A SPACE SUBDIVISION FRAMEWORK FOR VIRTUAL REALITY AND HAPTIC APPLICATIONS

by

Engin Deniz Diktaş

BS. in Mechanical Engineering, Boğaziçi University, 2002

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Systems and Control Engineering
Boğaziçi University
2006

FOREWORD

To my grandparents, Asım & Mualla Diktaş

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Doctor Ali Vahit Şahiner for his invaluable guidance and help during the preparation of this thesis. I would like to mention his patience, giving me inspiration and hope when I was stuck at dead-ends. I am also very grateful to Prof. Ahmet Ademoğlu, who has always helped me and pointed me the right direction in my research and also to Prof. Lale Akarun for having broadened my horizon in her Computer Graphics course.

I am in debt to my grandmother Mualla Diktaş, who supported me throughout my whole study. Without her support I would not be able to complete the present thesis. I also want to thank my dear friends Mustafa Doğan, Murat Günal, Erdem Erdemir, Ali Polat and Özkan Taşçıoğlu for their support in my desperate times. I thank them all.

ABSTRACT

A SPACE SUBDIVISION FRAMEWORK FOR VIRTUAL REALITY AND HAPTIC APPLICATIONS

This thesis presents a framework based on spatial subdivision of triangular mesh objects with octrees for virtual reality and haptic applications.

For efficient generation of octree representations a method based on Minkowski sums is developed. The calculation of Minkowski sums is achieved geometrically by direct manipulation of planes in 3D-space.

For the evaluation of proximity queries a new data structure called the *proximity octree* is introduced. A technique for tracking of geometric features of convex objects that are in close proximity to a specific point of interest is developed and tested for a number of applications including haptic interaction.

ÖZET

SANAL GERÇEKLİK VE HAPTİK UYGULAMALAR İÇİN SEKİZLİ-AĞAÇ TABANLI BÖLÜMLEME ALTYAPISI

Bu tezde sanal gerçeklik ve haptik uygulamaları için üçgenlerden oluşan nesnelerin sekizli-ağaçlarla bölümlenmesine dayanan bir yapı sunulmaktadır.

Sekizli-ağaçların verimli bir biçimde oluşturulabilmesi için Minkowski toplamına dayalı bir yöntem geliştirilmiştir. Minkowski toplamlarının hesaplanması düzlemlerin yerlerinin 3 boyutlu uzayda geometrik olarak doğrudan ötelenmesiyle sağlanmaktadır.

Yakınlık sorgularının değerlendirilmesi için *yakınlık sekizli-ağacı* adı verilen yeni bir veri yapısı sunulmaktadır. Bu veri yapısı aracılığıyla dışbükey bir nesnenin geometrik bileşenleri arasından, belirli bir noktaya en yakında olanlarının izlenmesine yönelik bir teknik geliştirilmiş ve bu yöntem haptik bir uygulama için denenmiştir.

TABLE OF CONTENTS

FOREWORD . . . iii
ACKNOWLEDGEMENTS . . . iv
ABSTRACT . . . v
ÖZET . . . vi
LIST OF FIGURES . . . ix
LIST OF TABLES . . . xii
LIST OF SYMBOLS/ABBREVIATIONS . . . xiii
1. INTRODUCTION . . . 1
  1.1. Overview . . . 1
  1.2. Thesis Outline . . . 2
2. HIERARCHIAL REPRESENTATIONS . . . 4
  2.1. Octrees . . . 5
    2.1.1. Memory Requirements in Adaptive Subdivision . . . 7
    2.1.2. Advantages & Applications . . . 11
3. OCTREE GENERATION USING MINKOWSKI SUMS . . . 12
  3.1. Minkowski Sums . . . 12
  3.2. A Geometric Method for Computing Minkowski Sums . . . 14
    3.2.1. Extending the Geometric Method to 3D . . . 17
  3.3. Computational Complexity of Minkowski Sum Calculation . . . 24
    3.3.1. Geometric Minkowski Sum Calculation . . . 24
    3.3.2. Incremental Convex Hull Construction . . . 25
  3.4. Octree Generation Using Minkowski Sums . . . 27
  3.5. Index Tracking . . . 28
  3.6. Computational Complexity of Intersection Detection . . . 29
    3.6.1. Intersection detection using Minkowksi sums . . . 30
    3.6.2. Intersection detection using classical tests . . . 31
    3.6.3. Intersection Detection using Separating Axis Overlap Test . . . 33
    3.6.4. Conclusion for intersection detection . . . 35
  3.7. Performance Statistics for Octree Generation . . . 36

4. PROXIMITY OCTREE 38
4.1. Need for proximity queries 38
4.2. Related Work 38
4.3. Proximity Octrees 39
4.3.1. Generation 40
4.3.2. Voronoi Regions 42
4.3.3. Intersection of Voronoi Regions and Cubes 45
5. APPLICATIONS AND PERFORMANCE EVALUATIONS 49
5.1. Haptic Interaction 49
5.1.1. Use of Proximity Octree for Haptic Interaction 51
5.1.2. Culling Performance of the Proximity Octree 52
5.2. Object-Object Intersections 54
5.3. Ray Casting 57
5.4. View Volume Culling 59
6. CONCLUSIONS 61
REFERENCES 62

LIST OF FIGURES

Figure 2.1. BSP tree representation of a scene 5

Figure 2.2. Balanced optimal BSP-trees 5

Figure 2.3. Uniform octree subdivision 7

Figure 2.4. Adaptive octree subdivision 7

Figure 2.5. Octree data structure 8

Figure 2.6. Adaptive octree subdivision for sphere 9

Figure 2.7. Adaptive octree subdivisions of various benchmark models 11

Figure 3.1. Triangle and Rectangle and their normals 15

Figure 3.2. Convex hull of all vertices from the set V 15

Figure 3.3. The stippled line is shifted along n_1 until v_1 and v_2 are hit 17

Figure 3.4. Edges of the Minkowski sum parallel to the edges of the triangle . 17

