As an example of this phenomenon, consider Figure 3.7, where the problem is shown in a one-dimensional setting for simplicity. The first-cut model, depicted in Figure 3.7b, is acceptable for some predefined tolerance (denoted by the dashed lines). However, the model fails the tolerance test when one more data point is added to it (Figure 3.7c). Further refinement (see Figure 3.7d) makes the model acceptable again. This last model is also better than the first as a tighter tolerance can now be supported, thereby demonstrating the general trend of improving model accuracy with refinement.

In this study it was assumed that every refinement improves the quality of the resulting model, disregarding the above examples to the contrary. The algorithms can be changed to account for these exceptions. Alternatively, a distinction can be made between two tolerance values: the input tolerance, which is a parameter of the RQT construction process, and the output tolerance, which is the maximum difference between the elevations given by the model and the input for the same location (i.e., the sense in which the term "tolerance" has been used so far). These two values are correlated but may differ, due to the phenomenon described above. Experimentation with varied datasets shows that only a small fraction of blocks contain points outside the input tolerance, regardless of the actual value used. However, different datasets may develop different output tolerances when constructed with the same input tolerance. Moreover, as we shall see, the top-down construction algorithm appears to produce better results in this regard than the bottom-up algorithm does.

3.3.2 Atomic Nodes

In terms of implementation, the relationship between the smallest nodes and the input needs clarification. The elevations stored at the vertices of RQT nodes, in the ideal case, are the actual input values. However, it is possible to store interpolated values, corresponding to

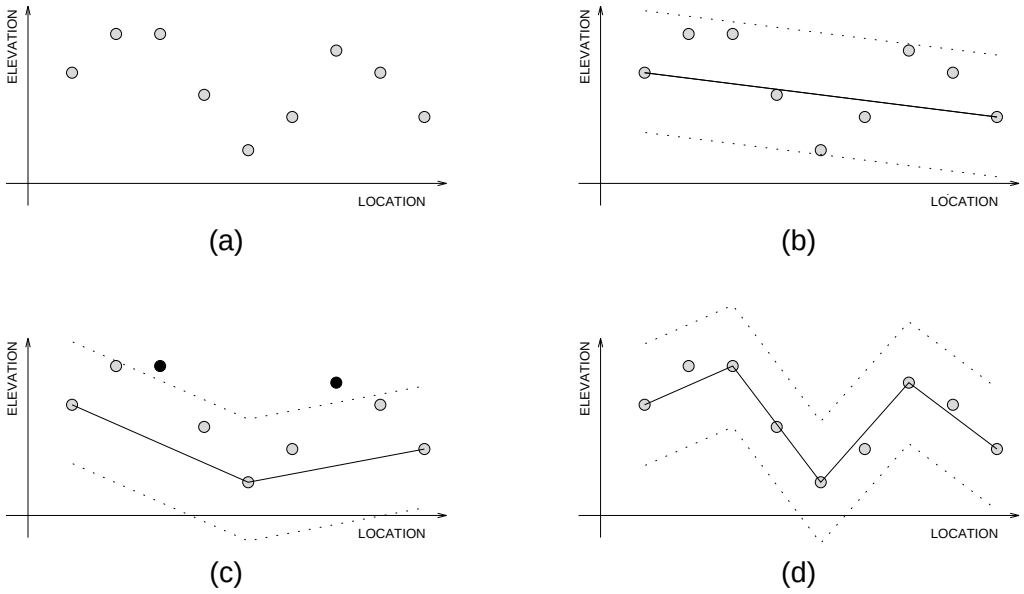

Figure 3.7: Refinement does not monotonically lead to better models. (a) A few consecutive elevation samples along a one-dimensional terrain. (b) A first-cut model of the terrain. The dashed lines indicate the range of values the model tolerance permits the data to take. (c) A refinement of the model in (b). Clearly, some data points (marked by solid circles) that earlier were within tolerance are now outside of it. (d) A further refinement of the model. Now all samples are again within tolerance. In fact, the tolerance can now be reduced almost threefold.

virtual data points, at locations that are not in the original dataset (i.e., non-grid points).

The following are three possible atomic node construction schemes:

1. A node is built around each data value, making it the center vertex of the node and assigning interpolated values to the other vertices (Figure 3.8a). Each atomic node consumes a data point exclusively, so covering the input area requires as many atomic nodes as there are data points.

2. Every four input data values whose positions form a unit grid square are used as the basis for an atomic node. The four elevations are stored in the atomic node's four corner vertices while the elevation at its center vertex is interpolated from the other four (Figure 3.8b). Each atomic node shares four elevation samples, but each data sample is shared among four atomic nodes. Therefore, on the average, there are still $4 \times \frac{1}{4} = 1$ data points per atomic node.

3. Atomic nodes may be constructed from a 3×3 subgrid (Figure 3.8c). Here all nine vertices are associated with actual input data values. Each atomic node uses one data sample exclusively (the center vertex) and shares eight others: four (the edge midpoints) with another node and four (at the corners) with three other nodes. Summing up, we get $1 \times 1 + 4 \times \frac{1}{2} + 4 \times \frac{1}{4} = 4$ data points per atomic node on the average.

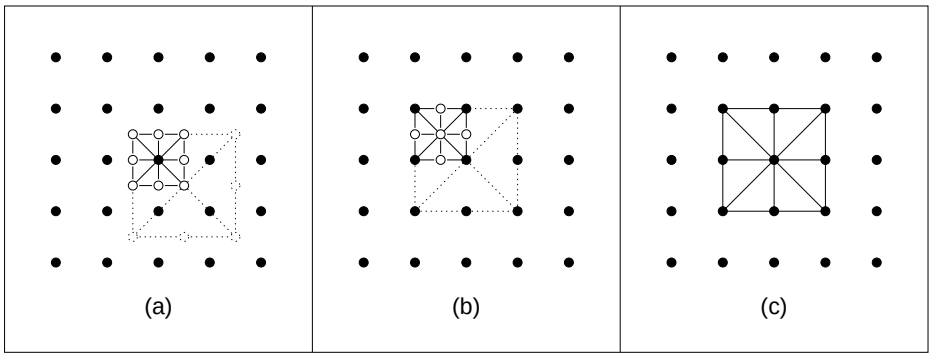

Figure 3.8: The construction of atomic nodes from raw input data. The solid circles represent the grid; the open circles are virtual data points whose values are interpolated. The ratios of input values per atomic node are (a) 1:1 centered; (b) 1:1 shared; (c) 4:1 shared. Note that when nodes merge to form larger nodes, only scheme (a) ends up with persistent interpolated values.

Approach (a) simplifies the relationship between the input data and the resulting tree. In this case, however, most of the elevation values that are eventually stored do not correspond to actual input elevation data samples. Instead, they are averages of two or four adjacent locations. Moreover, when adjacent nodes merge, the true data points are not propagated to the resulting node. It is therefore possible for the final model to contain only interpolated elevations. This may cause smoothing and other undesirable effects.

In approach (b), only one computed elevation value is stored in each atomic node, and that value is dropped completely when four atomic nodes are merged to form the smallest non-atomic node. The danger of the model ending up with many interpolated elevation values is therefore diminished considerably.

Approach (c) suffers from none of the above maladies, since atomic nodes constructed in this way do not call for any interpolated values at all. However, more input values go into the construction of a single node than in the previous schemes. As a result, difficulties may arise at the boundary of the data set if the number of rows or columns is not odd. Nevertheless, this is the approach that is used in the present implementation.

3.4 RQT Construction Algorithms

In this section we discuss the process by which an RQT model is constructed from raster (DTM) input. We present bottom-up and top-down algorithms. The bottom-up algorithm starts out with the most refined, and hence the largest RQT possible. It then prunes it as much as possible without the accuracy dropping below the predefined tolerance value. The top-down algorithm, on the other hand, starts with a single node and recursively decomposes it until the model's accuracy reaches that tolerance.

Both algorithms create nodes and then, if they are inadequate for the final model, destroy them. However, the bottom-up algorithm creates and destroys only nodes which could be subordinated to the leaves of the final result, while the top-down algorithm only goes through nodes which are superior to the leaves of the result. Due to the tree property of exponential

growth, it is expected that the top-down algorithm should run faster than the bottom-up algorithm. This was corroborated by our experiments.

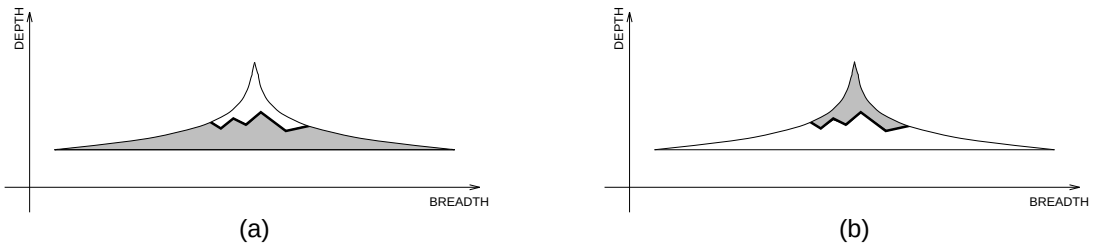

Figure 3.9: A comparison between the expected complexity of the bottom-up and top-down algorithms. For simplicity, the two-dimensional sprawl of the tree is depicted here in one dimension. The heavy line represents the leaf nodes of the RQT ultimately constructed. (a) Descendant nodes generated by the bottom-up algorithm. (b) Internal nodes that are generated by the top-down algorithm.

Below we describe the implementation of both algorithms in greater detail. Section 3.4.1 and Section 3.4.2 describe the bottom-up and the top-down algorithms, respectively. Experimental results are provided in Section 3.4.3.

3.4.1 The bottom-up Construction Algorithm

At the start we build an RQT which consists only of atomic nodes, representing the input data at the greatest level of detail possible for the given tree depth. Next, we merge adjacent nodes where doing so would violate neither the accuracy of the model nor the restrictions on neighbor sizes (Figure 3.10).

Step 2 in Figure 3.10, called the input phase of the algorithm, is where input is read and the atomic nodes are constructed. Depending on the implementation of atomic nodes in terms of data samples (as discussed in Section 3.3.2), the input is read in either row by row or two rows at a time. Note that this is done in the input's natural order. Since all atomic nodes have the same size, the resulting model obeys the RQT restrictions on neighbor sizes and therefore is an RQT.

Steps 5 and 6 of Figure 3.10 form the merge phase. In this phase each level of the tree, starting with the one just above the one constructed in the input phase, is visited in turn. At each level, every leaf node is tested for mergibility. If a node is mergible, then it is coalesced with the three other nodes with which it shares a parent. The four siblings are then deleted from the tree and replaced by their parent node. A leaf node is mergible if all the conditions in Figure 3.11 are met.

The algorithm terminates once all the nodes at a given level have been processed and no more merges can be performed. Termination is guaranteed since once the root level is reached, no more merges can take place.

Analysis: We assume that the dataset describes a square patch of $s \times s$ equally spaced data points, where $s = 2^m + 1$ for some integer m (see Section 3.3.1 for justification). Let

Bottom Up RQT Construction Algorithm

1. $R \leftarrow$ empty RQT.
2. Read input elevations in sequence, create an atomic node for each elevation and insert it into R.
3. $S \leftarrow$ size of atomic node.
4. **while** S is less than the size of the area covered by the RQT **do begin**
5. **for** each block B of size S **do**
   if B is mergible (see Figure 3.11 for mergibility conditions) **then**
   coalesce it with its siblings.
6. **if** no nodes were merged during the execution of the loop in step 5 **then**
   stop
   else
   $S \leftarrow 2S$.

end

Figure 3.10: Algorithm to construct an RQT from raster data using the bottom-up approach.

N denote the total number of points in the dataset, so $N = s^2$. Let e denote the number of elevation values which are used in the specification of single atomic nodes (depending on the definition, e could be either 1 or 4; see Section 3.3.2). The number of atomic nodes formed in the input phase is N/e. The only I/O associated with establishing an atomic node involves reading the input (T_{input}) and inserting the complete node into the RQT (T_{insert}). Therefore, the time to perform the input phase is given by

$$T_{input-phase} = N \times T_{input} + \frac{N}{e} \times T_{insert} = N(T_{input} + T_{insert}/e).$$

At each level, each node is tested. Each test operation involves retrieving the node, its three siblings and no more than eight of their collective neighbors (if they have more than eight neighbors combined, at least one of them must be small enough to block the merge). If the node passes the test, the four siblings are deleted and their parent inserted. The maximum time to process a merge is given by

$$T_{merge} = 12 \times T_{retrieve} + 4 \times T_{delete} + T_{insert}.$$

Conditions for Merging Nodes

1. The node is the north-west child of its immediate parent. This condition guarantees that each merge is considered only once, when the first eligible leaf is encountered.

2. The size of the node is equal to the sizes of its currently existing east, south and south-east neighbors.

3. The node and its three siblings have no optional vertices (those at the midpoints of their edges) or any such vertices that do exist may be eliminated. This can be done only if the elevation stored in any optional vertex is within tolerance of the elevation computed for the same location in its absence (Figure 3.12b).

4. The elevation values stored at the uninherited vertices of the nodes being merged must be within tolerance of the values computed for their locations in a parent node when it is created (Figure 3.12c).

5. Neither the node nor its three siblings have any smaller-sized neighbors. Note that if any such neighbor q existed, and the proposed merge were to proceed, then the merge would yield a node four times larger than q in q's neighborhood, in violation of the RQT definition.

Figure 3.11: Required conditions for merging nodes in an RQT.

The number of iterations made in the merge phase is dependent on the actual data. The worst case is that of a completely flat surface, which can be represented by an RQT with a single node. In this case, the bottom-up algorithm will need to merge all the nodes in the tree, incurring the greatest overhead. Conversely, the best case is represented by a surface whose variation is so great that no merging is possible at all. In such a case the algorithm can stop after one pass over the atomic nodes. Realistic running times should fall between those computed for these two extremes.

In the best case, the atomic nodes formed in the input phase are scanned but all fail the merge test. The quarter of the atomic nodes that are north-west siblings which do not fail requirement 1 of Figure 3.11 will fail requirement 4 (by assumption of best case). Although satisfaction of requirement 1 may be determined without any I/O, checking requirement 4 requires the retrieval of four siblings, thereby incurring a cost of $4 \times T_{retrieve}$ time units. Given

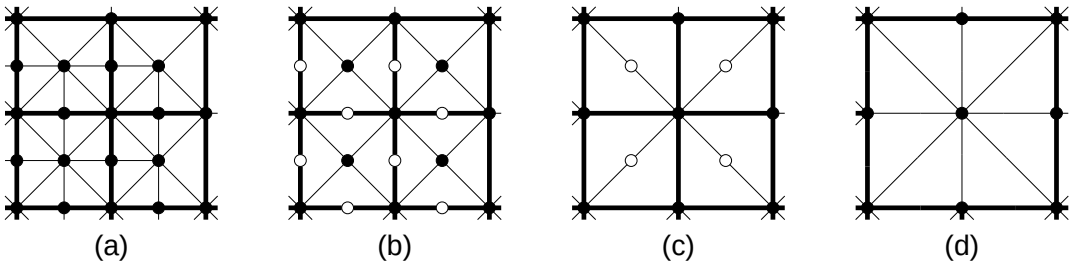

Figure 3.12: Merging nodes: heavy lines denote node extent, light lines denote internal node decomposition. (a) Initial four leaf nodes; (b) Intermediate stage I: open circles denote optional vertices. Such vertices must be removed for the merge to proceed. A vertex can be eliminated if the elevation it stores falls within tolerance of the computed value for the same location. (c) Intermediate stage II: Open circles denote uninherited vertices, which must also be removed. (d) Resulting merged node.

that there are N/e atomic nodes, the total time for best case merge phase is

$$T^{best}_{merge-phase} = \frac{N}{e} \times \frac{1}{4} \times 4 \times T_{retrieve} = N(T_{retrieve}/e).$$

In the worst case, all possible internal nodes are formed at some point. Since internal nodes total one third of the number of leaf nodes, with worst case the time needed to execute the merge phase is given by

$$T^{worst}_{merge-phase} = \frac{1}{3} \times \frac{N}{e} \times T_{merge} = N(T_{merge}/3e).$$

Therefore, the execution time $T_{bottom-up}$ of the bottom-up algorithm satisfies the following inequality:

$$T_{input-phase} + T^{best}_{merge-phase} \leq T_{bottom-up} \leq T_{input-phase} + T^{worst}_{merge-phase}.$$

The execution times associated with RQT operations (e.g., $T_{retrieve}$, T_{delete}, T_{insert}) are all related to the depth of the RQT, which is $\log_4(N) = \log_2(N)/2$. Therefore, the above bounds are both $O(N \log N)$, which means that the running time of the bottom-up algorithm is also $O(N \log N)$. Note that this time is for any RQT, irrespective of the actual elevation values or the size of the resulting RQT.

3.4.2 The top-down Construction Algorithm

This algorithm attempts to adapt the ideas of the predictive quadtree construction algorithm described in [Shaf87a]. In the course of constructing an area quadtree from raster data, the predictive algorithm only splits nodes—it never merges any. The algorithm is therefore optimal in the sense that the work it does is proportional to the size of the eventual output. To accomplish this, the algorithm maintains a partially-constructed minimal quadtree that is consistent with the data read so far by making optimistic assumptions about the unread portion of the input. As data is read in, only the minimal changes required to regain

consistency with the new input are applied to the quadtree. When all the input has been processed, the result is the desired quadtree.

Like the predictive quadtree construction algorithm, the top-down construction algorithm maintains a partially constructed RQT which is consistent with the data processed so far. However, the order in which the construction proceeds is driven by the levels in the tree rather than the order of the input.

The first step is to construct the root node, using the input elevation values associated with the locations indicated in Figure 3.13a. Once this has been done, and throughout the construction process, an elevation value is associated with every point within the map's extent, reflecting either an input (stored) value or an interpolated (computed) value. Initially, these values will probably represent only a poor approximation of the surface since only nine values are stored and the rest are computed. However, as more nodes are inserted, the computed values approach those of the desired surface.

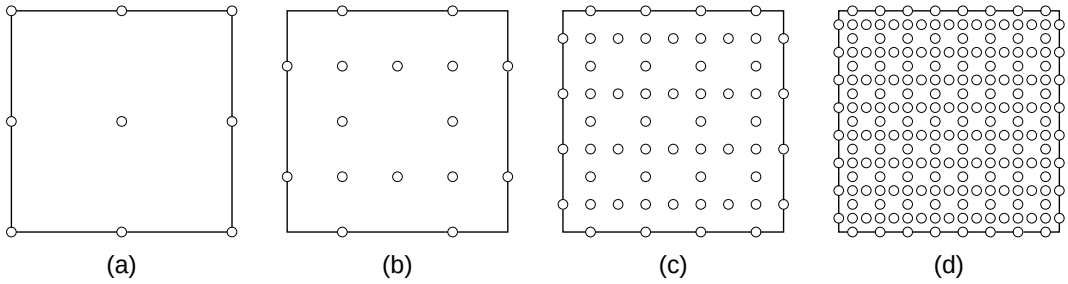

Figure 3.13: Locations of uninherited vertices in the first few levels of an RQT: (a) root level; (b), (c), and (d) levels 1, 2, and 3, respectively. The square represents the extent of the RQT.

On each level after the first, each possible node is considered in turn. For each node, the uninherited vertices are determined. The locations of these vertices for levels 1, 2, and 3 are shown in Figures 3.13b, 3.13c, and 3.13d, respectively. The elevation given in the input for each such uninherited vertex is compared with the value computed from the current tree. If the two values are sufficiently different, the node is constructed and inserted into the current tree.

However, in contrast to the bottom-up algorithm, the tree may need some preprocessing before the node can be inserted. Not all of the node's siblings and ancestors that are mandated by quadtree structure need be present at the time the insertion is attempted. They must be generated and inserted first.

To see how this may come about, consider Figure 3.14. Figure 3.14a depicts a node in the tree being constructed, say at level ℓ. It is possible that a pass over the next level, $\ell + 1$, will not yield any discrepancy with the input within the bounds of this node, as seen in Figure 3.14b. In the next step, at level $\ell + 2$, an input value which differs sufficiently from the computed elevation for that location is detected (solid triangle in Figure 3.14c). The node at level $\ell + 2$ which contains this point should now be inserted. However, it is too small to be inserted directly, since its neighbors would be larger than twice its size. Therefore, its parent node on level $\ell + 1$ needs to be inserted first, along with its siblings, as in Figure 3.14d. Only once this has been done can the small node be inserted (Figure 3.14e).