Figure 3.5. The stippled line is shifted along n_4 until v_3 and v_4 are hit 18

Figure 3.6. Edges of the Minkowski sum parallel to the edges of the rectangle 18

Figure 3.7. Multiple Copies B placed at all vertices of A 20

Figure 3.8. Facets parallel to the facets of A 20

Figure 3.9. Facets parallel to the facets of B 21

Figure 3.10. Facets formed by sweeping the edges of B along the edges of A . . 21

Figure 3.11. The Final Minkowski Sum 22

Figure 3.12. Copies of cubes of different sizes placed at triangle-vertices 28

Figure 3.13. Minkowski sums showing a *topological invariance* 28

Figure 3.14. Index tracking 29

Figure 3.15. Degenerate cases in triangle and cube intersection 32

Figure 3.16. AKIRA Spaceship and its octree representations 37

Figure 4.1. Closest features assigned to the quadtree-nodes 41

Figure 4.2. The proximity octree and a Voronoi region for a spherical object . 41

Figure 4.3. Voronoi region of a triangle of the pyramid 43

Figure 4.4. Voronoi region of an edge of the pyramid 44

Figure 4.5. Voronoi region of an vertex of the pyramid 45

Figure 4.6. Voronoi regions of some convex 3D objects 46

Figure 4.7. Unbounded Minkowski sum in 2D 48

Figure 4.8. Copies of the cube placed at all vertices of each Voronoi region . . 48

Figure 4.9. Intermediate Minkowski sums of Voronoi regions 48

Figure 5.1. Phantom Omni haptic device 50

Figure 5.2. Snapshot of the haptic interaction application 52

Figure 5.3. Distribution of features for various depths 53

Figure 5.4. Intersection of 2 sphere-like objects 57

Figure 5.5. Ray intersecting voxels with full intensity contribution 58

Figure 5.6. Ray intersecting voxels with various intensity contributions 59

Figure 5.7. View volume culling example (isometric view) 60

Figure 5.8. View volume culling example (axis views) 60

LIST OF TABLES

Table 2.1. Statistics for octree of the sphere (threshold = 1) 9

Table 2.2. Statistics for cctree of the sphere (depth = 5) 10

Table 2.3. Octree statistics for various benchmark models (depth = 5) 11

Table 3.1. Terms in separating-axis based intersection test 34

Table 3.2. Summary of computational complexities 36

Table 3.3. Elapsed time for generating the octree 36

LIST OF SYMBOLS/ABBREVIATIONS

A,B	Polyhedra
$d(x,\mathcal{F}_i)$	Euclidean distance of x to $\mathcal{F}_i$
e_i	i^{th} edge of a polyhedron
E	Set of edges
F_i	i^{th} facet of a polyhedron
$\mathcal{F}_i$	i^{th} feature of a polyhedron
H	Halfspace
M	Minkowski sum
n_i	i^{th} normal
N	Set of normals
t_i	i^{th} triangle of a polyhedron
T	Set of triangles
T_i	Homogenous transformation matrix
$T(v_i)$	Set of triangles incident to v_i
v_i	i^{th} vertex of a polyhedron
V	Set of vertices
$\mathcal{V}_{\mathcal{F}_i}$	Voronoi region of $\mathcal{F}_i$
AABB	Axis-Aligned Bounding Box
BSP	Binary Space Partitioning
DOF	Degrees Of Freedom
HDAPI	HapticDevice API
HLAPI	Haptic Library API
NURBS	Non-Uniform Rational B-Splines
OBB	Oriented Bounding Box

1. INTRODUCTION

1.1. Overview

Virtual reality and haptics applications play an important role in virtual prototyping and manufacturing, robotics, simulation and training systems. Evaluations performed on virtual models of real-world objects are much faster and less costly than tests on the actual prototypes.

In these applications one of the key issues is the interaction between the virtual objects themselves and the interaction between the virtual environment and a virtual component controlled by the user via a device such as a data glove or a haptic interface. In order to be able to detect these interactions and their levels, effective means need to be developed. Since interactions due to collision and proximity among objects are local, a large portion of the primitives can be culled when handling collision detection and proximity queries.

A common method to achieve this is to utilize hierarchial data structures that constrain these queries to as small as possible regions of the scene or the objects. Hierarchial representations can be also used for a variety of problems in graphics rendering such as view-frustum and occlusion culling, ray-casting, visibility problems and shadow calculation.

The octree data structure is a hierarchial data structure that is used for handling a broad range of queries and visualization tasks. It therefore can provide a basis for the unification of data structures used in a visualization environment. Addressing these problems within a framework that is based on the same type of hierarchial data structure is advantageous in the sense that it minimizes the overhead associated with the generation and handling of different data structures.

An octree data structure can be used in a variety of queries due to the high spatial

coherence among its nodes. This spatial coherence is the direct result of the recursive subdivision of the nodes into equal-sized octants that have the same orientation and which are enumerated in the same fashion. As a result the octree provides an efficient space-enumeration scheme that allows fast access to individual features in space.

Hierarchial representation of an object requires the calculation of intersection of the octree-cubes with the geometric features of the object. The fact that all nodes of an octree at a certain depth have the same size and orientation can be exploited to accelerate these intersection calculations by using Minkowski sums. Also the nodes of an octree at different depths have the same orientation resulting in a topological invariance in the calculated Minkowski sum. We can exploit this property when calculating the intersections of the geometric primitives with cubes at different depths.

Since the octree provides a simple space enumeration scheme for any region of space, this can be used to keep information about the geometric features that are in close proximity with a partitioned region. This allows proximity queries between a convex object and a point of interest to be performed effectively. In the thesis, an octree designed specifically to hold this type of proximity information is developed and named as *proximity octree*.

1.2. Thesis Outline

This thesis is divided into 6 chapters. In chapter 1 we present the motivation for the thesis and the outline.

In chapter 2, the two types of most commonly used hierarchial representations are introduced, namely *bounding volumes* hierarchies and *space partitioning* hierarchies. After stating the difference between them, we explain why we have concentrated on space partitioning methods, specifically on octrees.

In chapter 3 a new and efficient method for generating octree representations of triangular-mesh objects based on the use of Minkowski sums is presented. Here we

provide a review of the previous work on Minkowski sum calculation in the literature and introduce our method for calculating Minkowski sums of simple primitives and explain the effectiveness of this method. This chapter also presents computational complexity analyses for Minkowski sum calculation and intersection tests.

In chapter 4, we introduce an extension to the octree data struture to be used in proximity queries. We explain how octrees can be extended to be used in proximity queries and present efficient an efficient technique for generating the proximity octree. This involves assigning closest features to nodes of the octree by checking intersection of the octree-cubes with the Voronoi regions of the respective features.

In chapter 5, we evaluate our methods in the context of different applications. We first provide the details of a haptic rendering application, where the proximity octree is used to locate the feature of a convex object closest to the haptic cursor fast enough to achieve stable haptic interaction rates. Then we discuss applications related to object-object intersections for fast interference detection, ray-casting and view-volume culling for speeding up visualization.

In the final chapter, the conclusions and plans for future work are provided.

2. HIERARCHIAL REPRESENTATIONS

As stated in the introduction, there is a strong need for hierarchial representations of objects to perform specific types of geometrical queries efficiently. There are basically 2 types of hierarchial representations: *bounding volume hierarchies* and *space partitioning hierarchies*. Space partitioning hierarchies are based on the idea of subdividing a region of space occupied by an object, whereas in bounding volume hierarchies subdivision is carried along the boundary of the object.

Among the bounding volume hierarchies the most commonly used structures are bounding spheres [1, 2], axis-aligned bounding boxes [3, 4] and oriented-bounding boxes [5, 6]. Since the bounding volumes have the tendency to bound objects more tightly compared to the structures used in space partitioning, they are more suitable to be used in intersection detection queries. Among them hierarchies of oriented bounding boxes (OBB-trees) seem to show superior performance compared to other bounding volume hierarchies [6].

In contrast to bounding volume hierarchies, the space partitioning techniques are able to subdivide any region of space occupied by the object. In addition to being able to subdivide the boundary of an object, space subdivision techniques can also partition an arbitrarily large space around the object which is a desirable property when distance computations and proximity queries are considered. Space partitioning techniques are also able to capture the *spatial coherence* among the objects in a scene so that they provide a better means for efficient rendering of scenes.

There are different types of data structures and methods for space partitioning, like BSP-trees [7], Kd-trees and octrees [8]. BSP-trees and Kd-trees subdivide a region using separating planes such that features of interest are grouped as being in front or back of that separating plane. Once the features are grouped with respect to that plane, the grouped features themselves are subdivided recursively by constructing a new plane for each set of features. Kd-trees follow the same approach with the only

difference being that separating planes used for partitioning come from a pre-defined set of planes taking advantage of the special structure of the coherence among the objects. Figure 2.1 shows an example of the BSP-tree representation of a simple scene.

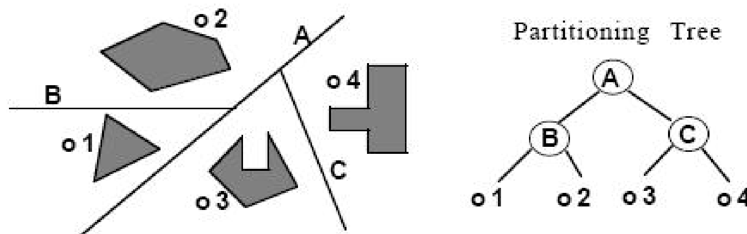

Figure 2.1. BSP tree representation of a scene [9]

BSP-trees provide an effective means to increase the look-up performance from $O(n)$ to $O(\log n)$ by exploiting the ordering. They are also cost-effective in terms of memory usage and tree-generation over-heads. The primary difficulty associated with BSP-trees is the selection of separating planes. A poor strategy employed for plane selection can easily lead to an unbalanced tree, thus resulting in less efficient queries as stated in [9]. The situation becomes more complicated for scenes where the objects have a non-uniform distribution as shown in Figure 2.2. In such a case, a balanced tree is not necessarily an optimal tree considering spatial distribution of geometric primitives. Also using BSP-trees is more challenging compared to octrees when proximity queries are considered.

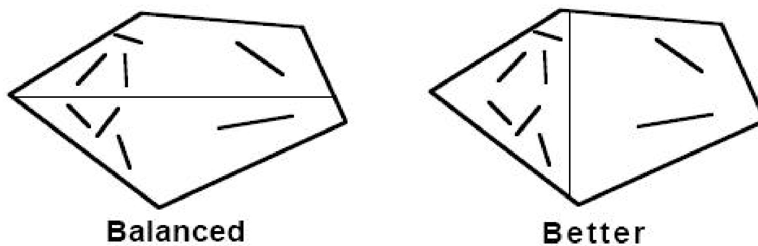

Figure 2.2. Balanced optimal BSP-trees [9]

2.1. Octrees

One commonly used hierarchical space partitioning technique involves the utilization of Octrees. Octree construction is based on recursive subdivision of nodes

into eight child nodes, starting with a root node whose geometric representation is a bounding cube containing all of the geometric primitives of the object under consideration, such that at the end of the subdivision process each leaf node of the octree contains a small number of primitives. As its name suggests, the prefix *oct* in the name octree comes from the word octant, meaning that the parent node is subdivided into equal-sized octants.

There are two types of subdivisions that can be performed: uniform and adaptive subdivision. In uniform subdivision all nodes of the octree are subdivided regardless of the number of primitives associated with that particular node. This type of subdivision can be ideal for cases where the data to be associated with the nodes of the octree is distributed uniformly, e.g. when dealing with volumetric data.

In cases where the data is distributed in a non-uniform fashion, e.g. if the octree representation of only the boundary of an object is required, it is more efficient to utilize *adaptive* subdivision. Adaptive subdivision is carried out on a node if the number of features assigned to that node exceeds a threshold value, resulting in a more compact octree compared to an octree that would be obtained by uniform subdivision.