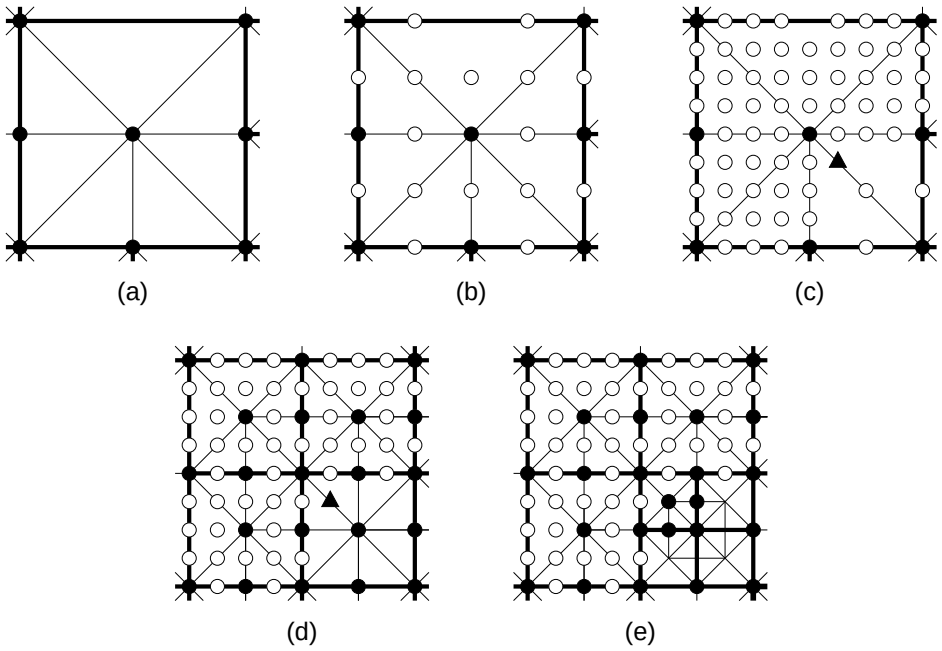

Figure 3.14: Example of node insertion when using the top-down algorithm: Solid circles denote stored elevation values while open circles denote locations where the input and computed values were found to be in agreement. (a) Initial state of a node q; (b) the situation after the construction pass over the level immediately below the one containing q; (c) the state when, during the pass over the second level below q, an input value (marked by a triangle) which significantly differs from the elevation computed for the same location is encountered; (d) first split; (e) final split.

Procedure incorporate manages these insertions in the implementation of the top-down algorithm. Figure 3.15 illustrates how procedure incorporate works. Assume a partially constructed RQT (Figure 3.15a) is decomposed when a small block B (highlighted in Figure 3.15b) needs to be incorporated into the tree. First, the smallest ancestor of B in the tree, call it A, is located and deleted (Figure 3.15c). Next, all the descendants of A which do not cover B are generated and inserted into the tree (Figure 3.15d). This process is repeated with the descendant of A which does cover B (Figure 3.15e). The process stops when A equals B, at which time B may be inserted (Figure 3.15f). Figure 3.16 shows procedure incorporate in pseudo-code.

The top level control structure of the top-down algorithm is shown in Figure 3.17. Loading the input data, following the general logic outlined above, is achieved by procedure load (Figure 3.18). It produces a quadtree with sufficiently many stored values to support the input tolerance. However, this quadtree is not necessarily restricted. Procedure restrict (Figure 3.19) converts the quadtree into an RQT. This approach produces simpler code and also supports the out-of-core version of the algorithm to be described below.

Procedure restrict operates on the output of procedure load in a separate pass. To keep track of the nodes it has already processed it maintains a bit per node called the mark bit. At any time during the execution of this procedure, the population of marked nodes does not violate the rules of restricted quadtrees. Unfortunately, marking a node q may not be

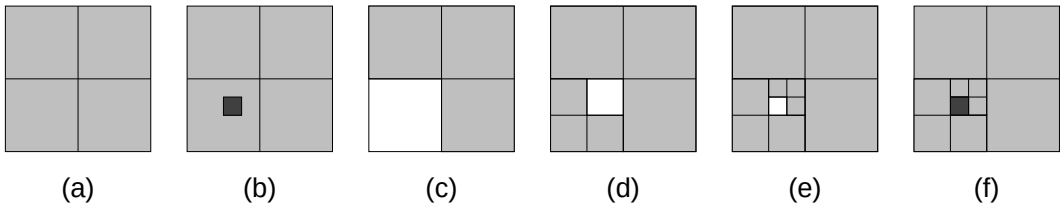

Figure 3.15: Steps in the execution of procedure incorporate. (a) A partially constructed RQT. (b) A small block (highlighted) needs to be incorporated into the tree. (c) The new block's ancestor is deleted from the tree. (d) The ancestor's descendants which do not cover the new block are inserted. (e) The process repeats at the next lower level. (f) Termination: the level of the new block has been reached and it can be safely inserted into the tree.

the last time node q is processed. Some unprocessed neighbor of q, say r, may exist that is too small to remain in q's neighborhood. This will force q to decompose later, when r is processed. In reprocessing marked nodes procedure restrict is helped by the recursive procedure backtrack (Figure 3.20).

Analysis: We adopt the same assumptions and notation used in the analysis of the bottom-up algorithm (Section 3.4.1)—that is, $N = s^2$ is the total number of data points in an input of $s \times s$ array where s is one more than a perfect power of two. In addition, let L_{final} be the number of nodes in the RQT that is ultimately constructed. Also, only disk I/O is considered in our analysis.

The total running time of the top-down algorithm, $T_{top-down}$, consists of the time spent examining the input ($T_{examine}$), constructing the result ($T_{construct}$), and restricting the resulting quadtree to form a RQT ($T_{restrict}$). The tasks of examining the input and inserting nodes into the resultant RQT are common to all constructions, and their complexity depends only on the size of the surface being modeled, not its shape. On the other hand, the cost of converting the quadtree into a RQT depends on the shape of the surface.

$$T_{top-down} = T_{examine} + T_{construct} + T_{restrict} \tag{3.1}$$

Examining the input is done in the load phase (step 2 of Figure 3.17, and given in greater detail by procedure load in Figure 3.18). The entire input dataset is scanned, at least to verify that the model is consistent with the data, even if no nodes are generated as a result. Each block at each level is examined in turn, a total of $4N/3$ blocks. Examining a block entails reading the nine elevation values corresponding to its vertices from the input. Note that the nine input values are unlikely to be found in consecutive locations on the input medium, and thus obtaining them may be expensive. The total time spent reading the input is given by

$$T_{examine} = \frac{4N}{3} 9 \times T_{input} \tag{3.2}$$

Operations that result in the insertion and deletion of nodes are scattered throughout both the load and restrict phases, but calculating the time spent executing them is straightforward. Unlike procedure bottom-up, in procedure top-down nodes are never merged. Other

```
Incorporate Node N Into Quadtree R

1. B_A ← the smallest block in R which covers N.
2. delete B_A from R.
3. while B_A is larger than N do begin
4.    for each child block B_C of B_A which does not cover N
      do begin
5.       N_C ← the RQT node whose extent is B_C and elevation
         data obtained from the input.
6.       insert N_C into R.
      end
7.    B_A ← the child block of B_A which does cover N.
   end
8. insert N into R.
```

Figure 3.16: Procedure incorporate which incorporates a node into an RQT. It is an auxiliary routine for the top-down algorithm.

than the root, nodes are inserted only when their parents are decomposed. Since their total number is known, the amount of work they require can be computed. If the final result contains L_{final} leaf nodes, then $(L_{final} - 1)/3$ internal nodes must have been present at various times during its construction. Each node was replaced by its four children when it was deleted. Rounding up this number to $L_{final}/3$ we find that constructing the RQT actually takes

$$T_{construct} = \frac{L_{final}}{3}(T_{delete} + 4 \times T_{insert}) \tag{3.3}$$

In the worst case, any single operation on the RQT requires a traversal of a path from the root to a leaf node, having a cost of $\log L_{final}$. For simplicity we assume that all of $T_{retrieve}$, T_{insert} and T_{delete} are proportional to $\log L_{final}$. Therefore, the above expression can be rewritten as

$$T_{construct} \propto L_{final} \log L_{final} \tag{3.4}$$

The third term contributing to $T_{top-down}$, the time required to restrict the quadtree (procedure restrict), is difficult to gauge exactly since it depends on the shape of the surface being modeled. However, it can be assessed using amortization analysis. Note that since nodes are never merged in the course of top-down construction, all deletions and insertions

```
                    Top Down Construction

1. R ← an empty RQT structure.

2. Load input data into R (see Figure 3.18).

3. Restrict R (see Figure 3.19).
```

Figure 3.17: Procedure top-down which constructs a RQT from raster data.

of nodes, including those performed by procedure restrict, have been accounted for in the calculation of $T_{construct}$. However, restrict also retrieves nodes from the partially constructed tree to determine the size ratios of neighboring nodes. We will show that although some nodes may be retrieved multiple times, on the average a node is examined a constant number of times.

Table 3.1: Costs associated with the different scenarios a node may be subject to in procedure restrict.

scenario	retrievals	unmarked	marked
1	8	+3	0
2	4	+4	−1
3	8	−1	+1

Let M denote the number of marked nodes and U the number of unmarked nodes at any given time during the execution of procedure restrict. Examination of the code reveals that when a node B is processed, there are only three possible outcomes, summarized in Table 3.1:

1. B has neighbors that are too small to coexist with it (line 4 in Figure 3.19). As many as eight neighbor nodes must be retrieved. B is then replaced with its four children. Since both B and its new children are unmarked, this operation's net result is to increase U by three, while leaving M unaffected.

2. B gets processed by the helper procedure backtrack (line 5 in Figure 3.19 and line 1 in Figure 3.20). Since this procedure is applied only to marked nodes, it follows B is marked. Procedure backtrack first searches for any neighbors that are larger than B. Since on any given side of a node, a larger neighbor must be the sole neighbor, only four retrievals are necessary to locate all of B's larger neighbors. The procedure continues by replacing the marked node B with its unmarked children. Consequently, U is increased by four and M is reduced by one.

```
Load Data Into an Empty Quadtree R

1. for S ranging through all permitted block sizes (from the
   size of the input down to size of an atomic node) do begin
2.    for each possible block B of size S do begin
3.       N_D ← the RQT node whose extents are B and whose
         elevations are obtained from the input.
4.       if B is the root block then
            insert N_D into R
         else begin
5.          N_I ← the RQT node whose extents are B but
            whose elevations are computed from R.
6.          if N_D and N_I disagree on elevation at any point by
            more than the predefined tolerance then
               incorporate N_D into R (see Figure 3.15).
         end
      end
   end
```

Figure 3.18: Procedure load which loads input data into an empty RQT.

3. B is compatible with its neighbors and requires no updating (only lines 6 and 7 of Figure 3.19 get executed). Again, as many as eight neighbors of B need to be retrieved to verify this. Subsequently, B is marked, reducing U by one and increasing M by one.

Let K_1 be the number of times scenario 1 is encountered in the course of processing a surface, and let K_2 and K_3 be defined similarly. The total number of retrievals $K_{retrievals}$ performed by procedure restrict is given by

$$K_{retrievals} = 8K_1 + 4K_2 + 8K_3 \tag{3.5}$$

Let $L_{intermediate}$ denote the number of nodes in the quadtree produced by procedure load, and, as above, let L_{final} be the number of nodes in the final, restricted quadtree. Initially $U = L_{intermediate}$ and $M = 0$, since all the nodes start out as unmarked. Following the execution of procedure restrict, $U = 0$ and $M = L_{final}$. Equation 3.6 restates these conditions for U while Equation 3.7 restates them for M:

$$L_{intermediate} + 3K_1 + 4K_2 - K_3 = 0 \tag{3.6}$$

Restrict quadtree R

1. $R \leftarrow$ an unrestricted quadtree with all its blocks initially unmarked.
2. **for** each unmarked block B in R **do begin**
3. locate all of B's neighbors in R.
4. **if** the size of any of B's neighbors is smaller than half the size of B **then**
   delete B and insert its four unmarked children.
   else begin
5. apply procedure backtrack to any of B's marked neighbors that is larger than twice the size of B (see Figure 3.20).
6. adjust the configuration of B and that of its neighbors to conform with the desired RQT (4-triangle or 8-triangle rules; see Section 3.2).
7. mark B.

   end

end

Figure 3.19: Procedure restrict which converts a non-restricted surface model into an RQT.

$$K_3 - K_2 = L_{final} \tag{3.7}$$

Eliminating K_2 we get

$$K_1 + K_3 = \frac{1}{3}(4L_{final} - L_{intermediate}) \tag{3.8}$$

From Equations 3.7 and 3.8 we have

$$K_2 < K_3 < K_1 + K_3 = \frac{1}{3}(4L_{final} - L_{intermediate}) \tag{3.9}$$

$$K_1 + K_2 + K_3 < \frac{2}{3}(4L_{final} - L_{intermediate}) \tag{3.10}$$

Recalling Equation 3.5:

$$K_{retrievals} < 8(K_1 + K_2 + K_3) < \frac{16}{3}(4L_{final} - L_{intermediate}) \tag{3.11}$$

Backtrack Over Marked Node N

1. Apply procedure backtrack (this procedure) to any of N's neighbors that is larger than twice the size of N.
2. Delete N from R.
3. Insert the 4 unmarked children of N into R.

Figure 3.20: Procedure backtrack which recursively handles large marked nodes. It is an auxiliary routine for procedure restrict.

In the worst (and practically impossible) case of $L_{intermediate} = 0$, the number of retrievals done by procedure restrict is no more than $\frac{64}{3} L_{final}$. This is a very crude approximation, but it is sufficient to show that $K_{retrievals}$ is $O(L_{final})$. Assuming again that $T_{retrieve}$ is $O(\log L_{final})$, the total time required to restrict the quadtree is

$$T_{restrict} \propto L_{final} \log L_{final} \tag{3.12}$$

The total running time for the top-down algorithm can now be stated as

$$T_{top-down} = T_{examine} + T_{construct} + T_{restrict} = O(NT_{input} + L_{final} \log L_{final}) \tag{3.13}$$

This seems to be the optimal result that the top-down algorithm was designed to attain—an execution time proportional to the size of the output produced rather than the size of the input (if not for the term containing N). It is interesting to observe that analyses of sub-linear algorithms often ignore the fact that, in principle, the input size may be driving the algorithm's execution time. It is usually assumed that modern operating systems are capable of reducing the overhead in making the input available to the point that it is negligible when compared with the other tasks performed by the algorithm being studied. However, in our case this assumption fails: the algorithm accesses input in an unpredictable manner. In fact, T_{input} can entail a physical disk access, and therefore cannot be ignored.

Clearly, the above concerns do not apply if sufficient random-access memory (RAM) is available to store the entire input dataset. The input values can then be accessed in any order without penalty. In order to handle real-world applications that are likely to involve larger datasets, we propose to decompose the input into blocks, along the lines of the decomposition that the first few levels of a quadtree would follow. The data relevant to each such block would be placed in a separate disk-based subfile in DTM format, using procedure split-file of Figure 3.21. The number of levels used is chosen based on the amount of available RAM; each such subfile should fit entirely into a RAM buffer.

Procedure split-file must obviously read every input item and then write it out, so essentially it moves $2N$ items. However, since consecutive data transfers are the most efficient, a

```
                    Split Input into Subfiles

1. n ← the side of the input.

2. d ← the depth of subfile decomposition.  There will be 4^d
   subfiles generated, arranged in a 2^d × 2^d array.

3. m ← n/2^d + 1 (the dimension of a subfile).

4. for m_1 = 1 through 2^d do begin

5.    for r = 1 through m do begin

6.       read a row from the input.

7.       for m_2 = 1 through 2^d do begin

8.          write the next m items from the row into
            subfile [m_2, m_1].

         end

      end

   end
```

Figure 3.21: Procedure split-file which splits the input file into subfiles, each small enough to fit entirely into the available RAM.

better measure of execution time is the number of such transfers initiated. Each row in each subfile requires a separate transfer since a row is the longest chunk of data the algorithm can move without interruption. If the input file (of $n \times n$ items) is split into 4^d subfiles, then $n(1 + 2^d)$ consecutive data transfers are required.

The main penalty of splitting the input into many smaller files is not time but space. Not only the input file and the result RQT need to be accommodated, but the subfiles as well. The subfiles take as much space as the input does, which could be considerable. On the other hand, the input can easily be reconstructed from the subfiles, so if space is an issue, the original input file may be discarded after it is split and its space released.

Procedure load of Figure 3.18 is then run on each subfile, either in sequence or, if the appropriate hardware is available, in parallel. The output from all the invocations of load is accumulated in a single RQT structure. This can be done because the part of the surface processed by each such invocation is disjoint from the part used by any other, and so are the extents of the resulting RQT nodes. The output streams from different invocations of load therefore do not interfere with each other.

After the entire input is processed, the RQT that has been built contains all the eleva-

tion data. However, in fact it is not an RQT; blocks of different sizes may freely border each other in violation of neighbor restrictions. This occurs not only inside subfiles, but mostly on the boundaries between subfiles. The nodes on both sides of such boundaries are created by different invocations of load which do not communicate and cannot resolve such inconsistencies. Here is where the utility of having the restrictions imposed in a second pass on the output comes in handy. Procedure restrict can now be run once on the total output, reforming it into a true RQT. This out-of-core version of the top-down procedure is described in Figure 3.22.

Out-of-core Top Down Construction

1. $R \leftarrow$ an empty RQT structure.
2. **for** each subfile SF **do begin**
3. read subfile SF into a RAM buffer.
4. load the contents of SF into R using procedure load (see Figure 3.18).

   end
5. restrict R (see Figure 3.19).

Figure 3.22: The out-of-core version of procedure top-down with the code of Figure 3.17 upgraded to work with very large datasets.

3.4.3 Experimental Results

The construction algorithms were tested on three datasets, each a raster of 513×513 elevations:

1. "Salisbury east 2,0" is part of an area on Maryland's eastern shore which is essentially very flat (Figure 3.26). Elevations on this map are between 0 and 5 meters.
2. "data", a map of unknown origin. It describes terrain of moderate variability (Figure 3.27). Its elevations range between 0 and 206 meters.
3. "Reno west 0,0" is a portion of an area near Reno, Nevada, in the midst of the Rocky Mountains, that is quite rugged (Figure 3.28). Its elevations reach 3244 meters.

The first and third maps are from the USGS 1-degree DTED DEM collection [USGS90]. They are both sections of size 513×513 cut from an original USGS sheet, which measures 1201×1201.

The results of construction times are summarized in Table 3.2 and shown graphically in Figures 3.23, 3.24 and 3.25. It is clear that the top-down algorithm is faster than the bottom-up most of the time, as expected. The bottom-up algorithm outperforms the other only for very low tolerances, where the final result is is very close to a complete tree. In that case, as explained in Section 3.4, the bottom-up algorithm can terminate shortly after its input phase, whereas the top-down algorithm must go through all the intermediate nodes first, and then perform an equivalent of the input phase.

Table 3.2: Experimental results of constructing various maps, using the 8-triangle rule. Execution times on a SUN SPARCstation 5 are given in seconds.

map name	map size	algorithm	tolerance					
			1	3	10	30	100	300
Salisbury	513×513	top-down	220	197	170	171	172	173
east 2,0		bottom-up	1100	1073	1046	1040	1037	1035
data	513×513	top-down		299	233	207	192	174
		bottom-up	1106	1032	1038	1027	1040	1046
Reno	513×513	top-down	1337	1250	876	413	256	204
west 0,0		bottom-up	1424	1696	1500	1166	1119	1079

An interesting phenomenon involves the behavior of the running times for the bottom-up algorithm. It was expected that its execution time would increase with the tolerance of the map being constructed, since for larger tolerances more nodes need to be merged. Experimentation shows that this is true initially, but then the time trends downward, just as top-down does.

The reason for this behavior stems from the inhomogeneity of the terrain. It would not occur if the terrain were evenly flat or evenly rugged, and is particularly pronounced in maps that have spots of great variability as well as relatively flat areas, such as "Reno west 0,0". When building with low tolerance, the first pass of the bottom-up algorithm ends up merging just a few nodes, leaving the leaf layer of the RQT very populated. In an inhomogeneous map, however, there are some regions of slow variability which require several passes to fully construct. (Recall that the bottom-up algorithm processes a single layer in each pass.) As a result, the relatively populated leaf layer of the RQT gets traversed several times. On the other hand, when the construction is done using a high tolerance, many of the nodes get merged directly in the algorithm's first pass, leaving fewer nodes to process in each of the subsequent passes.