Figure 2.3 illustrates the uniform (top) and adaptive subdivision (bottom) process for generating a quadtree for a 2d-object, namely a circle. A quadtree is the 2d-analogue of the octree and it is generated by subdivision into quadrants. Note that in Figure 2.3 those nodes containing the boundary of the object are subdivided until a desired maximum depth is reached. The octree data structure we use in our framework is shown in Figure 2.5. Each node has 5 fields:

- *Type* specifies in which region the cube represented by the node is placed with respect to the objects boundary. We call a node an *outer* node is the respective cube resides outside the boundary surface of the object. Therefore each node of the octree is either an *outer* node, an *inner* node or a *boundary* node.
- The *center* is the geometric center of the respective octree-cube.
- The *triangles* field keeps a pointer to a list of geometric primitives (in our case

triangles because we are considering only triangular meshes in our framework).

- The *children* field keeps a pointer to the child-nodes if the node is subdivided.

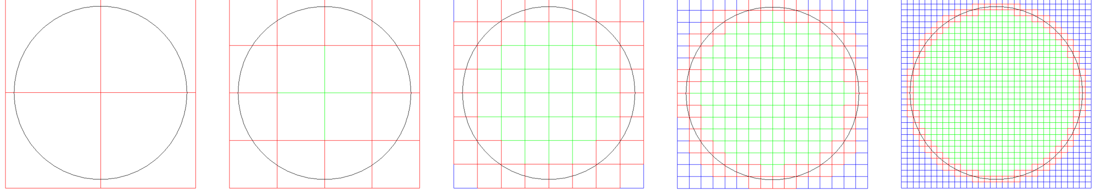

Figure 2.3. Uniform octree subdivision

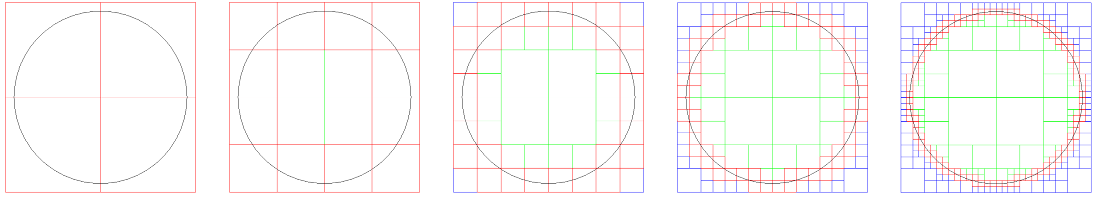

Figure 2.4. Adaptive octree subdivision

Note that the outer and the inner nodes do not keep any triangle information while the boundary nodes keep pointers to the triangles they contain or intersect. As shown in the figure, the child-node-array and triangle pointers of the inner and outer nodes are not defined.

2.1.1. Memory Requirements in Adaptive Subdivision

Figure 2.6 shows the adaptive octree-subdivision for a sphere consisting of 19800 triangles. Note that as the depth of the octree is increased the leaf nodes of the octree get smaller and start to follow the boundary of the object more closely.

Table 2.1 presents some statistics for octree generation for the sphere shown in Figure 2.6. The column titles used in these tables are as following:

- *Depth*: subdivision depth of the octree.
- *Total*: total number of nodes in the octree.
- *Leaves*: total number of leaf-nodes in the octree.
- *Occ*: number of occupied leaves as a percentage of the total number of leaves.
- *Min-Tri*: minimum number of triangles assigned to the nodes.

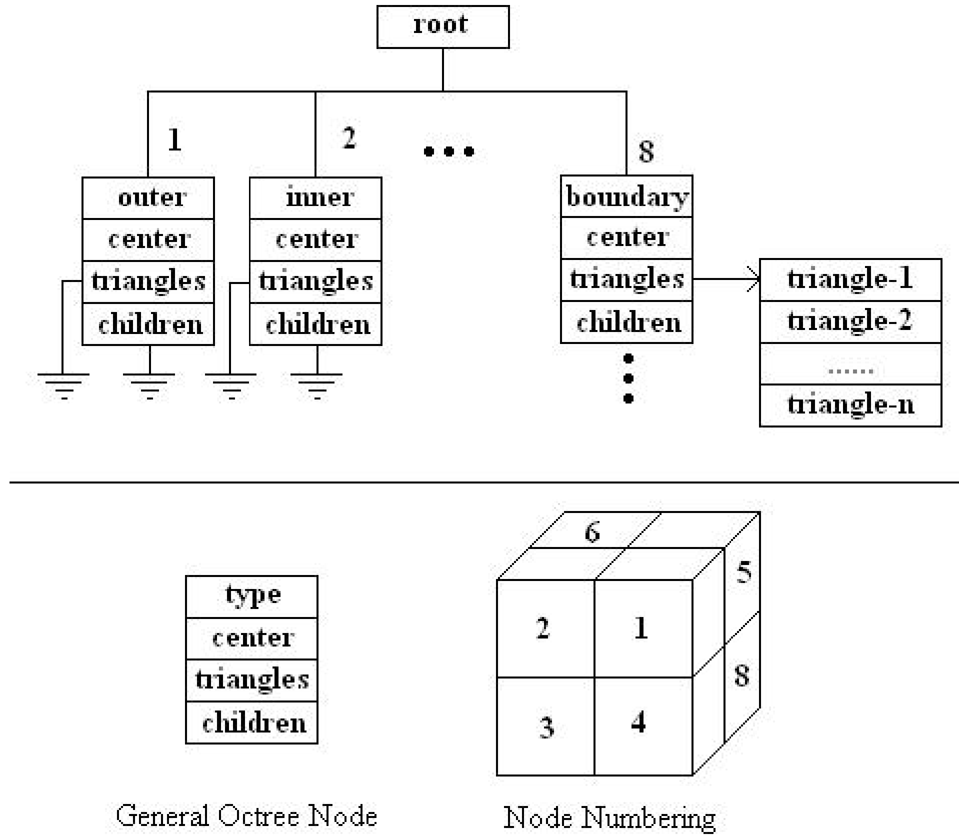

Figure 2.5. Octree data structure

- *Max-Tri*: maximum number of triangles assigned to the nodes.
- *Pointers*: total number of pointers used by all leaves.

As can be seen in Table 2.1 the number of pointers in general is much larger than the number of triangles, since a triangle is almost always intersected by more than one node, especially for deep octrees, where cubes of the octree are of the same size or smaller than the triangles they intersect.

Table 2.2 gives the statistics obtained by subdivision of the same sphere with different threshold values. We can easily observe that a low threshold value is likely to result in a large number of subdivisions, which has the potential to result in memory overflow problems. Although the maximum number of triangles given in Table 2.2

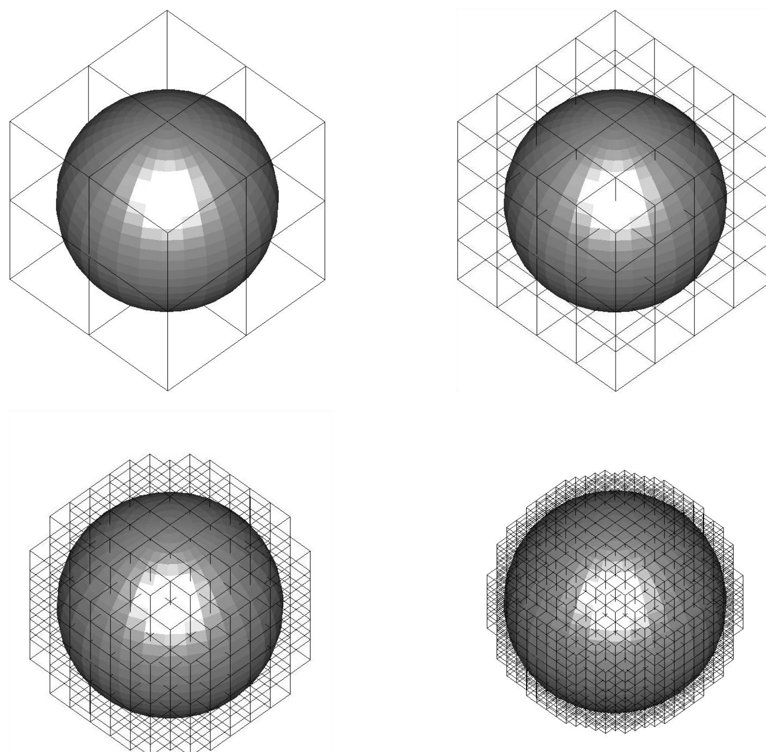

Figure 2.6. Adaptive octree subdivision for sphere

Depth	Total	Leaves	Occ.	Min-Tri	Max-Tri	Pointers
1	9	8	100%	2669	2950	22476
2	73	64	87.5%	79	1285	24332
3	521	456	59.6%	9	763	28052
4	2697	2360	49.2%	2	513	35720
5	11977	10480	45.4%	1	398	52480

Table 2.1. Statistics for octree of the sphere (threshold = 1)

assigned are the same for all threshold values, the number of resultant nodes are significantly smaller for larger threshold values, e.g. the octree consists of 11977 nodes when subdivided with a threshold value of 1, whereas the total number of nodes is 297 when the subdivision is done with a threshold value of 400.

As a result for large threshold values, the number of nodes is reduced without affecting the maximum number of triangles assigned to the nodes of the octree, which is an important parameter in evaluating the worst-case complexity of the octree-

Threshold	Total	Leaves	Occ.	Min-Tri	Max-Tri	Pointers
1	11977	10480	45.4%	1	398	52480
50	3017	2640	60.3%	2	398	39320
100	1417	1240	62.6%	4	398	33688
150	777	680	65.9%	9	398	30720
200	713	624	64.1%	9	398	30328
250	553	484	63.6%	9	398	29004
300	553	484	63.6%	9	398	29004
350	489	428	62.6%	9	398	28348
400	297	260	63.0%	11	398	26740

Table 2.2. Statistics for cctree of the sphere (depth = 5)

performance in various queries. Based on this observation a heuristics similar to the one outlined below, which can the node usage without worsening the worst-case performance, can be used:

1. Select a threshold value of 1 and perform the adaptive subdivision until the number of maximum triangles is reduced to an acceptable level.
2. Perform another adaptive subdivision but this time with a threshold value equal to the maximum number of triangles found in the first subdivision at the desired depth.

Table 2.3 presents octree generation statistics including the mean and standard deviation values for the distribution of the number of triangles among the leaf-nodes for the benchmark models shown in Figure 2.7. For each model we have applied our simple heuristic algorithm to optimize the node utilization. Note that when the octree is subdivided with a high threshold value, the resultant octree has less number of total nodes and the percentage of the occupied nodes increases, meaning that the effectiveness of the octree increases too.

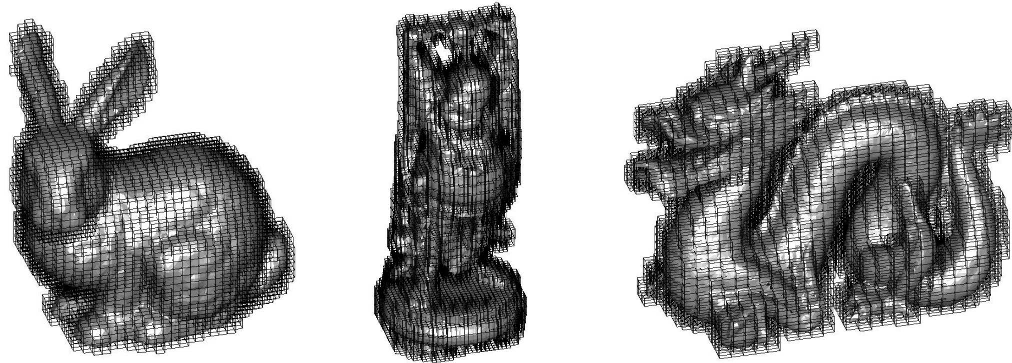

Figure 2.7. Adaptive octree subdivisions of various benchmark models

Model	Triangles	Threshold	Total	Leaves	Occ.	Mean	Std.
Bunny	16301	1	11137	9745	53.3%	9.03	10.36
Bunny	16301	47	5905	5167	73.5%	14.74	11.97
Buddha	15536	1	16297	14260	59.2%	8.70	9.48
Buddha	15536	61	3273	2864	81.0%	23.20	17.09
Dragon	47794	1	14257	12475	55.9%	17.32	19.86
Dragon	47794	89	7545	6602	77.8%	29.16	22.71

Table 2.3. Octree statistics for various benchmark models (depth = 5)

2.1.2. Advantages & Applications

The most important advantage of using octrees is that there is a high spatial-coherence among the cubes of an octree. This fact is due to the subdivision of a *parent* node into 8 equal-sized *child* nodes with the same orientation. The spatial coherence of the octree can effectively be utilized even if such a coherence is missing among the features of object subject to the subdivision.