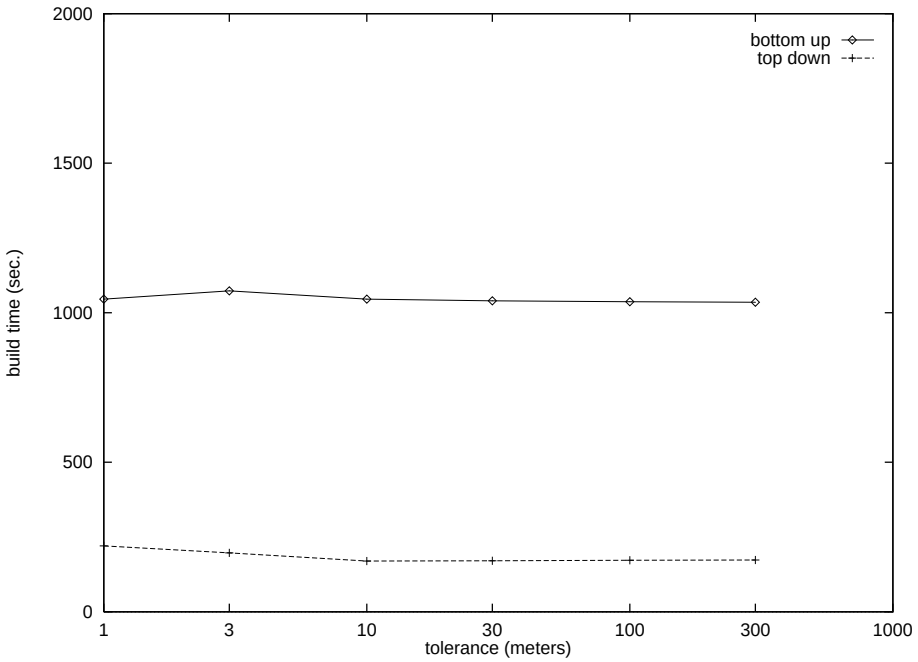

Figure 3.23: Construction times of map "Salisbury east 2,0" using bottom-up and top-down algorithms.

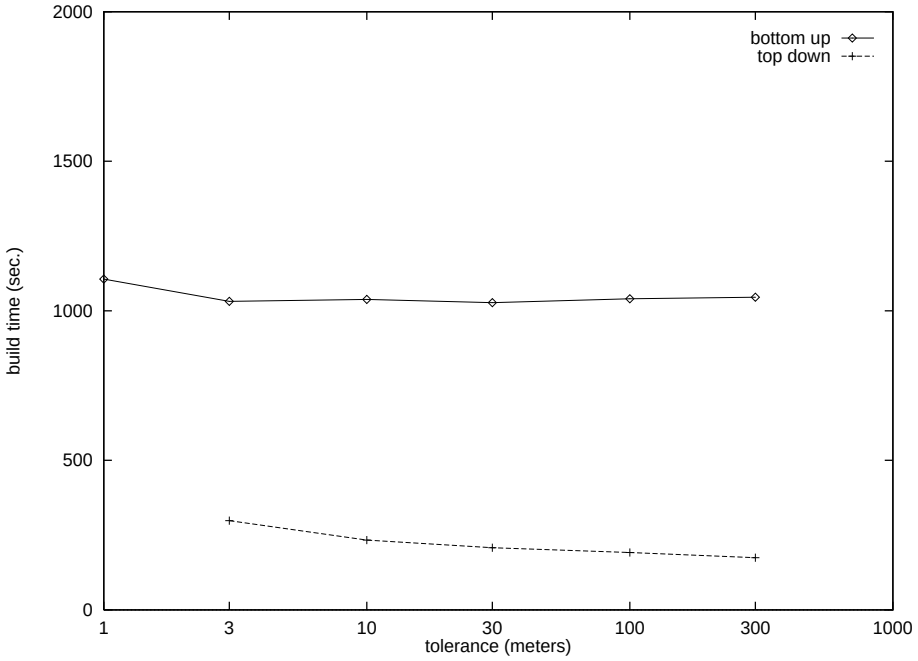

Figure 3.24: Construction times of map "data" using bottom-up and top-down algorithms.

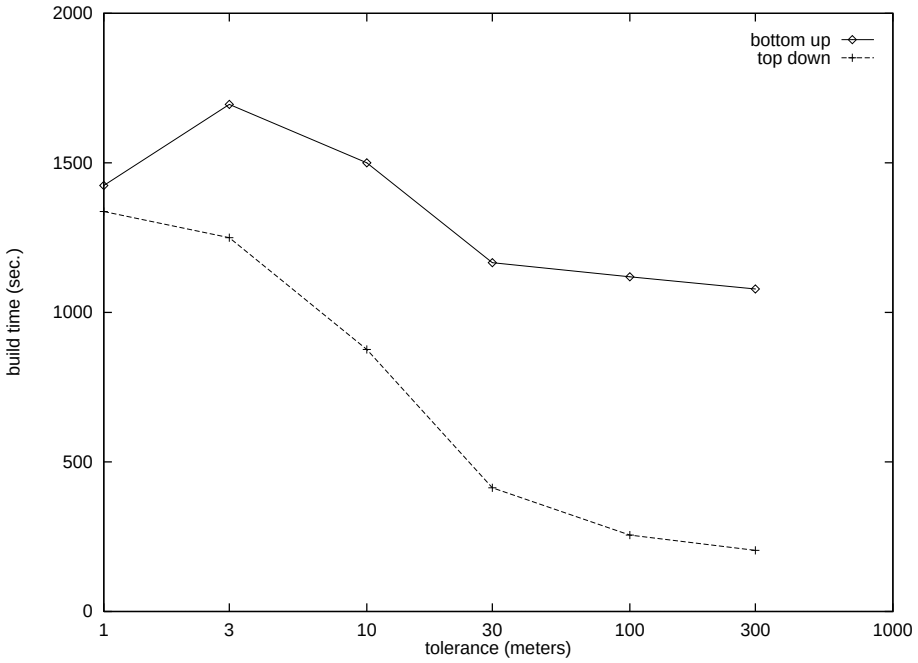

Figure 3.25: Construction times of map "Reno west 0,0" using bottom-up and top-down algorithms.

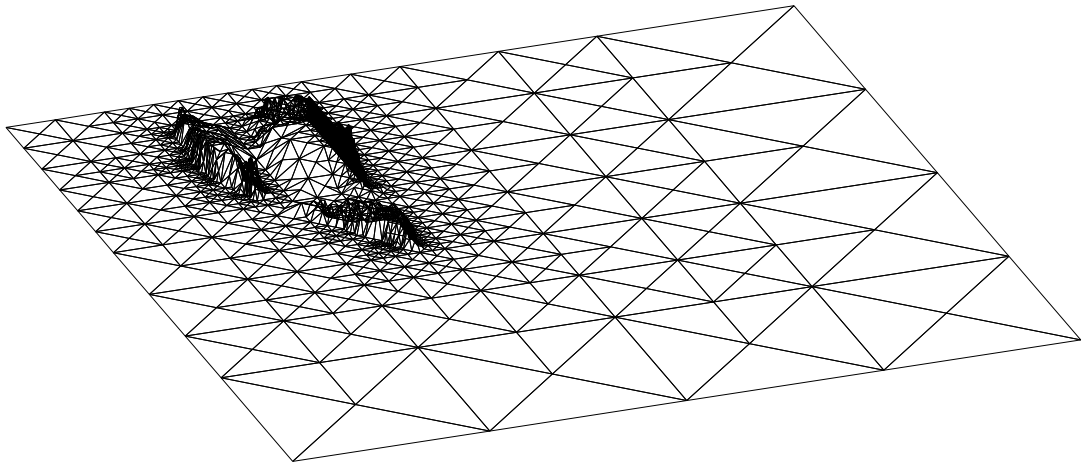

Figure 3.26: Perspective display of the 513×513 surface "Salisbury east 2,0". This is an example of a tame surface with little variation. Range of elevations: $0 - 5$ meters.

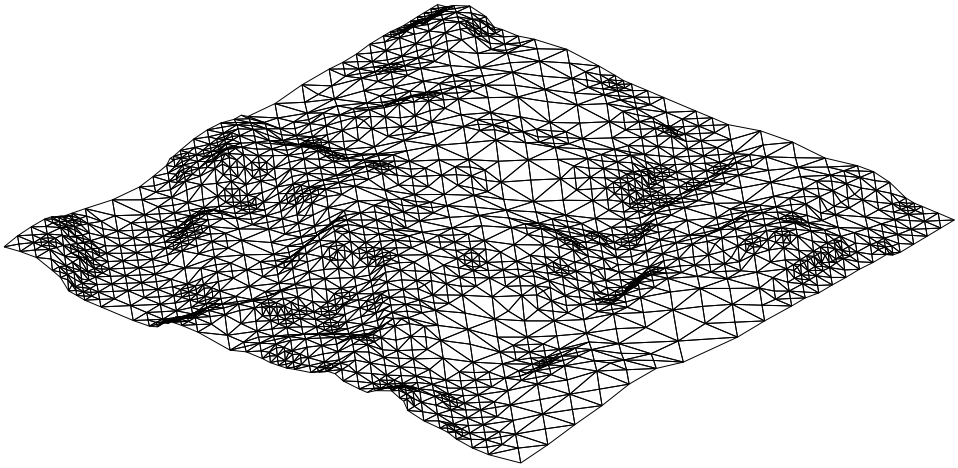

Figure 3.27: Perspective display of the 513×513 surface "data". This is a moderate surface example, whose elevation range between 0 and 206 meters.

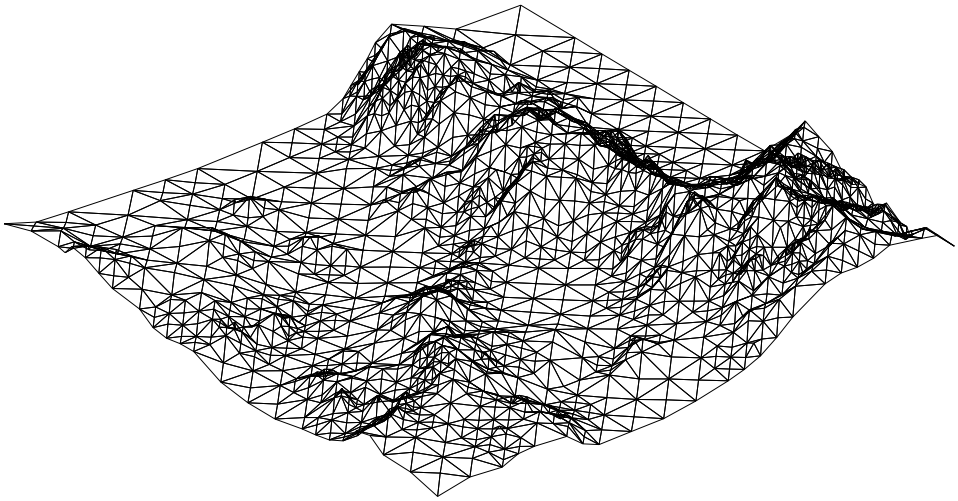

Figure 3.28: Perspective display of the 513×513 surface "Reno west 0,0". This is an example of rugged terrain. Range of elevations: 1341 – 3244 meters.

Chapter 4

The PMR Quadtree

4.1 Introduction

The PMR quadtree is a quadtree variant with many applications. In this dissertation it is used to support surfaces modeled by arbitrary triangulations (see Chapter 5). In the course of its implementation, however, some issues have come up that are interesting in their own right, and they will be covered in this chapter.

This chapter is organized as follows: After describing the PMR quadtree and what it is that makes it different from other quadtree variants (Section 4.2), algorithms for two operations that we have implemented are described: finding a nearest object to a given point in Section 4.3, and reporting all the objects found within a specified region of space in Section 4.4.

4.2 Definition

The PMR quadtree is a spatial data structure that manages spatial objects symbolically. It is effective in limiting searches to a vicinity of the search point to determine the existence of objects. Unlike the area quadtree, however, it is less suitable for queries involving the extents of objects.

While an area quadtree actually codes the extents of the objects it contains (by means of a raster), the PMR quadtree manipulates labels which identify objects, placing each in the parts of space in which the labeled object resides. This symbolic approach allows for greater flexibility. Thus the PMR quadtree can store objects whose descriptions are arbitrary (i.e., not confined to being a raster), and objects of different types can be combined in a single structure. The dimensionalities of the stored objects may be different from that of the space they are in (e.g. line segments embedded in the plane), and their extents may overlap.

The PMR quadtree decomposes space recursively into blocks, possibly of differing sizes. Each block is associated with the descriptions of the objects that spatially intersect it. Objects which span several blocks have their identifiers associated with each block that they intersect. The decomposition is carried out so that the number of objects associated with any block is bounded.

More specifically, the construction of a PMR quadtree is controlled by a parameter known as its splitting threshold (or simply threshold). If, following the insertion of a new object,

the number of objects associated with a block exceeds this threshold, the block is split once but only once. Although it is possible that the number of objects in one or several of the resulting child blocks still exceeds the threshold, they are not decomposed during this insertion cycle. There is no guarantee, therefore, that the number of objects associated with a block does not exceed the threshold. This nondeterministic approach has been found in practice to be quite effective [Nels86a]. Its drawback is that the structure of the tree is somewhat random (alluded to by the 'R' in "PMR") and is dependent on the order in which the objects are inserted. The PMR quadtree was presented in [Nels86a], where the application of such a structure to a collection of line segments in the plane was described (and the term "PMR" coined).

A PMR quadtree is defined as follows:

1. When inserting a new spatial object, associate it with each quadtree block spanned by the object.

2. If, as a result of insertion, the number of objects associated with a block exceeds the predefined threshold, split that block once.

3. Following the deletion of an object, all sets of sibling blocks whose combined population drops below the threshold are merged. In contrast with the case of insertion, merging is carried out recursively to completion.

4.3 Nearest Object

4.3.1 Motivation

Since the PMR quadtree keeps track of the spatial locations of objects, finding the nearest object to a given location (called the search point) is a natural query to pose. Such a query arises frequently in applications such as vector quantization ([Arya94]) and statistical pattern recognition ([Same94a]). The problem of finding the nearest line segment in two-dimensional space was studied in [Hoel91]. However, the principles of that algorithm apply to higher dimensions as well. The three-dimensional version of the algorithm is described below.

4.3.2 The Principle

Due to its construction, any non-leaf block in a PMR quadtree must contain at least a threshold number of objects. This is because the construction rules dictate that a block is decomposed (ceasing to be a leaf) only if its population exceeds the threshold. Conversely, when the population of a non-leaf block falls below the threshold the block's descendants are repeatedly merged until it becomes a leaf.

As a minimum, it may be assumed that a PMR quadtree's threshold is at least one. (A smaller threshold would entail a block splitting with every insertion, and would prevent blocks from merging on deletion. With enough objects, such a PMR quadtree will deteriorate into

a grid of atomic-sized blocks.) Combining the two results, it is guaranteed that any non-leaf block of every PMR quadtree contains at least one object.

This lower bound on density places an upper bound on the distance from any search point and the object nearest to it, when measured in block sizes. Let the leaf block containing the search point be the base block and its immediate ancestor be the parent block. The parent block contains the search point by virtue of containing the base block. From the argument made above, the parent block is guaranteed also to contain an object (since it is not a leaf). Let this object be the limiting object. Since both the limiting object and the search point are inside the parent block, the distance between them cannot exceed the longest diagonal of that block. There may be other objects, perhaps in neighboring blocks, that could very well be nearer the search point than the limiting object, but this diagonal represents an upper bound. The search can therefore be limited to the volume of a sphere centered about the search point and having a radius equal to the longest parent block diagonal (Figure 4.1).

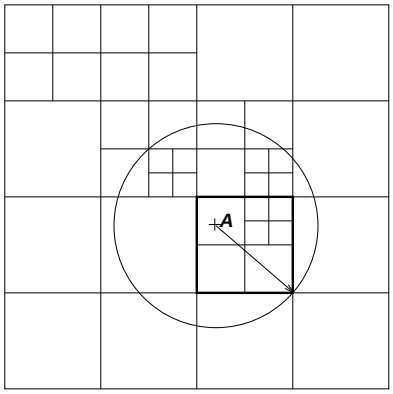

Figure 4.1: The scope of search for a nearest object in a PMR quadtree can be limited to a sphere whose radius is the distance between the search point (A) and the furthest point from it in the parent block (the parent of the block containing A).

It should be noted that this limit is on the number of blocks traversed, not on the number of objects tested. Since quadtrees are a method for decomposing space, algorithm complexity may be measured in terms of the number of quadtree blocks accessed. Assuming a 1-1 correspondence between block accesses and disk accesses, the number of blocks accessed can be a good predictor of execution time. From a geometric standpoint, a better idea would be to measure the work done in terms of the number of objects from which a nearest one must be chosen. However, using such a measure depends on establishing a connection between the number of objects in the space and the number of blocks needed to adequately cover them in a PMR quadtree. It is difficult to obtain such relationships in general, and the random character of the PMR quadtree makes it even more difficult; hence no attempt will be made to use such a measure here.

4.3.3 The Expanded Block List

This limit on the search region is given in terms of the search point and a radius, quantities that are unrelated to the elements of the PMR quadtree, i.e. blocks. Expressing this volume

in terms of quadtree blocks depends on the metric being used. The most convenient in this regard is the L_∞ (i.e., chessboard) metric, in which the locus of all points equidistant from a given point resembles a PMR quadtree block. However, in reality, the metric used most often is the Euclidean one (L_2) in which this locus (a sphere) is quite different from a block.

Assuming the more practical case of the Euclidean metric, it is helpful to determine which blocks intersect the search sphere in order to search only those. Such a list will be referred to as an expanded block list to differentiate it from a more concise list to be proposed later. Invariably, many of the blocks in the expanded list will only be partially inside the search sphere, so the volume covered by the union of all the list's blocks is larger than that of the search sphere. However, specifying the search region in terms of blocks considerably simplifies the access to the PMR quadtree.

To simplify the generation of the expanded block list, it is assumed that the PMR quadtree is populated with blocks of a single size, equal to the size of the parent block. The space to be searched is delineated by listing those equal-sized blocks that are in it, specifying their coordinates relative to the base block. In reality, a PMR quadtree will seldom be so uniform, and the algorithm must be able to handle mismatches between the blocks in the list and those actually found in the PMR quadtree (Figure 4.4). But under a uniform object distribution, this approach strikes a balance between covering too much space beyond the search sphere and making the list too long. As we shall see, even this list of uniform blocks can become prohibitively long in high dimensions.

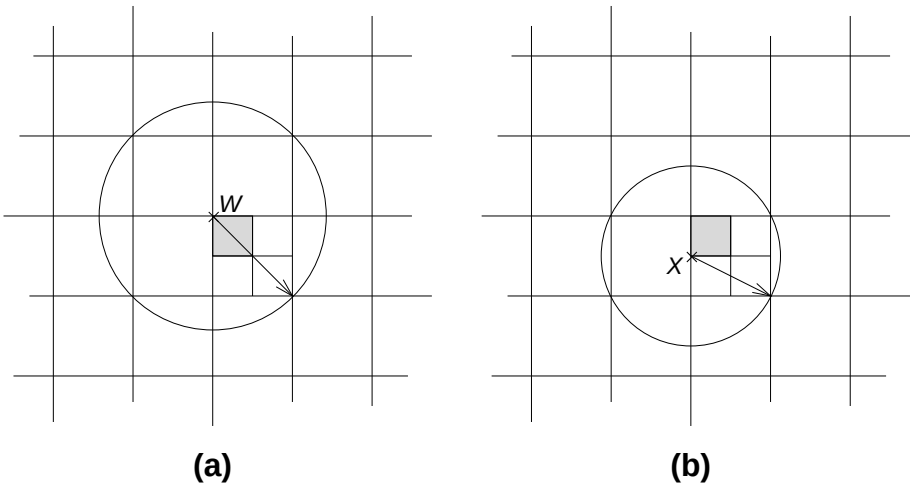

Figure 4.2: Different cases of search spheres in a two-dimensional PMR quadtree. (a) The sphere with the largest radius. (b) A different case, showing that the largest sphere does not subsume all other cases. The gray square indicates the base block.

It would be best if the expensive step of generating the expanded block list could be done once and for all, to cover all cases. Note that the union of the search spheres induced by all possible search points is not in itself a sphere. For example, consider Figure 4.2. The sphere with the largest radius results when the search point is at the extreme corner of the parent block W (Figure 4.2a). A smaller sphere is generated if the search point is X of Figure 4.2b, but nevertheless it is not subsumed by the larger sphere. The union of all such spheres is indicated by the light gray area in Figure 4.3. Note that the set of blocks which cover

this general search area is identical to the set required to cover only the maximal sphere. Consequently, the list of blocks to traverse is independent of the specific placement of the search point and hence is known prior to performing the search. Figure 4.3 shows in medium gray the blocks that need to be visited in the two-dimensional case. In the worst case, twelve blocks equal in size to that of the parent block must be accessed.