Octrees are used in variety of applications for occlusion and view-frustum culling [10, 11], for fast ray-primitive intersection calculations in ray-tracing and ray-casting [12, 13, 14], for interference detection [15, 16] and also in CAD/CAM/CIM applications [17, 18, 19, 20]. In all these applications, octree provides a means to eliminate a large percentage of primitives at early stages of the processing, thus reducing load for the main processing pipeline.

3. OCTREE GENERATION USING MINKOWSKI SUMS

When generating an octree for triangular meshes, every triangle in the scene is assigned to a node of the octree by checking if the respective cube is intersected by that triangle. This requires a very large number of computationally heavy intersection tests.

Our method for constructing an octree, is based on the use of Minkowski sums for intersection calculations. Since Minkowski sums are invariant under translation, computing the Minkowski sum of a triangle and a single cube would be sufficient for testing the intersection of that triangle with all the cubes of the octree at a certain depth. This is due to the fact that the cubes of an octree are nothing but translated copies of a prototype cube at a given depth of the tree.

The other computational simplification in intersection calculations comes from the observation that a node at a certain depth has the same orientation but only half the size of its parent node and hence Minkowski sums of a triangle and a node with different depth values produce topologically similar polyhedra.

Selection of Minkowski sums for intersection calculations is not solely on the basis of computational efficiency. Minkowski sum provides important geometrical information other than the intersections, that can be used in answering additional queries such as those related to proximity and penetration depth. Also Minkowski sums provide a general framework for detecting intersections for different types of geometric entities.

3.1. Minkowski Sums

If A and B are two convex polyhedra in R^3, then the Minkowski sum of A and B is defined as the convex polyhedron

$$M = A \oplus B = \{a + b \mid a \in A, b \in B\} \tag{3.1}$$

In other words M is the union of all translated polyhedra B^a, such that $M = \bigcup_{a \in A} B^a$, where B^a is the polyhedron translated by a single point a, $a \in A$.

In graphic and haptic rendering, Minkowski sums are mainly used in the context of collision detection. When checking collision for two objects, one of the objects (the pivot object) is shrunk to a point while the other is grown (dilated) by the amount equal to that of the pivot (the grown object actually is the Minkowski sum of the two objects). This process allows the detection of collisions between these objects simply by checking if the shrunk point is inside the grown object.

If the objects subject to collision tests undergo only translations with respect to each other, the growth operation needs to be performed only once. The single copy of the grown object can be tested against any reference point to answer all the collision queries, instead of costly object-object intersection tests. In addition to detecting collisions, Minkowski sums can also be used to answer directional penetration depth queries.

In the literature, various methods have been proposed to compute the Minkowski sum of two convex polyhedra in R^3. Flato and Halperin [21] present algorithms for robust construction of planar Minkowski sums. Their methods are based on an Arrangement package of a software library called Computational Geometry Algorithms Library (CGAL). Guibas et al. [22] introduced the definition of the convolution operation based on the kinetic framework in two dimensions, which is a superset of the Minkowski sum operation, and its use in a various algorithmic problems.

An output-sensitive algorithm for computing Minkowski sums of polytopes was introduced in [23]. Gritzmann and Sturmfels [24] obtained a polynomial time algorithm in the input and output sizes for computing Minkowski sums of k polytopes in R^d for a fixed dimension d, while Fukuda [25] provided an output sensitive polynomial algorithm based on linear programming concepts. Ghosh [26] presented a unified algorithm for computing 2D and 3D Minkowski sums of both convex and non-convex polyhedra based on a slope diagram representation. Computing the Minkowski sum

amounts to computing the slope diagrams of the two objects, merging them, and extracting the boundary of the Minkowski sum based on the merged diagram. Bekker and Roerdink [27] apply a few variations on this idea, where the slope diagram of a 3D convex polyhedron is represented as a 2D object, thus reducing the problem to a lower dimension. Fogel and Halperin [28] follow the same approach, where they first construct the dual representations of both 3D convex polyhedra under a special type of Gaussian Map called Cubical Gaussian Map (CGM), overlay the two planar arrangements of these dual representations, from which they pass back to the primal representation of the desired Minkowski sum. Their method results in significant performance improvements compared to other methods they list.

3.2. A Geometric Method for Computing Minkowski Sums

The methods proposed in the literature for computing Minkowski sums of general polyhedra usually involve utilization of complex data structures requiring costly initializations and updates and usually have high time-complexities. In this section, we propose a simple method for calculating the Minkowski sums for simple geometric primitives such as the ones used in our octree generation framework, namely for triangles and cubes.

Geometrically, the Minkowski sum of two polyhedra is nothing but the convex hull of a set of vertices obtained by placing copies of one polyhedron at all vertices of the other. This process is shown in Fig. 3.2 for two simple 2D convex objects, namely a triangle and a rectangle (Fig. 3.1). The bold line in the Fig. 3.2 illustrates the Convex Hull.