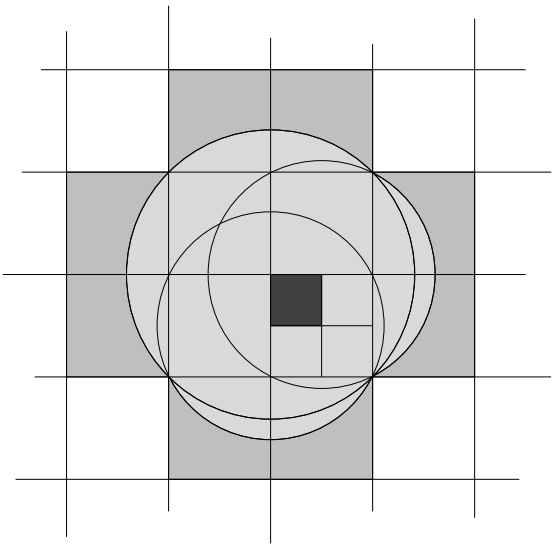

Figure 4.3: Blocks that should be searched in a two-dimensional PMR quadtree. The dark gray square is the base block. The light gray area shows the union of all possible search spheres. Medium gray indicates the blocks on the expanded block list.

4.3.4 The Algorithm

Given an expanded block list, the algorithm described in Figure 4.4 can be applied. It scans the list and determines the nearest of all the objects found in any of them. This is faster than scanning the complete PMR quadtree from which the list was drawn.

Several heuristics can be applied to improve the algorithm even further. The blocks in the list should be sorted according to their distances from the search point. That way the algorithm can terminate before the list is exhausted, in case it reaches a block whose distance from the search point is greater than that of the nearest object found so far (variable D of Figure 4.4).

The problem with keeping the expanded block list sorted is that the list can no longer be generic, as the block order depends on the actual location of the search point. Moreover, a sort is required each time step 7b of the algorithm is executed since at that point new blocks are added to the list. This latter difficulty can be minimized by sorting only the set of new blocks and then merging it with the main list, which is already sorted.

Find Nearest Object

1. $P \leftarrow$ the search point.
2. $R \leftarrow \phi$ /* the result */.
3. $D \leftarrow$ the distance between P and the vertex of the parent block farthest from it.
4. $L \leftarrow$ the expanded list with all its blocks marked "unprocessed".
5. **if** all the blocks in L are processed **then**
   output R and stop.
6. $S \leftarrow$ the first unprocessed block in L.
7. Search the PMR quadtree for block S. Let F be the block actually found. There are three possibilities:[a]
   a. $S = F$: go to step 8.
   b. $S \supset F$: replace the reference to S in the expanded list with references to all of its descendants. Mark them them all as "unprocessed" and go to step 6.
   c. $S \subset F$: scan the unprocessed blocks in L and remove references to any block which also falls inside F. (This is done to avoid redundant visits to the same block.) Then go to step 8.
8. **for** each object r in block F **do begin**
9. $d \leftarrow$ the distance between r and P.
10. **if** $d = D$ **then**
    $R \leftarrow R \cup \{r\}$.
    else if $d < D$ **then begin**
    $R \leftarrow r$.
    $D \leftarrow d$.
    end
    end
11. Go to step 6.

[a]Note that for any two quadtree blocks A and B, exactly one of the following relations holds: $A \cap B = A$, $A \cap B = B$ or $A \cap B = \phi$.

Figure 4.4: Algorithm for finding the object nearest to a given point in a PMR quadtree.

4.3.5 Nearest Object in 3-D

The list of blocks intersecting the search sphere in the case of three dimensions is illustrated in Figure 4.5 and drawn algorithmically in Figure 4.6.

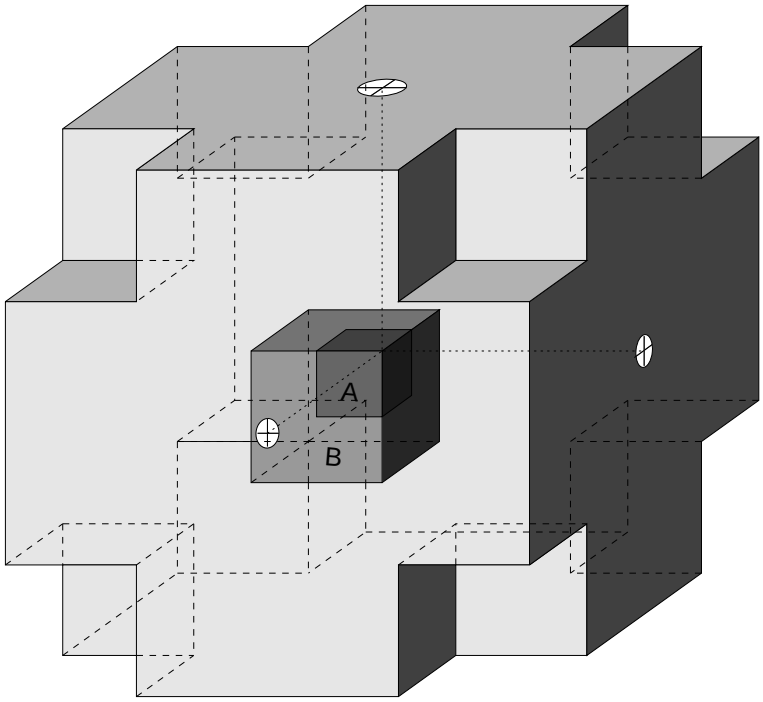

Figure 4.5: Blocks that should be searched in a three-dimensional PMR quadtree. A denotes the base block, while B denotes the parent block.

4.3.6 Nearest Object in Arbitrary Dimensions

There is a theoretical difficulty in conducting nearest neighbor searches in higher dimensions. The amount of space overhead (i.e. of space traversed even though it is outside the search sphere) can grow quite large as the dimensionality of the space increases. This is because the number of "corners" (i.e., the sections of a block near its vertices into which the circumscribed hypersphere cannot reach) grows exponentially with dimension. As a result, the ratio between the volume of a hypersphere and that of its circumscribing hypercube decreases exponentially with dimension. This property of spaces of high dimensions makes searching them particularly time-consuming [Spro91].

Table 4.1 contains the actual values of this ratio for dimensions ranging from 1 to 14. Equation 4.1 is the formula for calculating the volume of a d-dimensional hypersphere of diameter D (from, for instance, [Apos69, pp. 411–412]).

$$V_d(D) = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)} \left(\frac{D}{2}\right)^d \tag{4.1}$$

expanded block list for 3-D

Let P denote the vertex shared by the base and the parent blocks (there is exactly one such vertex);

1. The parent block.
2. All blocks sharing one of the parent block's faces which are incident on P (at most three).
3. All blocks sharing one of the parent block's edges which are incident on P and have not been listed already (at most three).
4. All blocks incident on P that have not been listed already (at most one).
5. All blocks sharing a face with any of the blocks listed in items 1, 2, 3 and 4 (at most 24).
6. All blocks sharing an edge with any of the blocks listed in items 1, 2, 3 and 4, excluding blocks listed in item 5 (at most 24).

All blocks in the expanded list have the same size as the parent block.

Figure 4.6: Algorithm to find the blocks in the expanded search list for a three-dimensional PMR quadtree.

Equation 4.1 can be simplified (avoiding the Γ function notation) if odd and even dimensions are calculated separately. Equation 4.2 can be used for odd dimensions, while Equation 4.3 applies to even ones.

$$V_d^{odd}(D) = \frac{2\pi^{\frac{d-1}{2}}(\frac{d+1}{2})!}{(d+1)!}D^d = \frac{\pi^{\frac{d-1}{2}}}{\frac{d+1}{2}\cdot(\frac{d+1}{2}+1)\cdot(\frac{d+1}{2}+2)\cdot\ldots\cdot d}D^d \tag{4.2}$$

$$V_d^{even}(D) = \frac{\pi^{\frac{d}{2}}}{2^d(\frac{d}{2})!}D^d = \frac{\pi^{\frac{d}{2}}}{4\cdot 8\cdot 12\cdot\ldots\cdot 2d}D^d \tag{4.3}$$

The scheme described in Section 4.3.4 may be extended to an arbitrary number of dimensions provided the list of blocks covering the search sphere can be produced automatically. Tabulating an expanded block list in advance can reduce the inefficiency inherent in high-dimensional searches. Although generating such a list even off-line is a lengthy process, it

Table 4.1: The volume of a d-dimensional hypersphere having unit diameter.

dim	formula	volume	dim	formula	volume
1	1	1.00000	2	$\frac{\pi}{4}$	0.78540
3	$\frac{\pi}{2\cdot 3}$	0.52360	4	$\frac{\pi^2}{4\cdot 8}$	0.30843
5	$\frac{\pi^2}{3\cdot 4\cdot 5}$	0.16450	6	$\frac{\pi^3}{4\cdot 8\cdot 12}$	0.08075
7	$\frac{\pi^3}{4\cdot 5\cdot 6\cdot 7}$	0.03691	8	$\frac{\pi^4}{4\cdot 8\cdot 12\cdot 16}$	0.01585
9	$\frac{\pi^4}{5\cdot 6\cdot 7\cdot 8\cdot 9}$	0.00644	10	$\frac{\pi^5}{4\cdot 8\cdot 12\cdot 16\cdot 20}$	0.00249
11	$\frac{\pi^5}{6\cdot 7\cdot 8\cdot 9\cdot 10\cdot 11}$	0.00092	12	$\frac{\pi^6}{4\cdot 8\cdot 12\cdot 16\cdot 20\cdot 24}$	0.00033
13	$\frac{\pi^6}{7\cdot 8\cdot 9\cdot 10\cdot 11\cdot 12\cdot 13}$	0.00011	14	$\frac{\pi^7}{4\cdot 8\cdot 12\cdot 16\cdot 20\cdot 24\cdot 28}$	0.00004

needs to be done only once per dimension: once completed, it can be applied to any PMR quadtree of the associated dimension.

One way to create an expanded block list is to determine a part of space that is guaranteed to contain all possible search spheres (this can be done since the search radius is bounded a priori), and then to exhaustively test all blocks within it. Any block situated so that it could be part of some search sphere is included in the list.

The difficulty with this approach lies in the sheer size of the resulting list, which at higher dimensions can become prohibitive. The ratio S_d between the volume of the search sphere and that of the parent block provides a lower bound on the size of the expanded block list. Since the union of the listed blocks covers the search sphere, their total volume must be at least as large as that of the sphere. From the fact that the radius of the search sphere is the longest diagonal of the parent block, and hence proportional to the square root of the dimension, and Equation 4.1, an expression for this ratio S_d can be written:

$$S_d = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)}(\sqrt{d})^d \tag{4.4}$$

Using Stirling's approximation for the Γ function,

$$\Gamma(x+1) \approx \sqrt{2\pi x}x^x e^{-x} \tag{4.5}$$

Equation 4.4 can be approximated as

$$S_d \approx \frac{(2\pi e)^{\frac{d}{2}}}{\sqrt{\pi d}} \tag{4.6}$$

which is $\Omega(2^d)$. Moreover, in order to obtain the expanded block list we may have to inspect all the blocks within a cube circumscribing the search sphere. A search sphere of radius $\sqrt{d}$ is circumscribed by a cube whose volume is $2\sqrt{d}^d$, which can pack $O(d^d)$ blocks of parent-block size. Fortunately, the number of symmetries embedded in the geometry of this construction offers several heuristics which drastically reduce both the size of the expanded block list and the amount of computation involved.

The only factor determining whether a block should be included in the expanded block list is its distance from the search point. Euclidean distance is computed by summing squares of coordinate values. The distance function is therefore invariant to changes in the signs of the coordinates (since they are squared) as well as to permutations they may undergo (since summing is commutative). A given set of coordinates can represent several distinct points, all equidistant from the search point, if their signs and order are allowed to change. It is therefore possible to generate a core list, in which each block is representative of many blocks appearing in the the expanded block list (hence the term "expanded"). Specifically, a block B_c in the core list represents any block B_e whose coordinates can be derived from those of B_c by permuting their order and changing their signs. Since any block so derived has the same distance (from the search point) as B_c, if the latter intersects the search sphere so will all the derived blocks. If the blocks in the core list are sorted by their distances from the search point, then this property facilitates the generation of the expanded block list directly in sorted order as well; we simply add to the expanded block list all the blocks derived from a single block in the core list before processing any subsequent core list blocks.

The size of the core list also grows exponentially with dimension, but it is several orders of magnitude smaller than the corresponding expanded list. Table 4.2 compares the size of the core list with that of the expanded block list, as well as with the size of the search sphere (expressed in terms of parent block volumes).

core list for any dimension d

1. $R \leftarrow$ the radius of the maximal search sphere $(= \sqrt{d})$.

2. Nest d loops, one per coordinate; each loop starts with the current value of its predecessor loop, as follows:

```
for ( c_1 = 0 to R) do
   for ( c_2 = c_1 to R) do
      for ( c_3 = c_2 to R) do
                        ⋮
         for ( c_d = c_{d-1} to R) do
            if (∑_{i=1}^{d} c_i^2 < d) then
               add (c_1, c_2, ..., c_d) to the core list
```

3. stop.

Figure 4.7: The list of PMR quadtree blocks to traverse to find nearest objects in the case of three dimensions.

Figure 4.7 provides the details of the algorithm used to derive the core list. It uses a stack of nested for-loops whose depth is determined at run time. In practice, this can be

Table 4.2: A comparison of the size of the core and expanded block lists for various dimensions. The volume of the search sphere (in units of parent block volume) is provided as a benchmark for the shortest expanded list theoretically possible.

dim	core list	expanded block list	volume of search sphere
1	1	2	2.00
2	2	12	6.28
3	3	56	21.76
4	4	240	78.95
5	6	1152	294.25
6	8	6336	1116.23
7	10	35968	4287.69
8	12	196352	16624.5
9	15	1031168	64924.6
10	19	5384192	255016
12	27	1.569×10^8	3.987×10^6
14	38	4.751×10^9	6.317×10^7
16	50	1.419×10^{11}	1.010×10^9
18	67	4.296×10^{12}	1.629×10^{10}
20	87	1.313×10^{14}	2.643×10^{11}
25	156	6.867×10^{17}	2.854×10^{14}
30	265	3.665×10^{21}	3.145×10^{17}
35	422	1.984×10^{25}	3.513×10^{20}
40	648	1.081×10^{29}	3.963×10^{23}
45	963	5.975×10^{32}	4.507×10^{26}
50	1394	3.312×10^{36}	5.156×10^{29}
60	2742	1.032×10^{44}	6.845×10^{35}
70	5037	3.261×10^{51}	9.214×10^{41}
80	8779	1.041×10^{59}	1.253×10^{48}
90	14671	3.349×10^{66}	1.717×10^{54}
100	23672	1.084×10^{74}	2.368×10^{60}

achieved by a recursive function, where each invocation represents one loop. It is also useful to include in each loop (or in the single recursive function) a test to check if the sum of the squares of all the coordinates fixed so far (in all enclosing loops, including the present one) already exceeds the space dimension, in which case we break out of the present loop. This test, although not necessary for correctness, cuts the running time by 99.9% in dimension 70, for example.

4.4 Windowing

Another useful operation supported by the PMR quadtree is windowing (also known as range query), which is concerned with determining which objects are included in a given region of space. The semantics of the query can either be exclusive, selecting only objects completely contained in the window region, (Figure 4.8b), or inclusive, where all objects having a non-empty intersection with the query region qualify (Figure 4.8c). In the inclusive case there is the further choice of reporting objects only partially in the region in their entirety or clipping them to the region (Figure 4.8d).

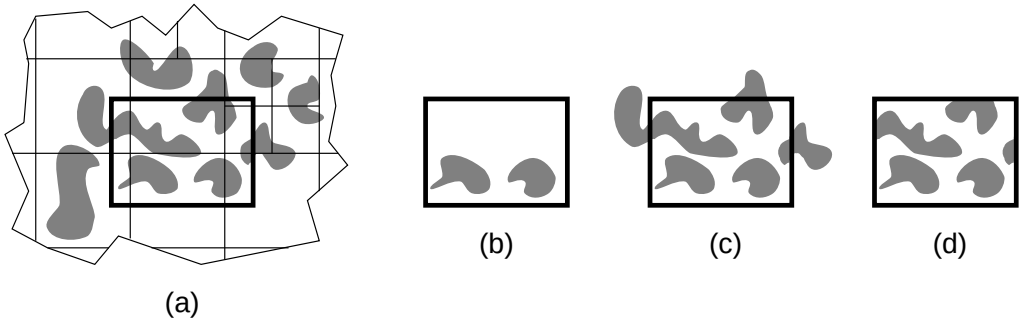

Figure 4.8: Various ways of interpreting a windowing operation. (a) A section of a PMR quadtree containing a query window. (b) Exclusive: only objects completely contained in the window are selected. (c) Inclusive: all objects intersecting the window are output in their entirety. (d) Clipped: objects not completely contained in the window are clipped.

The mechanics of any of the windowing algorithms depends on the PMR quadtree well as on the properties of objects included in the query window. To determine whether an object is completely contained inside the query window, or to clip it against that window if it is not (operations called for by the inclusive and clipped versions of the algorithm), a complete description of an object is required. We confine the present discussion to that part of the algorithm that is common to all three approaches, which pertains only to the PMR quadtree and is independent of the actual objects involved.

4.4.1 Previous Work

Practical PMR quadtrees are large enough to be disk resident, so each block read may, in reality, involve a disk access. A good algorithm, then, would attempt to minimize the number of PMR quadtree blocks accessed and the number of times each one of them is read. Ideally, blocks having non-empty intersections with the window should each be read once, and other blocks not at all.

An algorithm having such optimal behavior is described in [Aref92c]. It is limited to rectangular query regions that are axis-aligned (i.e., rectangles whose edges are parallel to the major axes). It uses the region quadtree decomposition ([Same90a]) of the window itself. The blocks that comprise this decomposition window drive the search for blocks in the underlying quadtree where the objects are stored.

The algorithm in [Aref92c] is therefore not limited to a specific quadtree type or a particular query. When focusing on PMR quadtrees, however, several assumptions can safely be made that are not true in general. We revisit this algorithm with these assumptions to yield simpler code, without sacrificing the benefit of accessing blocks that overlap the window once and others not at all.

The algorithm [Aref92c] employs three support data structures in addition to the PMR quadtree being windowed:

1. A quadtree decomposition of the window, implemented as a linked list of blocks.

2. An active border to keep track of the part of the window already processed. The active border is implemented with two linked lists representing the western and eastern boundaries of the area covered. Note that the northern border is immaterial and the southern one is taken care of directly in the algorithm.

3. An unspecified mechanism for returning the result of the window query back to the user.

The algorithm presented here provides a mechanism for returning the result (another PMR quadtree). It also removes the restrictions on the shape and alignment of the query window.

4.4.2 The Algorithm

Algorithm window described here accepts a collection of objects arranged in a PMR quadtree, labeled source, and a region of space serving as a window represented in any suitable way. The algorithm eventually produces the subset of objects which intersect the window. The window may have any shape and the algorithm is applicable to spaces of any dimensionality.

The role of returning the result of the window query is carried out in the present implementation by another PMR quadtree, labeled result. Although convenient from an implementation standpoint, this places an additional requirement on the algorithm which is not shared by the one in [Aref92c]. In particular, in addition to listing the objects found in the window, our algorithm window must also provide the appropriate PMR quadtree decomposition of the space that they occupy. Fortunately, this decomposition is closely related to that of the source quadtree, as explained below, so that the overhead incurred is usually small.

Observation: if block $s \in$ source and parent(s) $\subseteq$ window then $s \in$ result. The block decomposition of any non-degenerate subtree of the result quadtree is identical to that of the corresponding subtree in the source quadtree whenever its root is completely contained within the query region. This follows from the property of the PMR quadtree discussed in Section 4.3.2 which states that the number of objects in a non-terminal PMR quadtree block is at least as large as the threshold. If the parent of a block is completely contained within the window, then all the objects it contains are included in the result. Since block decomposition in the PMR quadtree is driven only by the distribution of objects

within it, the conditions which led to the particular decomposition of the source quadtree for this block are also present in the corresponding result quadtree. Blocks of this nature can account for the bulk of the query window if it is large in terms of the sizes of the source quadtree blocks that it overlaps.

The algorithm scans all the blocks (terminal and non-terminal) that are found in a tree representation of the source quadtree. In reality, it is not essential that a tree representation be used; this is assumed here only for the purpose of facilitating the description of the algorithm's operation. The blocks of the source quadtree are classified as follows:

- blocks completely outside the window,
- blocks completely inside the window, and
- the remaining blocks: those which intersect the window boundary.

Blocks residing outside the window can be safely ignored, since nothing they (or their descendants) may contain can be relevant to the result. Non-terminal blocks completely inside the window, according to the observation above, serve as roots of subtrees which are replicated in the result verbatim. Terminal blocks inside the window and any block intersecting the boundary require further consideration: the former since their presence in the result is not guaranteed, the latter because they may contain objects that are not in the window at all.

Specifically, the algorithm performs a top-down depth-first traversal of the source quadtree. Any block falling outside the window is discarded. Any non-terminal block inside the window is copied, along with all its descendants, into the result quadtree. Boundary non-terminal blocks are decomposed and this classification is applied recursively to their children.

Eventually, only terminal blocks remain. Those outside the window are discarded. The rest are examined for the objects they contain, which are placed in an auxiliary list. Blocks completely inside the window contribute all their objects to the list, while the objects in boundary leaves must be tested individually, since such leaves may conceivably contain objects that are outside the window.

Once the traversal of the source quadtree is done, a partial result quadtree has been generated as well, consisting of the subtrees rooted in blocks completely inside the window. In addition, any objects that may still be missing from the result are available in the object list mentioned above. Inserting the objects on the object list into the result quadtree completes the production of the desired result. It is assumed that the insertion methods of both the result quadtree and the object list eliminate duplicates, so that an object residing on the boundary of two or more blocks does not end up being inserted multiple times.

Pseudo code for the top-level control structure of the window algorithm is given in Figure 4.9 and its recursive component, window-block, in Figure 4.10. The algorithm presented here has the semantics of inclusive windowing, as depicted in Figure 4.8c. However, simple changes can accommodate the other variants. Changing step 5 of window (Figure 4.9) to clip each object against the window before it is inserted into the result quadtree will result in the clipped version of the query, as in Figure 4.8d. Objects introduced into the result

via direct copy cannot require clipping, at least not in that part, since only blocks that are completely inside the window are copied. Likewise, if step 1 of Figure 4.10 is changed to read "is contained in" instead of "intersects", the algorithm should produce the exclusive windowing version, as in Figure 4.8b.

Build a PMR quadtree From Objects in Window

1. $L \leftarrow$ an empty list of objects.
2. $R \leftarrow$ an empty PMR quadtree with the parameters of the source quadtree.
3. $B \leftarrow$ the root block of the source quadtree.
4. window-block(B, L, R) (see Figure 4.10).
5. **for** each object O in L **do**
   Insert O into R.
6. Free L, return R and stop.

Figure 4.9: Algorithm window: find all the objects in a PMR quadtree intersecting a given window.

Figures 4.11 through 4.15 demonstrate the operation of the window algorithm by going through an example. Starting with the object set and the window shown in Figure 4.11a, each of the figures (except the first and last) shows the operation at one level of the tree. The blocks being considered at each level are marked by the heavy lines, while the thin lines show the block decomposition of the underlying source quadtree. The portions of the space the algorithm has finished processing are marked by a hatched pattern. The figures are split into three columns: the left column displays the source quadtree and the status of its processing; the middle column shows the increments made to the result quadtree; the objects being accumulated on the object list are shown in the left column.

After considering the root block (Figure 4.11b) and finding that it cannot be classified as either inside or outside the window, the algorithm turns to its children (Figure 4.11c). Level 1 of the tree is still too coarse, so another level is attempted (Figure 4.12d). Here some blocks can already be classified; those on the right are outside the window and are discarded (Figure 4.12e). Two blocks are found inside the window (Figure 4.12f) and are copied to the result (Figure 4.12g). Objects in the leaves encountered (top right and bottom left of Figure 4.12h) are stored on the object list (Figure 4.12j).

The process is repeated for levels 3 and 4 in Figures 4.13 and 4.14, respectively. In level 4 no blocks were found to be completely inside the window so no additions were made to the result quadtree at that step.

```
                Get Windowed Objects in Block B

1. if B intersects boundary of window then
   begin
2.    if B is a leaf block in the source quadtree then
      begin
3.       for each object O in B do
4.          if O intersects the window then
               Add O to the object list L.
      end
      else
5.       for each child C of B do
            window-block (C, L, R).
   end
6. else if B is contained in the window then
   begin
7.    if B is a leaf block in the source quadtree then
8.       for each object O in B
            Add O to the object list L.
      else
9.    incorporate(B, R) (see Figure 3.16).
   end
```

Figure 4.10: Algorithm window-block which extracts the objects in the block that are also in the query window.

When processing of the source quadtree is complete, we have obtained the partially constructed result quadtree and the object list. Figure 4.15 repeats the relevant panels from the previous figures and shows the products of their accumulation. Finally, the objects in the list are inserted into the result quadtree to produce the final result in Figure 4.15v.

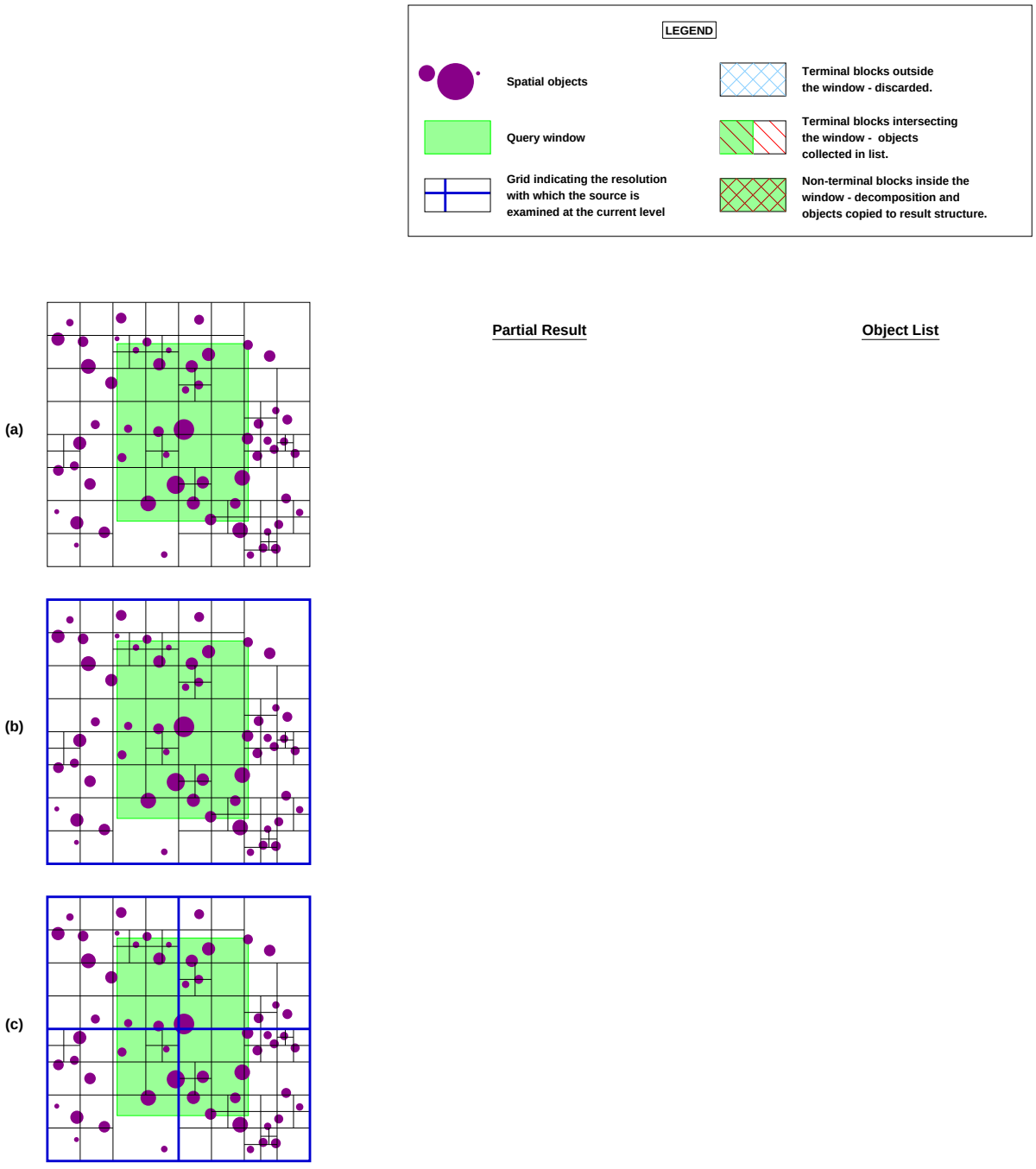

Figure 4.11: A walk through the operation of window at levels 0 and 1. The PMR splitting threshold is 2. Notice the use of solid decomposition lines to indicate the level of the PMR quadtree that is being examined. (a) The original dataset and query window. (b) Level 0—inspecting the root block. (c) Level 1.

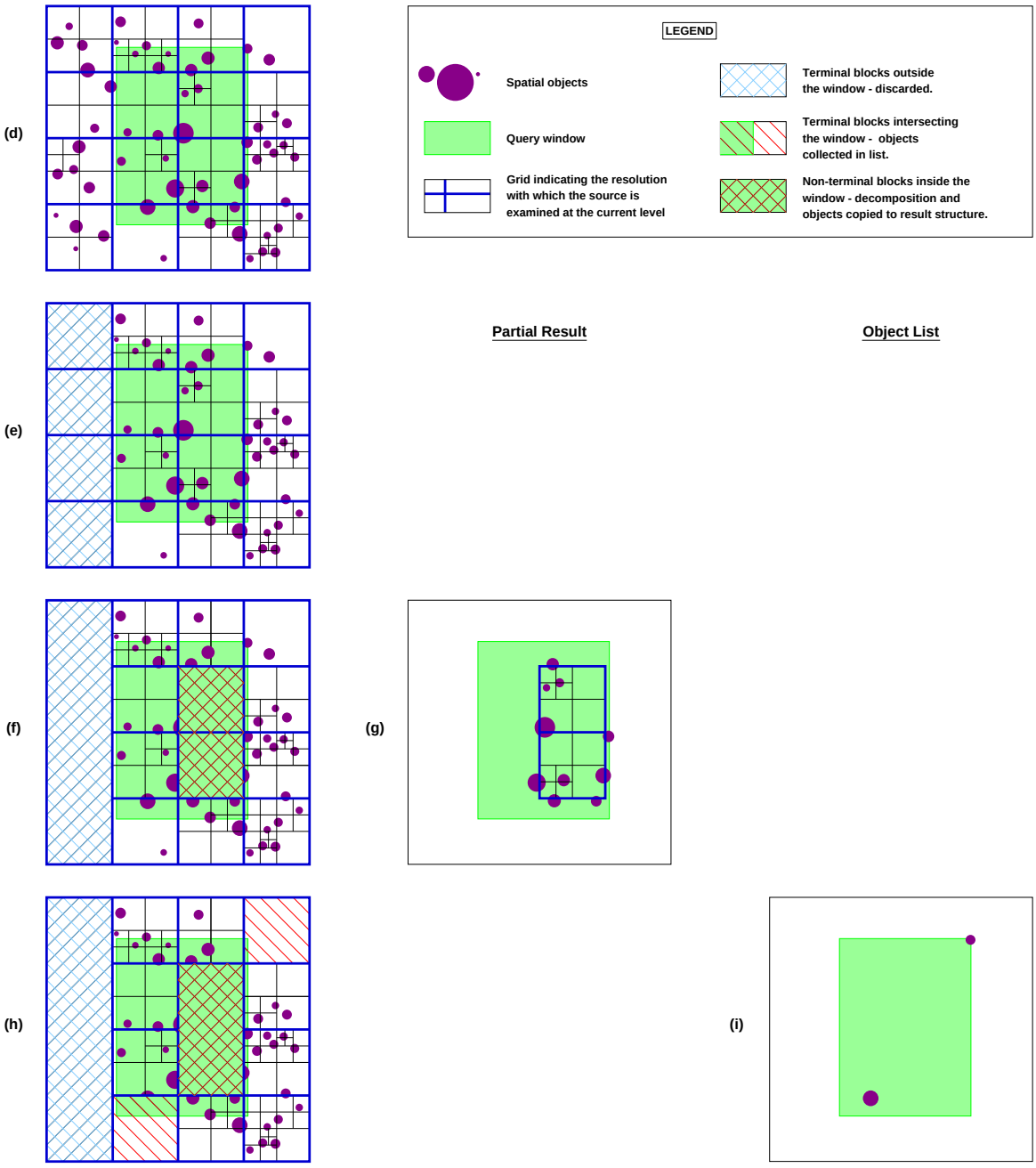

Figure 4.12: A walk through the operation of window: level 2. (d) Level 2 grid of blocks to inspect. (e) Objects in terminal blocks outside the window are discarded. (f) Non-terminal blocks inside the window are copied to the result. Both the objects and the underlying decomposition are recorded. (g) The information extracted in (f) is copied to the result. (h) Terminal blocks intersecting the window are examined. Any objects they contain which themselves intersect the window are copied to the object list. (i) The objects found in (h) are added to the object list.

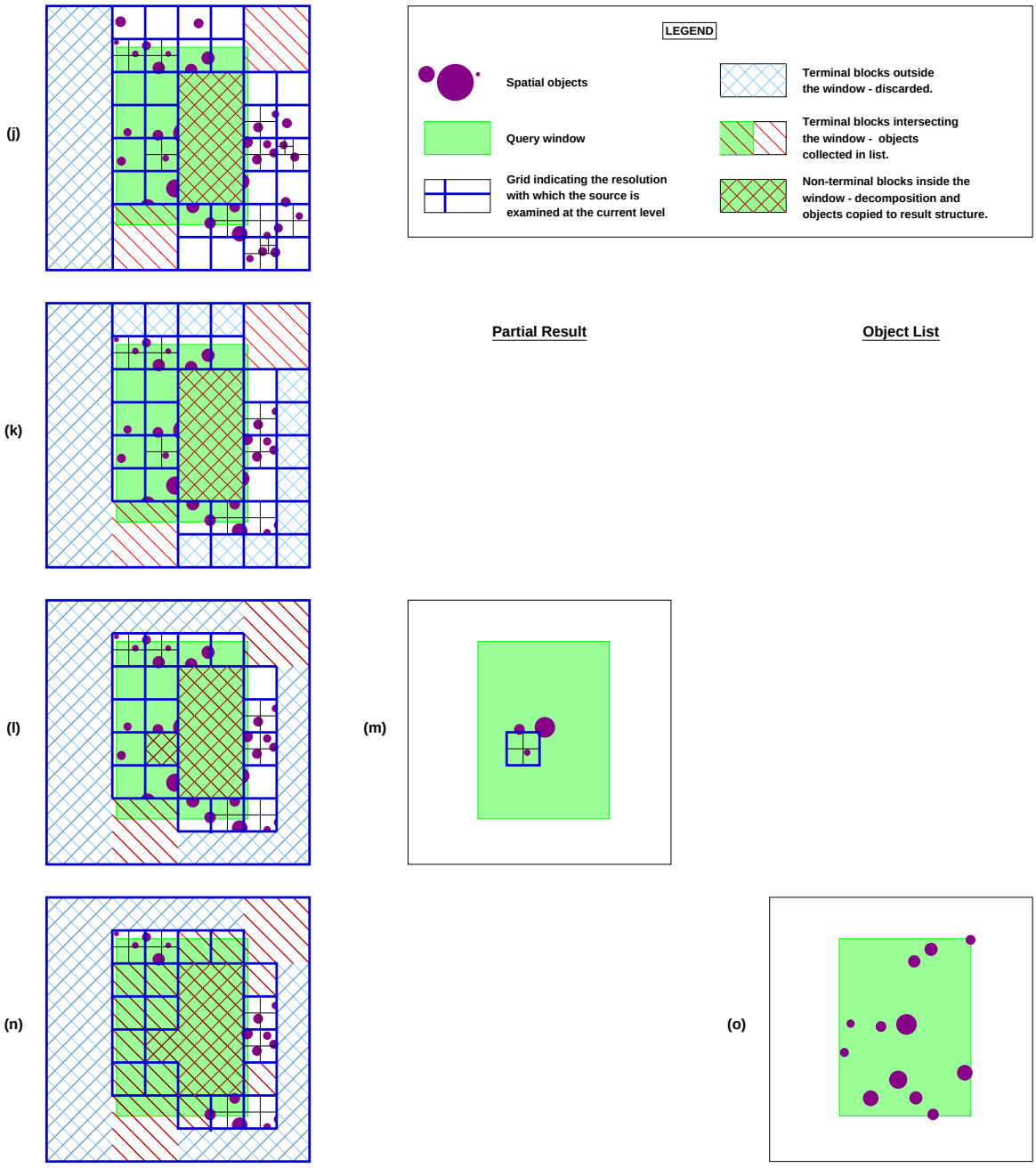

Figure 4.13: A walk through the operation of window: level 3. (j) Level 3 grid of blocks to inspect. (k) Objects in terminal blocks outside the window are discarded. (l) Non-terminal blocks inside the window are copied to the result. Both the objects and the underlying decomposition are recorded. (m) The information extracted in (l) is copied to the result. (n) Terminal blocks intersecting the window are examined. Any objects they contain which themselves intersect the window are copied to the object list. (o) The objects found in (n) are added to the object list.

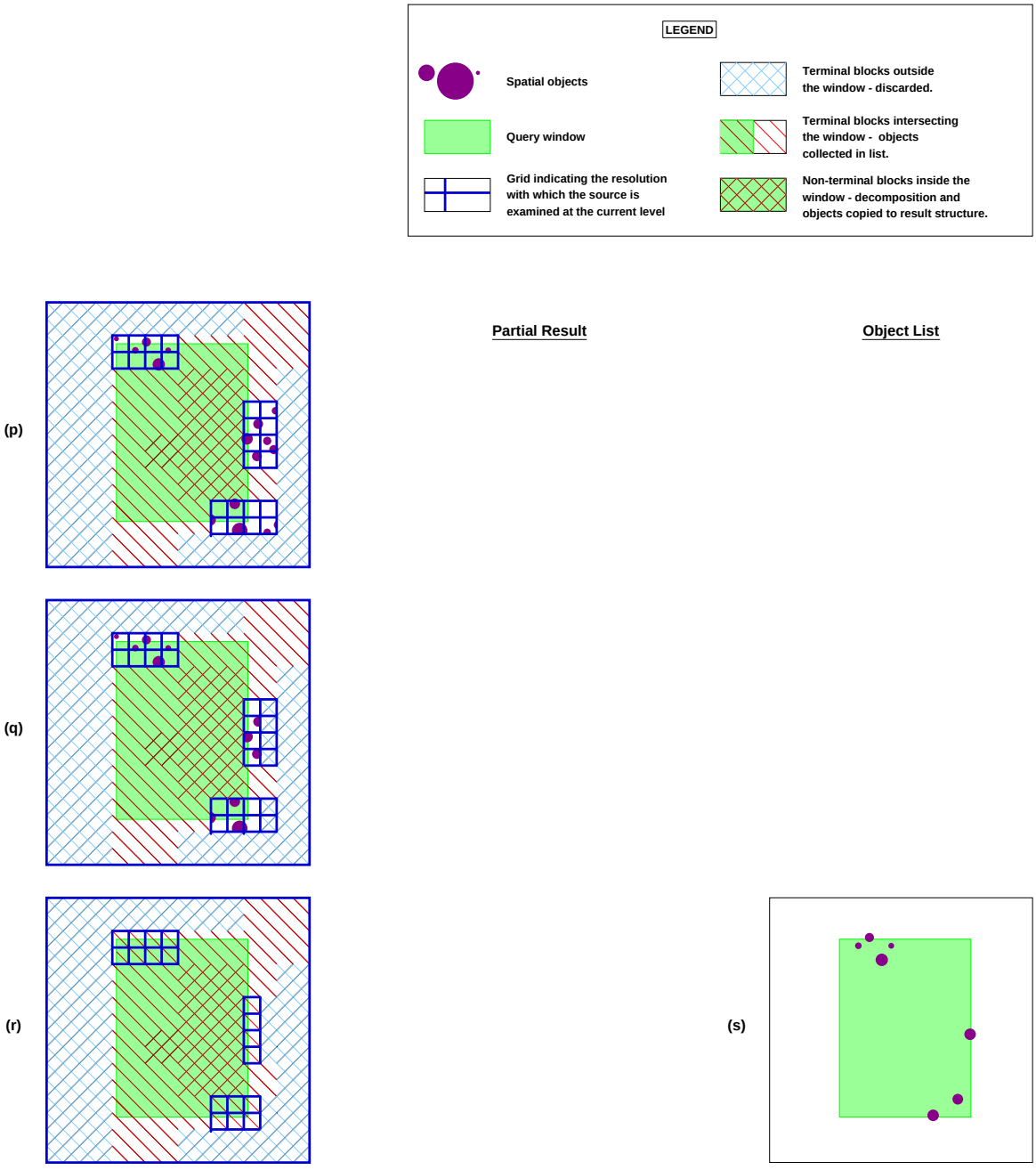

Figure 4.14: A walk through the operation of window: level 4. (p) Level 4 grid of blocks to inspect. (q) Objects in terminal blocks outside the window are discarded. Note that there are no more non-terminal blocks inside the window at this stage, so the step of copying them into the result is skipped. (r) Terminal blocks intersecting the window are examined. Any objects they contain which themselves intersect the window are copied to the object list. (s) The objects found in (r) are added to the object list.