The set of vertices $V = \{v_i\}$, whose convex hull is to be computed, can be given by

$$V = (V_T \times V_R) \cup V_T \tag{3.2}$$

where V_T is the set of all vertices of the triangle, V_R is the set of all vertices of the

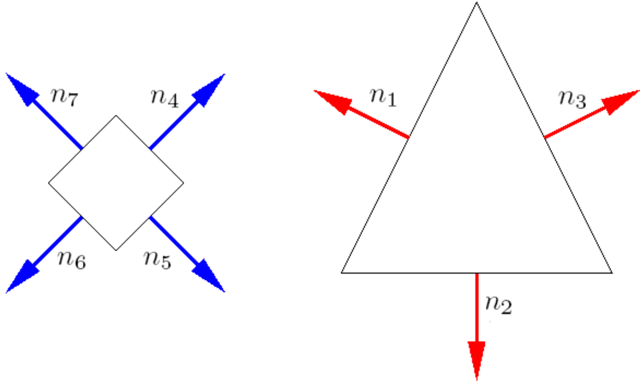

Figure 3.1. Triangle and Rectangle and their normals

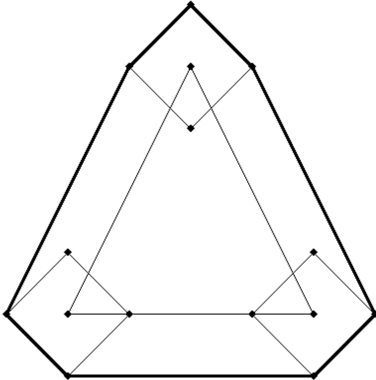

Figure 3.2. Convex hull of all vertices from the set V

rectangle and $\times$ denotes the Cartesian product operation.

The geometric method we describe generates the convex hull directly by translating lines in 2D space (planes in 3D space), instead of generating the convex hull from the set of vertices using conventional convex hull algorithms, like Graham's method in 2D or Incremental Method in 3D [29], which have large computational overheads when applied to simple convex objects.

Our method is based on the fact that every edge normal in the 2D Minkowski sum comes from the set of edge normals of both polygons, i.e. every edge in the Minkowski sum is parallel either to an edge of the triangle or to an edge of the rectangle. The vertices in the Minkowski sum are those that are extremal along these normals.

For the example given in Fig. 3.1, the set of normals N is given by

$$N = N_T \cup N_R \tag{3.3}$$

where N_T is the set of normals of the triangle and N_R is the set of normals of the rectangle. That is, N consists of the set $\{n_1, n_2, \cdots, n_7\}$, where the first 3 normals come from the triangle and, the remaining ones come from the rectangle.

Minkowski sum calculation for this example is composed of the following two cases:

(1) Calculating edges parallel to those of the triangle: This is simply achieved by shifting a line passing through the origin with normal $n_i \in N_T$, in the direction of its normal until the most distant vertex is found.

Figure 3.3 shows this process for the line (stippled) with normal n_1. Here v_1 and v_2 are the extreme vertices, also the end-points of the edge e_1. The set of extreme vertices for a given normal n_i is thus given by

$$\{\, v_k \mid v_k \cdot n_i = \max_j \ v_j \cdot n_i \ , \quad v_j \in V,\ n_i \in N \,\} \tag{3.4}$$

where V is the set of all vertices given by Eq. 3.2. Repeating this procedure for all normals $n_1 \ldots n_3$ coming from the triangle, we get the edges shown in Fig. 3.4.

(2) Calculating the edges parallel to the edges of the rectangle: Here the same procedure is repeated for the lines with normals coming from the rectangle. Shifting the stippled line in Fig. 3.5, which is parallel to one of the edges of the rectangle along the normal n_4 we get extreme vertices v_3 and v_4, and therefore the edge e_4 of the Minkowski sum.

Repeating this procedure for all normals $n_4 \ldots n_7$ coming from the triangle, we get the edges shown in the Fig. 3.6.

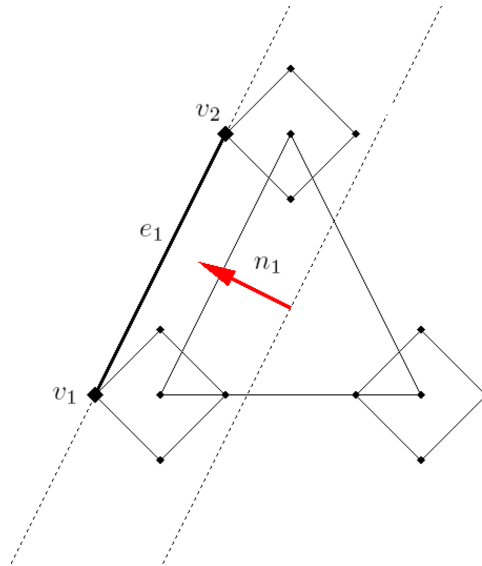

Figure 3.3. The stippled line is shifted along n_1 until v_1 and v_2 are hit

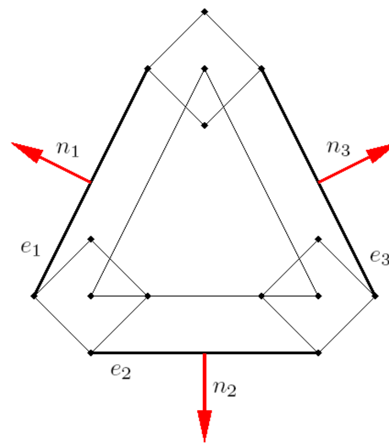

Figure 3.4. Edges of the Minkowski sum parallel to the edges of the triangle

3.2.1. Extending the Geometric Method to 3D

The Minkowski sum of two polyhedrons A and B, where B is the pivot polyhedron can be computed by

1. Generating the vertex set V from which the extreme vertices will be chosen
2. Generating the normal set for all possible facets of the resultant polyhedron
3. Generating the facets by pushing the planes along their normals starting from the origin until extreme vertices along their corresponding normals are encountered