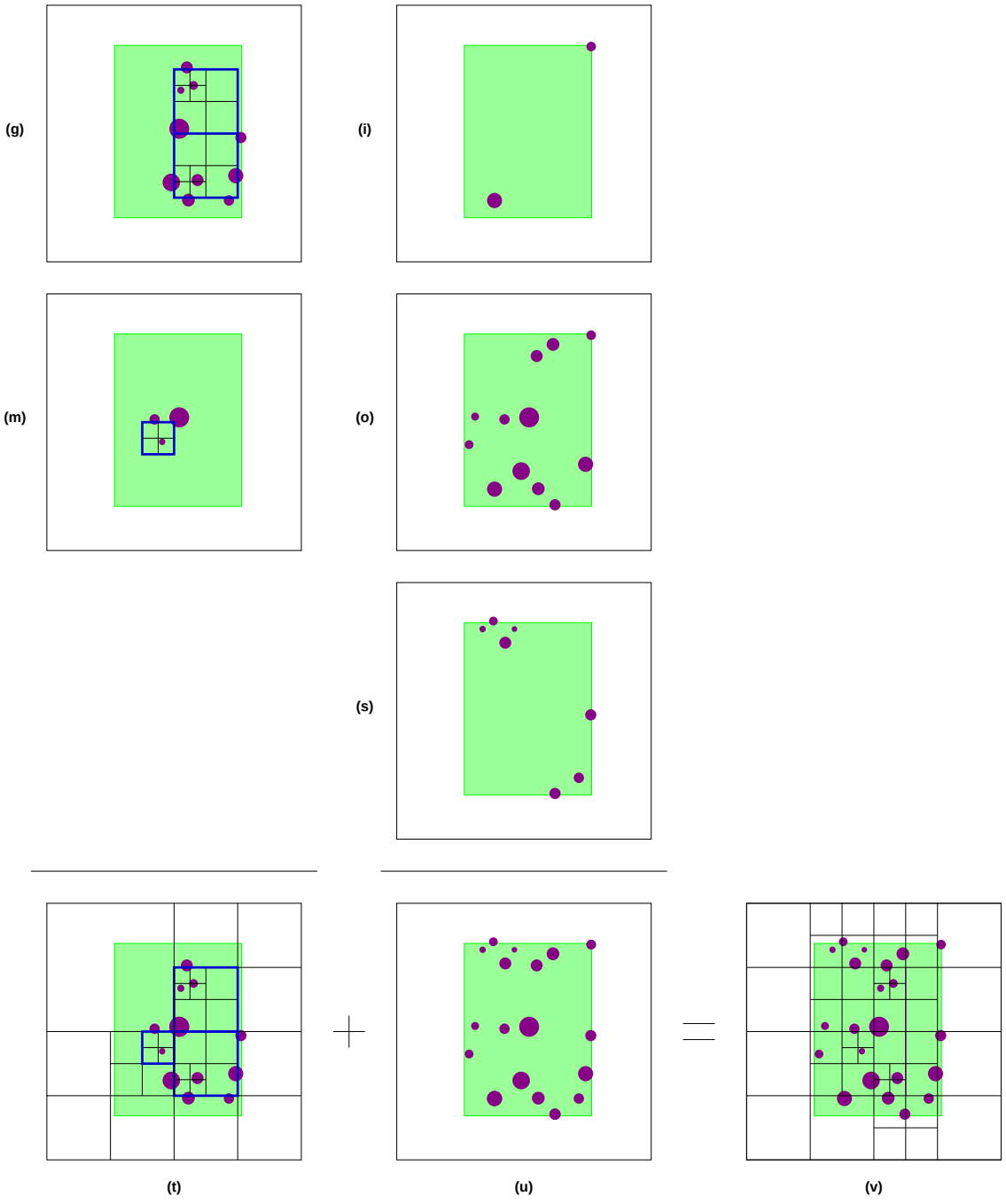

Figure 4.15: A walk through the operation of window: conclusion. Note that panels (g), (i), (m), (o) and (s) are just copied from the previous figures. (t) The result quadtree after processing the source quadtree but without the elements in the object list. (u) The object list. (v) The final query result as produced by the union of (t) and (u).

Chapter 5

Irregular Triangulations and Quadtrees

5.1 Motivation

The triangulations used in Chapter 3 for surface modeling were in registration with the underlying quadtree; the triangles were constrained in shape, size and placement so they neatly fit into the quadtree blocks. It is also possible to use quadtrees for arbitrary triangulations.

A common surface model utilizing arbitrary triangulations is the Triangulated Irregular Network or TIN [Peuc75]. TINs are described in Section 1.3.3, but for the purpose of the present discussion, the particulars of the TIN utilized are secondary to the application made of the quadtree structure. For simplicity, TINs representing Delaunay triangulations [Prep85] of DTMs are used.

The lack of registration of the triangle boundaries with those of the blocks of the underlying quadtree does make a difference. The restricted quadtree of Chapter 3 is inapplicable as it has no support for triangles spanning quadtree blocks. The region quadtree [Same90a], in which a region is described by enumerating its interior at some resolution, suffers from all the difficulties associated with rasterizing vector-based entities (triangles in this case) due to the information lost in the process.

We propose to use the PMR quadtree described in Chapter 4, which can organize arbitrary objects in space. This chapter describes this adaptation of PMR quadtrees to surface modeling.

5.2 Implementation

As described in Section 4.2, the PMR quadtree is capable of organizing any collection of objects in space. Two aspects of this organization, however, are deliberately left open and must be determined for each collection:

- The decomposition rule: under what circumstances should a PMR quadtree block be decomposed into its descendantsΓ
- Object-block intersection: when do a given object and a given PMR block intersectΓ

In practice, either aspect is too complex to be conveyed to a generic implementation of a PMR quadtree using a single parameter, or even a set of parameters. In the present

implementation, user-defined functions are used to describe the desired behavior in both cases.

5.2.1 Decomposition Rule

A PMR quadtree implementation bases its decomposition rule on object density, helping to control the population of objects associated with any one block in the structure. This enables PMR quadtree blocks to be realized in a limited amount of space, and also caps the time required to find an object within a block. A single block, then, can be implemented in both time and space complexities of $O(1)$.

The bucket PMR quadtree sets a fixed limit to the number of objects a block may contain, and splits any block containing more than that number of objects, known as the bucket capacity. This would achieve both goals stated above: a fixed amount of storage and a fixed block searching time. In some situations, however, such a rule may be problematic. For example, consider a collection of line segments, m of which share a common endpoint p. Assume further that this collection is stored in a bucket PMR quadtree using a bucket capacity of c. The PMR quadtree block that eventually contains point p must also contain all the m line segments incident on p. If $c < m$, that block violates the decomposition rule and no amount of splitting will rid the structure of it.

The PMR quadtree, as originally described in [Nels86a], overcomes this problem by requiring that the implementation support overpopulated blocks. Also, instead of a bucket capacity it uses a concept of a splitting threshold defined as follows: a block that exceeds its splitting threshold (as a result of an insertion) is split once and only once. No further action is taken even if some of the resulting sub-blocks remain overpopulated. This avoids the excessive decomposition resulting from the PMR quadtree's futile attempt to localize inseparable objects, such as the line segments incident on point p in the example above.

Collections of line segments may or may not have many occurrences of such inseparable objects. However, this situation exists for virtually all triangles of a triangulation; other than the triangles on the perimeter of the area described, every triangle is incident on at least six more triangles. To help further reduce unnecessary splitting we propose to regard all touching objects as a single object when counted towards meeting the splitting threshold. Only objects that have no edges or vertices in common are regarded as distinct in this context.

Consider Figure 5.1a where three touching objects are depicted. With a splitting threshold of 2 and no consideration for touching objects, decomposition will proceed until blocks of the smallest size possible are generated. The block within which the objects meet will inevitably contain three labels, more than permitted by the splitting threshold. On the other hand, when touching objects are counted as one, as in Figure 5.1b, only one big block is generated, which, admittedly, contains more than the splitting threshold of labels, but as we have seen, this cannot be avoided.

5.2.2 Object-Block Intersection Rule

Another choice the user of a PMR quadtree must make has to do with the object-block intersection conditions, which determine when objects need to be associated with a PMR

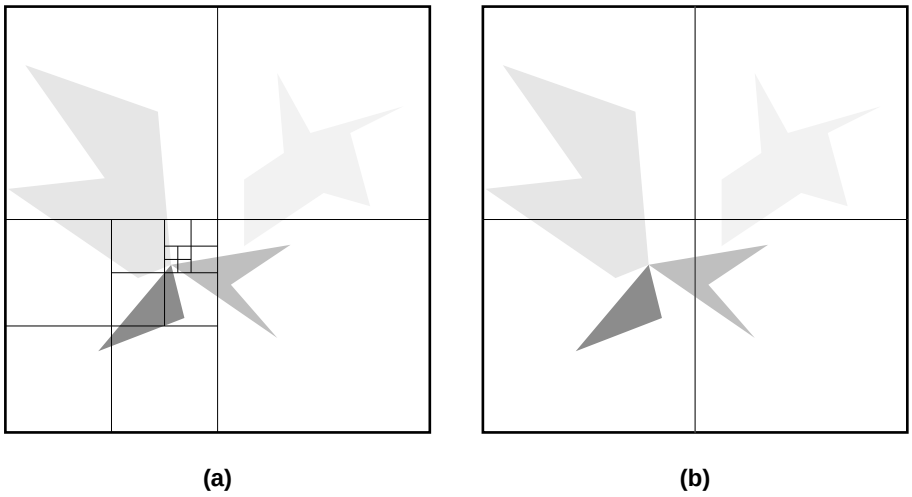

Figure 5.1: Touching objects in a bucket PMR quadtree. (a) No special consideration; (b) touching objects considered as one. The tree depth is 5 and the splitting threshold is 2 in both cases.

quadtree block. Again, difficulties may arise around boundary situations. A common approach to quadtrees in general posits that two of a block's borders, say north and west, are closed, whereas the other two are open. Such a scheme gives ownership of any point in space to one and only one block, a desirable property.

However, in its application to TINs, blocks having open boundaries may cause problems. A block which has only the vertex of a triangle on its border will not be associated with that triangle if the border in question is an open one. If the neighbors of the triangle need to be found, care must be taken to look for them in the block which contains the vertex in question on its closed boundary. This may complicate the design of algorithms and is an opening for programming errors.

The disadvantage of having all of the block's boundaries closed is that objects that otherwise would be associated with one block must now be associated with two. However, objects are invariably associated with multiple blocks in a PMR quadtree because arbitrary objects are not likely to be placed neatly inside blocks' interiors. Most implementations which use PMR quadtrees to store complex objects do not store the objects themselves inside the tree, but store only pointers to them to avoid the data duplication that would otherwise result. Since the occurrence of vertices on boundaries is in general a relatively rare event, the small amount of added storage does not seem significant.

Chapter 6

Field of View: a Test Case

6.1 Field Of View

An interesting application of surface models is the automatic determination of visibility: identifying the parts of the surface that are visible from a given a point. The need for such determination arises both when visibility is desirable as well as when it is not. For example, placing broadcasting towers or surveillance posts can be economized by finding a minimal set of locations from which every point on the surface can be seen. Conversely, stealth navigation involves finding a path not visible from any known observation point. Both of these and many similar tasks can make use of automated field of view generation.

The field of view algorithm (field-of-view) presented here is sufficiently general to support any polyhedral surface model. Since the algorithm is introduced specifically in order to test the two approaches to surface modeling discussed in this work, it was important not to rely on any idiosyncrasies peculiar to only one of them. The choice of algorithm and implementation details reflect this by maximizing encapsulation, not performance. As a result, field-of-view is not necessarily the fastest field of view algorithm possible, but it can be used with both the RQT and QTN quadtree surface representations with no adaptation.

Determining the field of view may be regarded as the converse of the process involved in producing an image of the surface. The purpose in the graphical display case is to generate an image, keeping no track of the individual surface facets contributing to it. When generating a field of view, it is precisely the extent of those facets that is recorded, while an image is produced, if at all, only as a byproduct.

An often-used graphical display algorithm is the depth sort algorithm [Newe72]. (A simpler variant of this algorithm is called the "painter's algorithm" [Fole90, p. 673].) In this algorithm, the components of the scene are projected onto the display device in decreasing distance from the viewpoint. Nearer objects are painted after, and therefore over, more distant ones, simulating the visualization process as it occurs in reality.

In determining the field of view, a reverse process can be used. Beginning with the surface facets nearest to the viewpoint, the "shadow" each casts is accumulated. As more distant facets are considered, only the part that is not obscured by the collective shadow is viewable. The main difference in processing is the order in which the surface facets are sorted: farthest first while displaying, nearest first in field-of-view.

This sorting utilizes the special capabilities of the surface models. It is therefore done

differently in the cases of the RQT and the QTN quadtrees. Section 6.4 delves into the details of the sort step of the algorithm.

Another difference between field-of-view and the depth sort algorithms is that while the result of the graphic display can conveniently be integrated on the display device itself, there is nothing that can naturally play this role in the case of field-of-view. A special data structure, horizon, is devised for this purpose and is described in Section 6.3. In line with the philosophy of encapsulation stated above, most of the computation involved in calculating the field of view is done inside this horizon structure. The field-of-view algorithm is described in pseudo code in Figure 6.1.

Field of View

1. $FOV \leftarrow$ an empty field-of-view.
2. $H \leftarrow$ an empty horizon.
3. Determine the smallest polygon P made of surface facets surrounding the viewpoint (Section 6.2).
4. Use P to initialize H and FOV.
5. **for** each facet F in sorted order (Section 6.4) **do**
   begin
6. $V \leftarrow$ visible part of F (Section 6.3).
7. Update H to reflect the impact of F (Section 6.3).
8. $FOV \leftarrow FOV \cup V$.
   end

Figure 6.1: Algorithm field-of-view: determine the part of the surface visible from a given viewpoint.

Finally, results of test runs made with actual surfaces are presented in Section 6.5. Surface data was obtained from the USGS representing several locations in the US. Both relatively flat (east coast) and rugged (Rocky Mountains) terrain examples are included.

6.2 Priming

The first step of the algorithm involves the initialization of the field of view and horizon structures. Assuming that the viewpoint is somewhat elevated above the surface, all the facets that are incident on it are completely visible. Consequently, there is no need for occlusion

calculations when considering these facets. They are added to the result of field-of-view without further processing. They are also used to initialize the horizon structure. The smallest polygon surrounding the viewpoint that is made of surface facets is found and used to form the initial horizon.

In the simple case the viewpoint is inside one of the facets, as is shown in Figure 6.2a. The facet containing the viewpoint can then serve to form the initial horizon. If the viewpoint is situated on an edge between two facets (Figure 6.2b) or on a vertex of the triangulation (Figure 6.2c), some additional processing is called for. To form a polygon to which the viewpoint is internal, the edges of all the surface facets incident on the viewpoint are combined into a list. The desired polygon is formed from all the edges in the list that are not themselves incident on the viewpoint. The polygons generated in each case are marked by heavy lines in Figure 6.2.

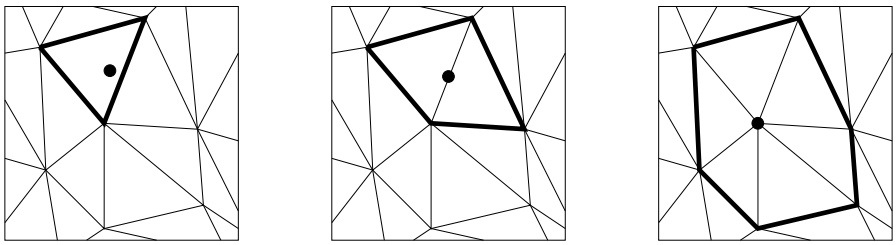

Figure 6.2: The possible relationships between the viewpoint and the surface facets incident on it. (a) The viewpoint is interior to a single facet. (b) The viewpoint is on an edge between two facets. (c) The viewpoint is on a vertex, incident on many facets.

In practice, the algorithm uses the fact that any edge incident on the viewpoint must appear in the list twice.[1] This is because any edge in the surface rendition belongs to two facets, those on each of its sides. When all the edges are placed in a list, each one of these facets contributes an instance. Consequently, the algorithm removes all the edges that have multiple appearances in the list and retains only the singletons. The list is then sorted to form the desired polygon.

The next step of the algorithm (Section 6.3.2) requires the existence of a complete horizon, one that provides an elevation value in every direction from the viewpoint. Since the viewpoint is internal to the initial horizon polygon, any ray emanating from the viewpoint must intersect one of its edges. Hence the projection of those edges onto the wings (see Section 6.3.1) forms a complete horizon.

Once the initial horizon is in place, the viewpoint is external to all subsequent surface facets. The projection of a surface facet onto the wings extends, therefore, only a single wing, or two if it is situated somewhere along the axes bisectors. In extreme cases, facets that are both large and close to the viewpoint may extend three wings; see facet 3 in Figure 6.3.

[1]This argument assumes the viewpoint is not on the very edge of the map. However, in case it is, the edge it is on must be included even though it is incident on the viewpoint, so the algorithm presented will still perform correctly.

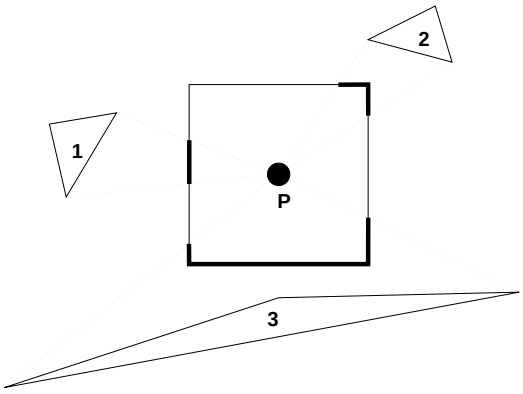

Figure 6.3: Possible combinations of surface facet distance and size, resulting in the projection being confined to a single wing (1), two wings (2), or three wings (3).

6.3 A Data Structure For Horizon Modeling

As stated above, the purpose of the horizon data structure is to facilitate incremental field of view generation. As each surface facet is presented to the horizon data structure, the latter computes which parts of this facet, if any, are visible. It also integrates the impact this surface facet has with that of all the previous ones, preparing the structure for the next surface facet. For the structure to actually perform correctly, however, it is essential that no surface facet be presented if it can possibly obscure a facet that has already been presented. The responsibility for sorting the surface facets lies with the user of the horizon.

6.3.1 The Wings

In order to represent the horizon, the structure employs a screen onto which the edges of the facets of the surface model are projected. These projections delineate, in each direction, the highest line of sight that still grazes the part of the surface seen so far. Ideally, a unit sphere about the viewpoint would provide the least distorted image of the horizon. However, projecting onto a curved surface is computationally difficult, so a flat screen was chosen instead. In fact, in order to avoid potential singularities, four flat screens, called wings, are used, one in each principal direction at unit distance from the viewpoint (Figure 6.4). The horizon is accumulated in the form of four contiguous lists of line segments, one for each wing.

Since the wings are at unit distances from the viewpoint, each wing is two units wide. If a Cartesian coordinate system is constructed with the viewpoint as its origin, the north and south wings are parallel to the x axis, spanning the range $[-1, 1]$. The east and west wings span the same range along the y axis. These bounded ranges are the reason four wings are used. All wings span all values of z, but unless the viewpoint is positioned at the very edge of a cliff, these values are bounded as well.

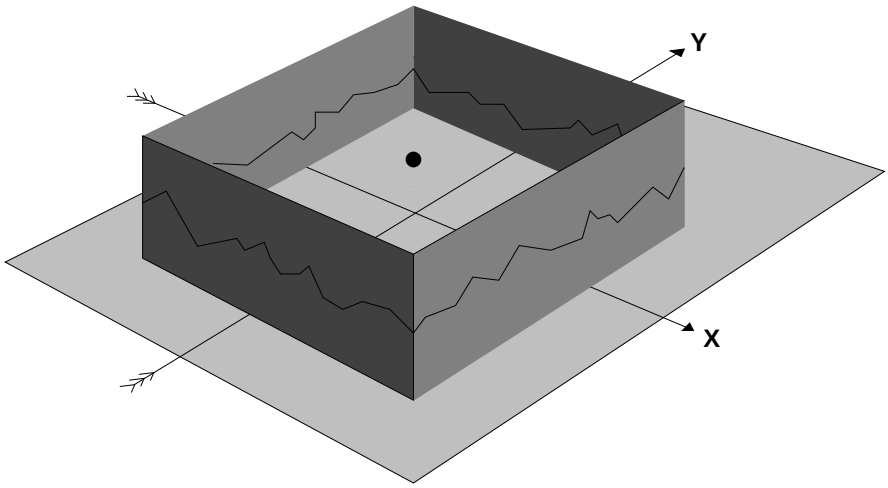

Figure 6.4: The four wings onto which surface facets are projected, accumulating the horizon. The black dot in the center represents the viewpoint.

6.3.2 Processing a Surface Facet

Processing a facet involves two related activities. The first determines what part of the facet, if any, is visible from the viewpoint. The second calculates the shadow the current facet casts on those farther away. The two results are clearly related: a facet casts a shadow if and only if some of it is visible. The shadow is used to update the internal representation of the horizon to reflect the impact the current facet has had, thus preparing the structure for the processing of the next facet. The calculated visible portion of the facet is returned to the caller. The process is described in greater detail in Figures 6.5 through 6.8.

First, the current facet, whose location and orientation are arbitrary, is projected onto the appropriate upright wing (Figure 6.6). If the facet's projection extends two or three wings (see Figure 6.3), each affected wing is processed in turn.

Next, the facet's wing projection is compared with the current state of the horizon on that wing. If any part of the projection protrudes above the horizon line, the polygon(s) bounded by the horizon and the parts of the projection that are above it are found (Figure 6.7).

To update the horizon line, the edges of these polygons are added to it, to form the new horizon. This maintains the horizon's property of delineating the highest lines of sight in all directions.

The polygons are also used to determine the visible part of the current facet. Each is projected back onto the plane of the facet, defining the parts of its facet that are visible (Figure 6.8).

In practice, the algorithm uses several heuristics to avoid redundant tests. A facet of the model facing away from the viewpoint (i.e., if the viewpoint is below the plane the facet is in) is not considered, for example. It is assumed that the surface is continuous and any edge of such a facet, if indeed visible, must belong also to a facet that is directed towards the viewpoint, and will eventually be registered when that other facet is processed.

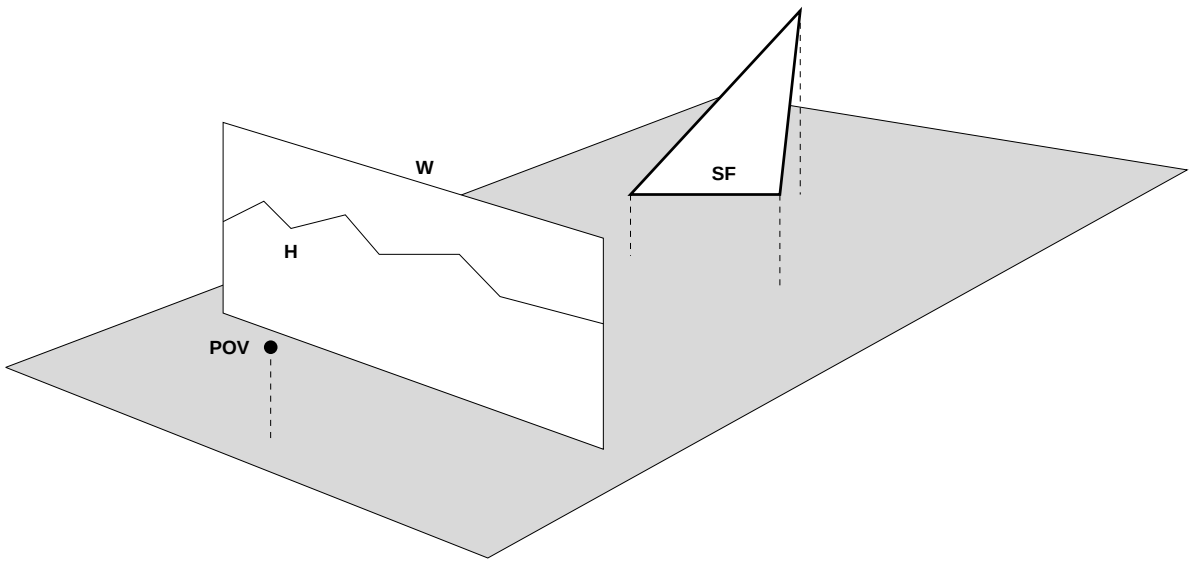

Figure 6.5: A walk through the operation of horizon. (1) Initial state, being presented with a new surface facet. POV—the viewpoint. W—the projection plane. For simplicity, only one of the wings is shown. H—current horizon. SF—a surface facet.

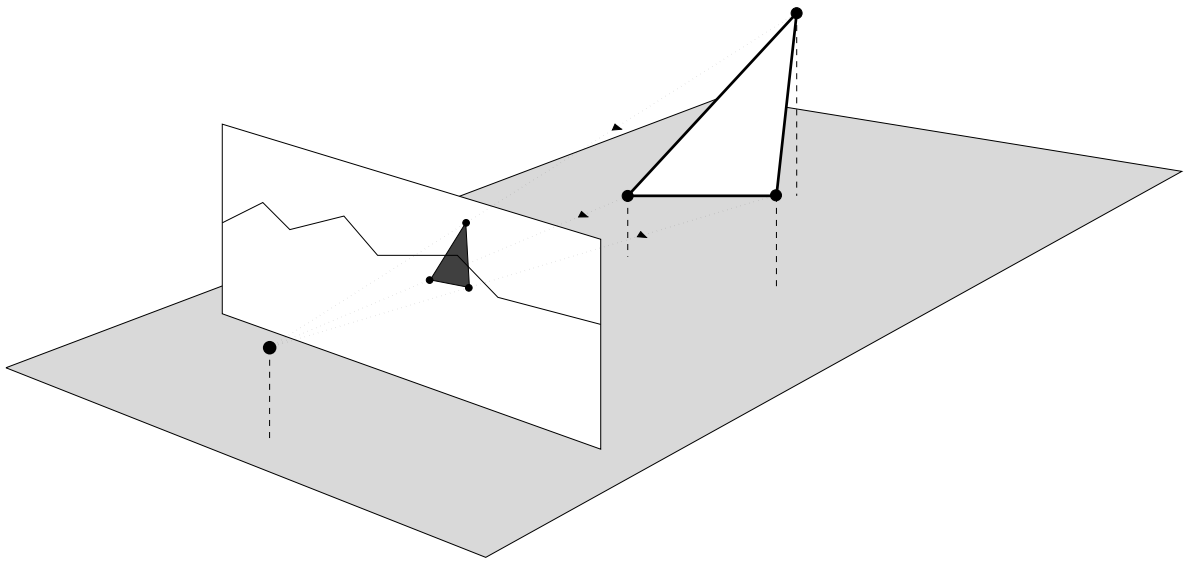

Figure 6.6: A walk through the operation of horizon. (2) Projecting the facet onto the projection plane ("wing").

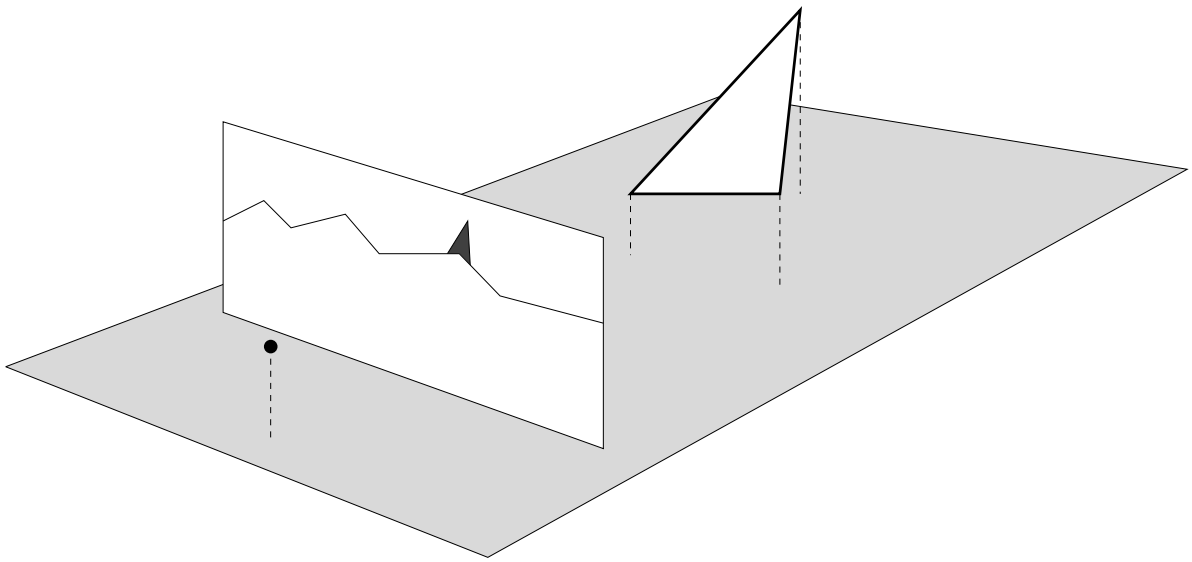

Figure 6.7: A walk through the operation of horizon. (3) The impact this facet has on the horizon.

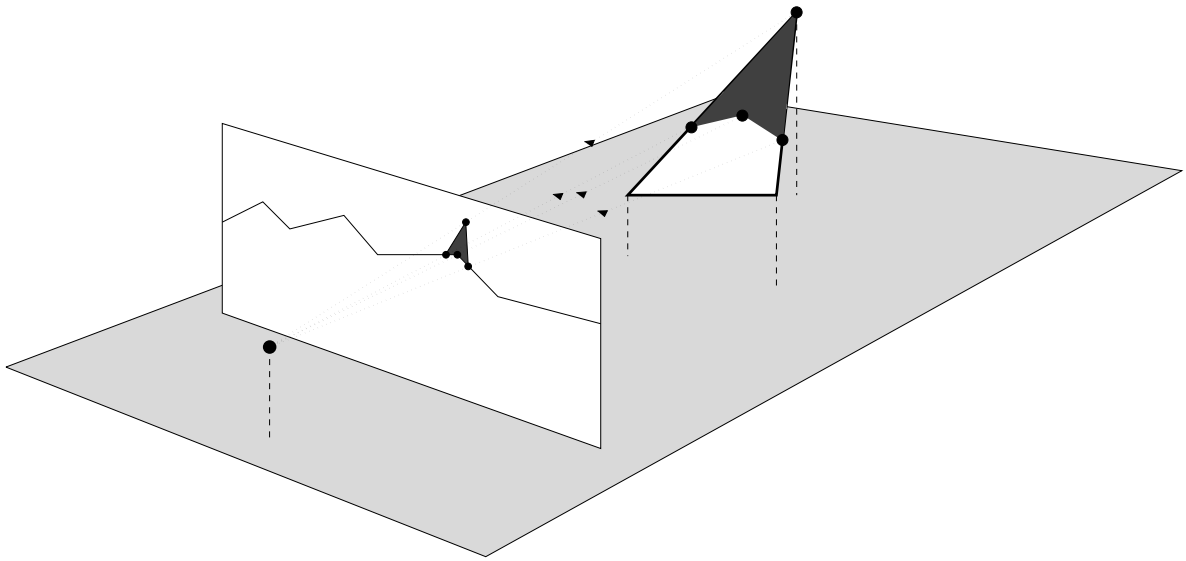

Figure 6.8: A walk through the operation of horizon. (4) project the visible portion back onto the facet.

6.4 Sorting the Facets

For the horizon data structure to produce the field of view correctly, the facets of the surface model must be presented to it in order of occlusion; a facet may not occlude a facet that has already been processed. This can be accomplished by a special sort pass over the facets of the model, producing a list of the facets in the required sequence. Note that occlusion depends on the viewpoint, so that even for the same surface, computing the field of view from different viewpoints requires separate sorting phases.

However, the very purpose of spatial indices is to make such special processing unnecessary. Both the RQT and QTN surface models can be made to generate the facets in the required sequence without an explicit sort phase by relying on their spatial indexing capabilities. Due to the different approach the two structures take to indexing, however, the way this is done is specific to the model. The way the RQT surface model sorts the facets is discussed in Section 6.4.2 and the counterpart procedure for the QTN model is described in Section 6.4.3. Issues common to both schemes are discussed in Section 6.4.1.

6.4.1 Common Issues

Observation: Three-dimensional occlusion may be excluded based on two-dimensional considerations. Let P be a viewpoint and f_1 and f_2 be two facets of a surface model in three-dimensional space. Furthermore, let P', f_1' and f_2' be their corresponding projections on the x-y plane. f_1 may occlude f_2 when viewed from P only if f_1' occludes f_2' when viewed from P', i.e., there is a point Q' on f_2' such that $\overline{P'Q'}$ intersects f_1'. This is because if f_1 occludes f_2 there is a line of sight from P to a point Q on facet f_2 that is frustrated by f_1. The projection of that line is $\overline{P'Q'}$.

Consequently, if the partition of the x-y plane induced by projecting the surface model onto it can be sorted, then that order is a sorting of the corresponding three-dimensional facets as well. This property allows the algorithms to make all their determinations based on the two-dimensional projection of the surface model, thus simplifying the sorting task.

It is important to note that it is not always possible to sort the facets of a model. Arbitrary triangulations may produce occlusion cycles, such as the one depicted in Figure 6.9. For the situation shown in Figure 6.9b to occur, the surface would have to contort in a way that our continuous, $2\frac{1}{2}$-D surfaces are incapable of, but the configuration in Figure 6.9a is possible in arbitrary triangulations. It is proved in [DeFl91], however, that Delaunay triangulations are immune to such cycles. Therefore, we restrict the QTN model to surface models derived from Delaunay triangulations.

6.4.2 Sorting the Facets in the RQT Model

The restricted quadtree surface model (RQT) is special in that all its facets fit neatly into the square blocks of the underlying quadtree. It is therefore possible to sort the blocks first and then sort the facets within each block later.

The blocks are sorted through a traversal of the internal nodes of the quadtree. The space covered by the surface is divided into four zones with respect to the viewpoint (Figure 6.10)

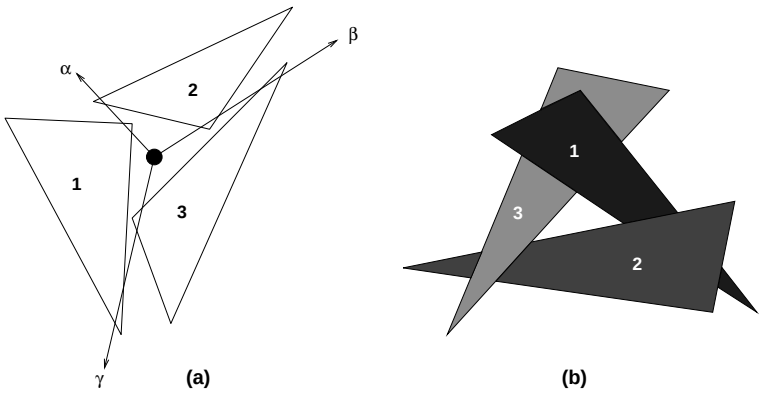

Figure 6.9: An example of a set of triangles defying spatial sorting. (a) An arrangement of three triangles on the plane that cannot be sorted with respect to the given viewpoint (solid circle). Along line-of-sight α, triangle 1 precedes triangle 2; along β, 2 precedes 3; and along γ, 3 precedes 1. (b) An image of three unsortable triangles.

formed by parallels to the x and y axes passing through the viewpoint. If the center of an internal node is inside, say, the north-west zone, then clearly all the facets in its descendant marked 1 in the figure are nearer the viewpoint P than any in the other descendants. That property holds for all the zones, with the exception of the descendants marked 2 and 3, which are at equal distance and could be interchanged.

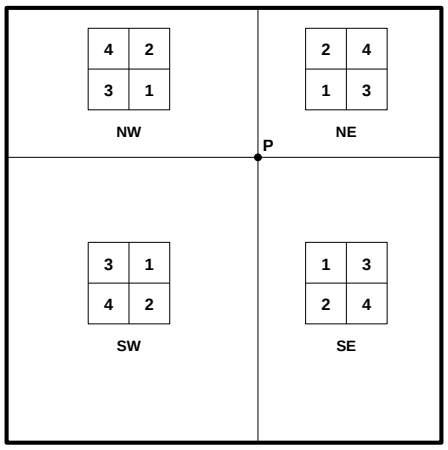

Figure 6.10: The zones induced by a viewpoint for RQT block sorting. The order in which descendants are processed in each zone is indicated by the numbers in the sample blocks.

Similarly, the facets within a terminal node are sorted according to the orientations they may have with respect to the viewpoint. The eight possible orientations and the ordering each induces among the possible facets are shown in Figure 6.11.

Figure 6.11 assumes blocks with eight facets each, but an RQT block may be configured with fewer facets (see Section 3.2.3, Figure 3.4). In that case, some (or all) of the facets are larger, each covering two adjacent facets of the size shown in the figure. When such configurations are encountered, a large facet takes the place the first (in sorting order) small

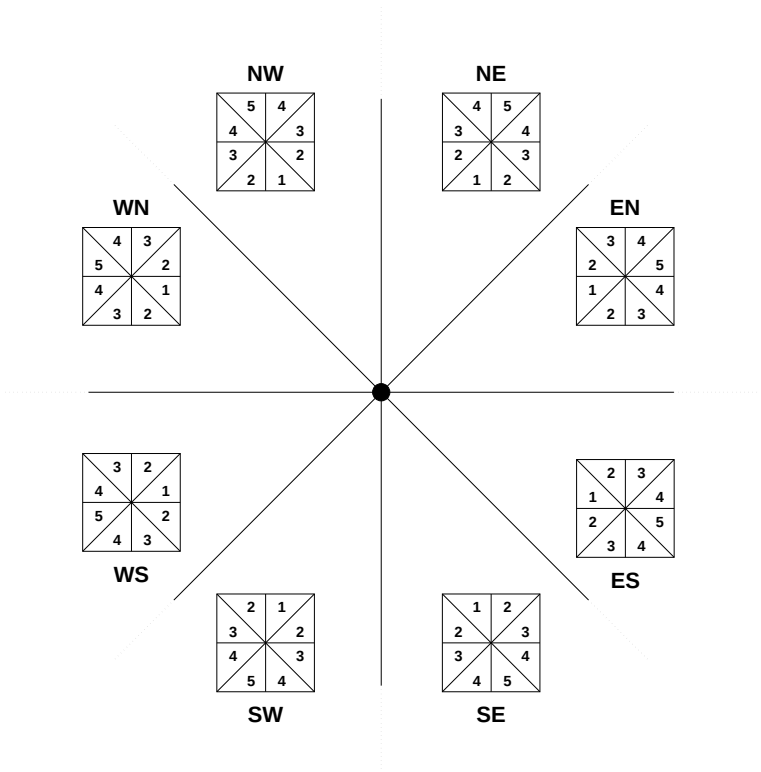

Figure 6.11: The eight orientations a block may have with respect to the viewpoint and the ordering each orientation induces among the facets in the block. Similarly numbered zones may be scanned in any order.

facet it covers would have taken.

Given the classifications of internal and terminal nodes described above, the formulation of the sorting algorithm is straightforward and is given in Figure 6.12.

The performance of algorithm rqt-sort depends on the implementation chosen for the representation of the quadtree. Each node, internal and terminal, is visited once, as well as each facet. The number of internal nodes is smaller than the number of terminal nodes in any tree of fixed degree. Moreover, since the number of facets per node is bounded from above and from below, the number of terminal nodes is smaller than the number of facets. If finding a descendant of a given node is an $O(1)$ operation, then the sorting can be done in time proportional to the number of facets in the model.

6.4.3 Sorting the Facets in the QTN Model

Unlike the RQT case, the facets in the QTN model are not in registration with the boundaries of the nodes of the underlying quadtree, so sorting them is not helpful in this case. Algorithm qtn-sort is an adaptation of one presented in [DeFl89a] for sorting Delaunay triangulations in general. It has been modified to make use of the spatial index the QTN model has to offer.

Algorithm qtn-sort maintains an active border which is always star-shaped about the

```
RQT-Sort(RQT R, viewpoint P)

1. B ← root of R.
2. if B is an internal node then begin
3.    Z ← the zone B is in with respect to P (Figure 6.10).
4.    for each descendant C of B, in the order defined by Z
      do
        Sort(C, P).
   end
5. else begin // B is a terminal node
6.    O ← the orientation B is in wrt P (Figure 6.11).
7.    for each facet F in B, in the order defined by O do
        Process F
   end
```

Figure 6.12: Algorithm rqt-sort to sort the facets of an RQT surface model according to their distances from a given viewpoint. The processing of facets done in line 7 is in the sorted order.

viewpoint. Its edges are also edges of the triangulation, i.e. the projection of the surface model onto the x-y plane. It thus separates the triangles that have been processed from those that have yet to be. Initially, the active border is set to be the boundary of the initial polygon, described in Section 6.2. The algorithm then picks an edge on the active border at random, and considers the unprocessed triangle the edge is incident upon. If certain conditions hold (see Figure 6.15), the triangle is processed and its edge (or edges) that were not part of the active border replace the edges (or edge) that were. A theorem from [DeFl91] guarantees that, provided the triangulation is Delaunay, there is always a triangle for which the conditions are true. This step is repeated until all triangles are processed. Figure 6.13 shows a sample Delaunay triangulation and several possible stages in the development of the active border.

At each step, algorithm qtn-sort picks an edge, say E, from the current active border at random. Unless the edge is at the boundary of the surface, it is incident on an unprocessed triangle T by virtue of the fact that the active border is at the boundary between the processed triangles and the unprocessed ones. There are three possible cases, depicted in Figure 6.14:

1. Another edge of T is included in the active border, as in Figure 6.14a. (Clearly, E and

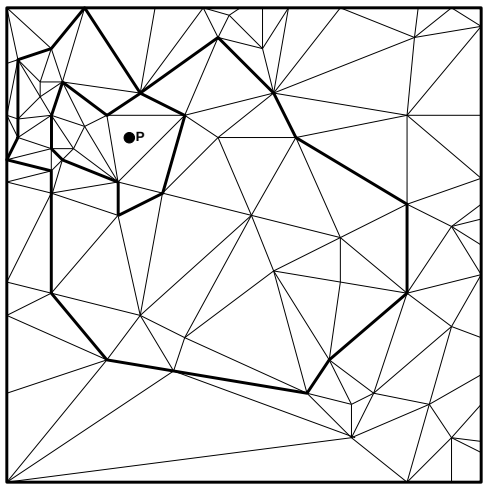

Figure 6.13: A sample Delaunay triangulation and several stages in the development of the active border during the execution of qtn-sort.

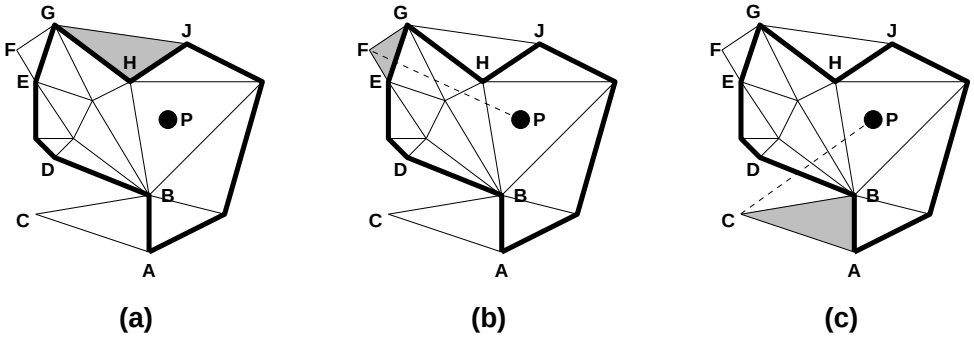

Figure 6.14: Algorithm qtn-sort incremental step: the three possible configurations. (a) qtn-sort selected $\overline{GH}$. $\triangle GHJ$ is acceptable since two if its edges, $\overline{GH}$ and $\overline{HJ}$ are both on the active border. (b) qtn-sort selected $\overline{EG}$. $\triangle EFG$ is acceptable because $\overline{PF}$ intersects the selected edge $\overline{EG}$. (c) qtn-sort selected $\overline{AB}$. $\triangle ABC$ is not acceptable since $\overline{PC}$ does not intersects the selected edge $\overline{AB}$, indicating that there is a triangle ($\triangle BCD$ in this case) that is closer to P.

E' must be adjacent on the active border, a fact which simplifies testing this case.) T is acceptable and may be processed next. Then E'', T's remaining edge, replaces both E and E' on the active border.

2. Both of the T's remaining edges E' and E'' are not part of the active border, hence T has a vertex V that is not on the border as well. If $\overline{PV}$ intersects E, as in Figure 6.14b, again the triangle is acceptable and may be processed next. E' and E'' then replace E on the active border.

3. The triangle is as described in item 2 except that $\overline{PV}$ does not intersects E, as in Figure 6.14c. In this case, there is an unprocessed triangle that is closer to the viewpoint than T and must be processed first. T is therefore rejected and this step is wasted.

Algorithm qtn-sort is given in pseudo-code in Figure 6.15.

In implementing algorithm qtn-sort, provision must be made for maintaining the active border since it does not fit into the QTN model. The spatial index comes in handy when the triangles incident on a particular border edge are sought. If the surface model contains N triangles, each triangle takes $O(\log N)$ time to search. The total search involves processing all triangles, which requires a minimum of $O(N \log N)$, if the configuration described in case 3 never occurs. In the worst case, a complete scan of the active border yields one acceptable triangle. If triangle distribution is uniform, the active border is expected to have $O(\sqrt{N})$ segments since it represents the perimeters of $O(N)$ triangles. The worst case behavior of this algorithm could then be $O(N^{\frac{3}{2}} \log N)$.

6.5 Experimental Results

The field-of-view algorithm was implemented and tried out on several actual datasets. In addition to the datasets "Reno west 0,0", "data" and "Salisbury east 2,0" described in Section 3.4.3, tests were performed on a 129×129 section of the "data" map. The tests consisted of measuring the time required to determine the viewable area from various viewpoints. Maps constructed from the various datasets at several tolerance values were employed. The results are summarized in Table 6.1. Each number in the table represents an average of several measurements taken from different viewpoints.

Table 6.1: Experimental results of field of view determination. The results represent averages taken over several viewpoints. Execution times are in seconds.

	RQT			QTN		
tolerance (meters)	facet count	execution time (sec)	time/facet (milisec)	facet count	execution time (sec)	time/facet (milisec)
Salisbury east 2,0 (513×513)						
1	3178	372	117	336	55.2	164
3	666	64.5	97	78	13.4	172
Data (129×129)						
3	17728	1181	67	8540	1017	119
5	12268	537	44	4462	489	110
10	4806	294	61	1532	176	115
30	1090	27	25	250	45	180
100	4	.5	125	4	2.3	575
Reno west 0,0 (513×513)						
100	7769	570	73	2067	742	359
300	926	65.5	71	325	177	544

To compare the efficiency of the RQT and QTN surface models, the timing data was processed in two ways. One interpretation measures the time required to process a single

facet of the model. Since the field-of-view algorithm examines every triangle of the model, larger models with more facets are expected to take longer to process. To eliminate the effect of size, the ratio of total execution time to the number of facets in the model is computed. It is argued that the time to process a facet has a generic component, the time spent in the common horizon structure (Section 6.3), and a model specific component, the time taken to sort the triangles (Section 6.4). It is assumed that over a large number of trials the difference between this time-per-facet figure for the two models is attributable to the model specific component, and indicates the model's relative effectiveness in carrying out the required sorting task. Execution time results are plotted against model size in Figure 6.16

The other interpretation of the results compares the performance of maps of the two types constructed from identical data sets using the same tolerance. Also, the same set of viewpoints is used in the testing the two models. The results of execution time as a function of model tolerance are plotted for the smaller "data" map in Figure 6.17.

Execution times per triangle are consistently lower for the RQT. This should mean that the RQT is able to sort triangles faster than the QTN can. This result is to be expected, since the RQT performs the sort by manipulating the blocks which contain them, avoiding the overhead involved in accessing the individual triangles.

On the other hand, the triangles in a TIN are not constrained by the underlying quadtree as the RQT triangles are, and thus may have a higher information content. Consequently, a QTN can model a given surface to a given tolerance with fewer triangles than an RQT would require.

These two capabilities are in competition, and can make either model outperform the other, depending on the nature of the surface. For surfaces whose variation is moderate, an RQT model may be burdened with many unnecessary faces, mandated by the restriction on the sizes of neighboring blocks. This compounds the RQT's propensity to produce larger models. As a result, RQT models display poorer performance in this case, as seen from the results for "Salisbury east 2,0".

The RQT model seems to be doing better for surfaces with great variability, on the other hand. When the ruggedness of the surface requires many triangles to faithfully model it, the disadvantages of the RQT model are not as pronounced, and its faster processing can lead it to outperform the QTN model, as seen in the results for "Reno west 0,0".

For the "data" map, which exhibits moderate variation in elevation, the results for the two models are quite close, lending support to the argument made above.

```
QTN-Sort(QTN Q, viewpoint P)

1.  L ← list of the edges of the initial polygon
    (Section 6.2).
2.  while there is an edge in L that is not on the map's
    boundary do begin
3.    E ← an edge in L that is not on the map's boundary.
4.    T ← the triangle incident on E farthest from P.
5.    if two of the edges of T are in L then begin
6.      Remove from L the edges of T it contains.
7.      Add to L the edge of T it did not contain.
8.      Process T.
      end
      else begin
9.      Q ← the vertex of T not incident on any edge in L.
10.     if the line from P to Q intersects E then begin
11.       Remove E from L.
12.       Add to L the edges of T other than E.
13.       Process T.
        end
      end
    end
```

Figure 6.15: Algorithm qtn-sort to sort the facets of a QTN surface model according to their distance from a given viewpoint. The triangles processing done in line 8 and 13 is done in sorted order.

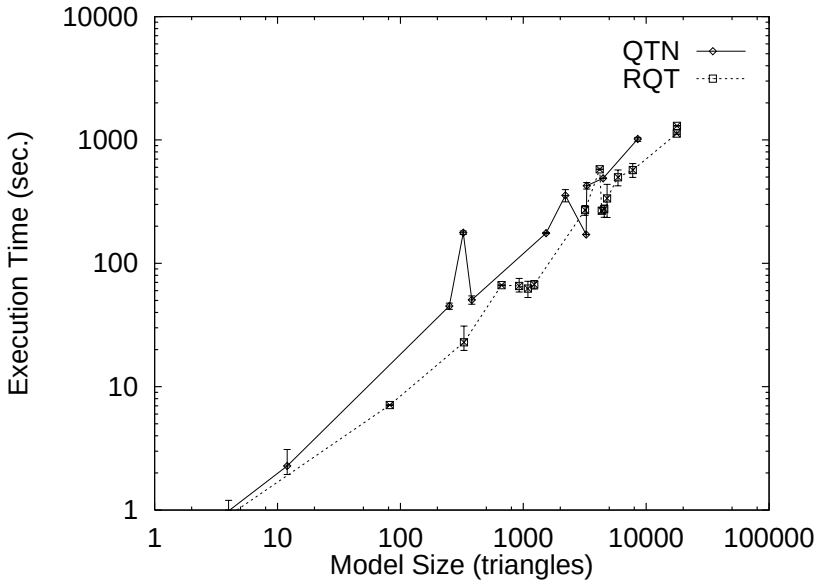

Figure 6.16: field-of-view algorithm execution times vs. model size.

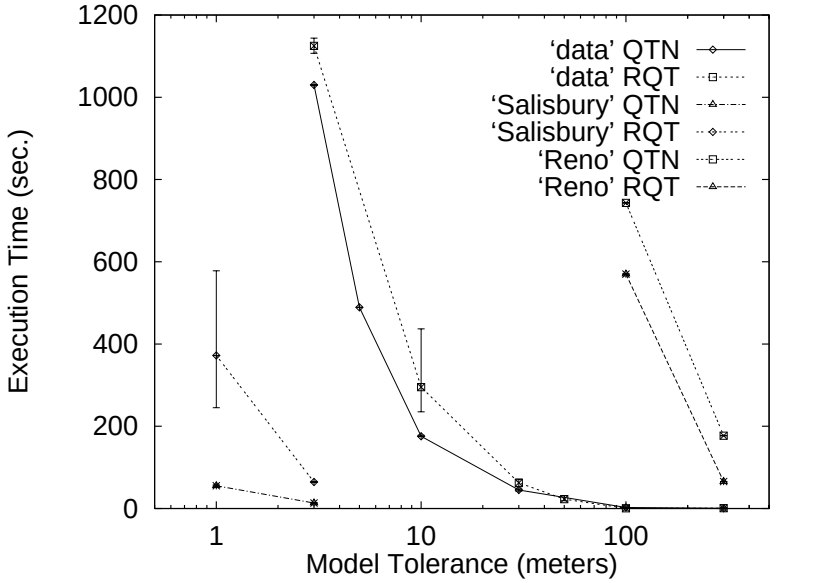

Figure 6.17: field-of-view algorithm execution times vs. model tolerance.

Chapter 7

Conclusions and Future Research

We have studied two ways in which polyhedral models of $2\frac{1}{2}$-D surfaces can be embedded into quadtree spatial indices. The restricted quadtree was found to be useful in supporting regular elevation grids such as are found in DEMs. Two algorithms for constructing restricted quadtrees from DEM data were presented, and the conditions under which each of them may be preferable were shown. For irregular data, such as found in TINs, using a PMR quadtree was found to be appropriate.

A field of view algorithm that can be used with both surface models was presented. It was used to exercise the two implementations and determine when it is better to use the one or the other. It was found that the RQT model can perform individual tasks faster than the QTN. The TIN, however, can model a given surface to a given tolerance with fewer triangles than the RQT . In balance, it was found that the QTN's relative slowness can be compensated for by its smaller size for more moderate maps, but that the RQT outperforms it when modeling more rugged terrain.

Further study is required to find what applications could use RQTs to their advantage. It seems that applications which operate on two or more surfaces defined over the same area could be helped by the consistency provided by a common registration of the models. For example, operations involving comparison or intersection of two surfaces simultaneously require the parts of the surfaces that project to the same area in the xy plane. Those parts would correspond in particularly simple ways if they are both nodes in registered quadtrees, as would be the case if RQTs were used.

In the course of this research, a testbed for comparing the RQT and QTN was implemented, utilizing to a great extent shared object-oriented code. Other surface applications that can be implemented in these terms, such as perspective display, could be used to further test the appropriateness of these two surface models for various tasks.

Bibliography

[Apos69] T.M. Apostol. *Calculus*, volume II. John Wiley & Sons, New York, second edition, 1969.

[Aref92c] W.G. Aref and H. Samet. An efficient window retrieval algorithm for spatial query processing. Computer Science Department TR-2866, University of Maryland, College Park, MD, March 1992.

[Arya94] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, Arlington, VA, January 1994.

[Bank83] R.E. Bank, A.H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. In R. Stepleman et al., editor, *Scientific Computing*, pages 3–17. IMACS/North Holland Publishing Company, 1983.

[Barr87] R. Barrera and A. Hinojosa. Compression methods for terrain relief. Engineering Projects Section, Department of Electrical Engineering CINEVESTAV—IPN, Polytechnic University of Mexico, Mexico City, 1987.

[Bern90] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. In *Proceedings of the Thirty-first Annual IEEE Symposium on the Foundations of Computer Science*, pages 231–241, St. Louis, MO, October 1990.

[DeFl84] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. A hierarchical structure for surface approximation. *Computers & Graphics*, 8(2):183–193, 1984.

[DeFl85] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. Efficient selection, storage and retrieval of irregularly distributed elevation data. *Computers and Geosciences*, 11(6):667–673, 1985.

[DeFl89] L. De Floriani. A pyramidal data structure for triangle-based surface description. *IEEE Computer Graphics and Applications*, 9(2):67–78, March 1989.

[DeFl89a] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. Polyhedral terrain description using visibility criteria. Unpublished, October 1989.

[DeFl91] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. On sorting triangles in a Delaunay tessellation. *Algorithmica*, 6(4):522–532, 1991.

[DeFl92b] L. De Floriani and E. Puppo. A hierarchical triangle-based model for terrain description. In A. Frank, I. Campari, and U. Formentini, editors, *International Conference, GIS—From Space to Territory: Theory and Methods of Spatio-Temporal Reasoning*, Pisa, Italy, September 1992. Springer Verlag, Berlin.

[Dutt84] G. Dutton. Geodesic modelling of planetary relief. *Cartographica*, 21(2–3):188–207, Summer – Autumn 1984.

[Feke90] G. Fekete. Rendering and managing spherical data with sphere quadtrees. In *Proceedings of Visualization 90*, pages 176–186, San Francisco, CA, October 1990.

[Fole90] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

[Four82] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.

[Fowl79] R.J. Fowler and J.J. Little. Automatic extraction of irregular digital terrain models. *Computer Graphics*, 13(2):199–207, August 1979. Also *Proceedings of the SIGGRAPH '79 Conference*, Chicago, IL, August 1979.

[Gome79] D. Gomez and A. Guzman. Digital model for three-dimensional surface representation. *Geo-Processing*, 1:53–70, 1979.

[Gour71] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, 20(6):623–629, June 1971.

[Hara69] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.

[Herb84] F. Herbert. Fractal landscape modelling using octrees. *IEEE Computer Graphics and Applications*, 4(11):4–5, November 1984.

[Hoel91] E.G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In O. Günther and H.J. Schek, editors, *Advances in Spatial Databases—2nd Symposium, SSD'91*, pages 237–256. Springer-Verlag, Berlin, 1991. (Lecture Notes in Computer Science 525.)

[Kirk83] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, February 1983.

[Mand68] B.B. Mandelbrot and J.W. Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM Review*, 10(4):422–437, October 1968.

[Moor92] D.W. Moore. *Simplicial Mesh Generation with Applications*. PhD thesis, Cornell University, Department of Computer Science, Ithaca, NY, December 1992. (Cornell University Technical Report 92-1322.)

[Nels86a] R.C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, August 1986. Also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, TX, August 1986.

[Newe72] M.E. Newell, R.G. Newell, and T.L. Sancha. A solution to the hidden surface problem. In *Proceedings of the ACM National Conference*, pages 443–450, 1972.

[Peuc75] T. Peucker and N. Chrisman. Cartographic data structures. *American Cartographer*, 2(2):55–69, April 1975.

[Poli92] M.F. Polis and Jr. D.M. McKeown. Iterative TIN generation from digital elevation models. In *Proceedings of Computer Vision and Pattern Recognition '92*, pages 787–790, Champaign, IL, March 1992.

[Prep85] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer–Verlag, New York, 1985.

[Same90a] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[Same90b] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[Same94a] H. Samet and A. Soffer. Automatic interpretation of floor plans using spatial indexing. In S. Impedovo, editor, *Progress in Image Analysis and Processing III*, pages 233–240. World Scientific, Singapore, 1994.

[Scar92] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographic coherence. *CVGIP: Graphical Models and Image Processing*, 54(2):147–161, March 1992.

[Schm86] F.J.M. Schmitt, B.A. Barsky, and W.H. Du. An adaptive subdivision method for surface-fitting from sampled data. *Computer Graphics*, 20(4):179–188, August 1986. Also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, TX, August 1986.

[Shaf87a] C.A. Shaffer and H. Samet. Optimal quadtree construction algorithms. *Computer Vision, Graphics, and Image Processing*, 37(3):402–419, March 1987.

[Shaf90b] C.A. Shaffer, H. Samet, and R.C. Nelson. QUILT: a geographic information system based on quadtrees. *International Journal of Geographical Information Systems*, 4(2):103–131, April-June 1990. (Also University of Maryland Computer Science TR-1885.1).

[Sibs78] R. Sibson. Local equiangular triangulations. *Computer Journal*, 21(3):243–245, August 1978.

[Spro91] R.F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[USGS90] U.S. Geological Survey, Department of the Interior, Reston, VA. *Digital Elevation Models*, data users guide, fifth edition, 1990.

[VonH89] B. Von Herzen. *Applications of Surface Networks to Sampling Problems in Computer Graphics*. PhD thesis, California Institute of Technology, Pasadena, CA, July 1989.