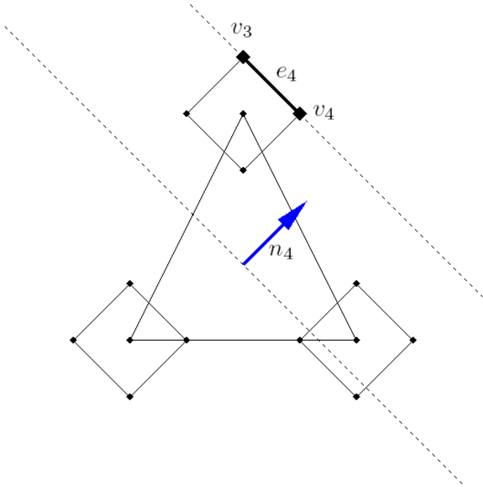

Figure 3.5. The stippled line is shifted along n_4 until v_3 and v_4 are hit

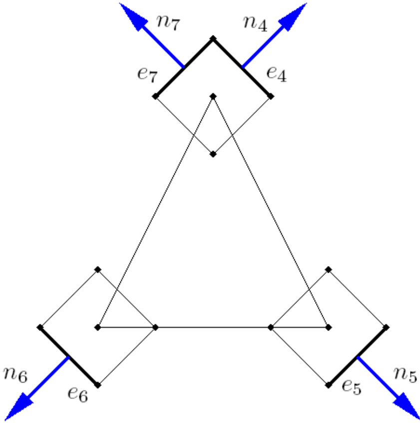

Figure 3.6. Edges of the Minkowski sum parallel to the edges of the rectangle

Now assume an example where the polyhedron A is a cube and B is an octahedron. We can generate the vertex set V simply by placing one copy of B at each vertex of A, as shown in Fig. 3.7. V can be mathematically defined as

$$V = (V_A \times V_B) \cup V_A \tag{3.5}$$

where

$$V_A \times V_B = \{v_{ij}\} = T_i \cdot v_j^B \qquad i = 1 \ldots |V_A|,\ j = 1 \ldots |V_B| \tag{3.6}$$

where v_j^B is the jth vertex of B and T_i is the homogenous transformation matrix defined as

$$T_i = \begin{pmatrix} 1 & 0 & 0 & (v_i^A)_x \\ 0 & 1 & 0 & (v_i^A)_y \\ 0 & 0 & 1 & (v_i^A)_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

where v_i^A is the ith vertex of the polyhedron A.

In considering the second step, we have bear in mind that the facets in the Minkowski sum have normals coming from three possible sources (as opposed to the two sources in 2D case):

- normals coming from the facets of A (Fig. 3.8)
- normals coming from the facets of B (Fig. 3.9)
- new normals coming from facets generated by sweeping the edges of B along the edges of A (Fig. 3.10)

The normal set N_M of the Minkowski sum M is a subset of the normal set N, or $N_M \subseteq N$, where N is given by

$$N = N_A \cup N_B \cup N_{AB} \tag{3.8}$$

where N_{AB} consists of normals $\{n_{ij}\}$ with

$$n_{ij} = e_i^A \times e_j^B \qquad i = 1 \ldots |N_A|,\ j = 1 \ldots |N_B| \tag{3.9}$$

where e_i^A is the direction vector of the ith edge of the polyhedron A and e_j^B is the direction vector of the jth edge of the polyhedron B. In step 3, we generate a set of planes each with a normal drawn from the set N and re-position these planes by *pushing* them until extreme vertices from the set V are encountered. This is done by

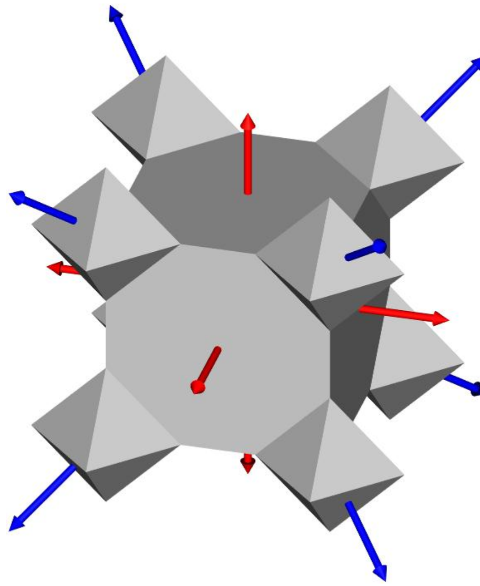

Figure 3.7. Multiple Copies B placed at all vertices of A

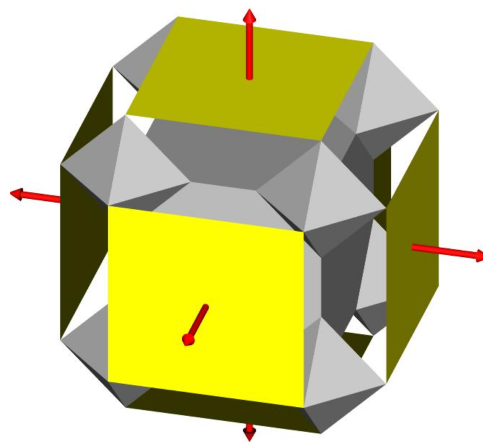

Figure 3.8. Facets parallel to the facets of A

projecting each vertex onto the normal vector of the corresponding plane and keeping track of all vertices forming the same *maximum* dot-product with the corresponding plane-normal. During this stage of the algorithm the candidate plane is assigned the number of extreme vertices it contains, i.e. all vertices forming the same extremal dot-product with the candidate plane-normal are assigned to that candidate plane. Once this set contains at least 3 non-collinear vertices, the plane is guaranteed form a facet of the Minkowski sum. Another way of stating this is as follows:

$$V_{F_i} = \{\, v_k \mid v_k \cdot n_i = \max_j \; v_j \cdot n_i \,, \quad v_j \in V,\; n_i \in N_M \,\} \tag{3.10}$$

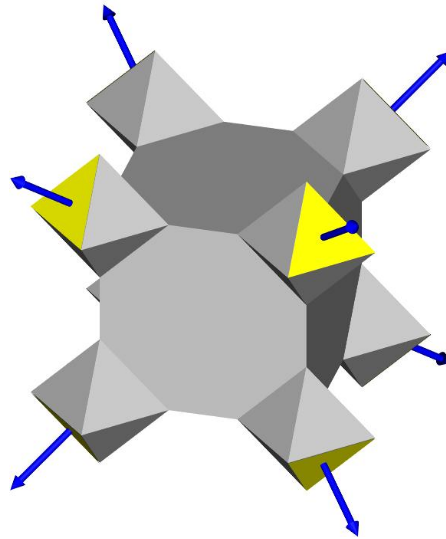

Figure 3.9. Facets parallel to the facets of B

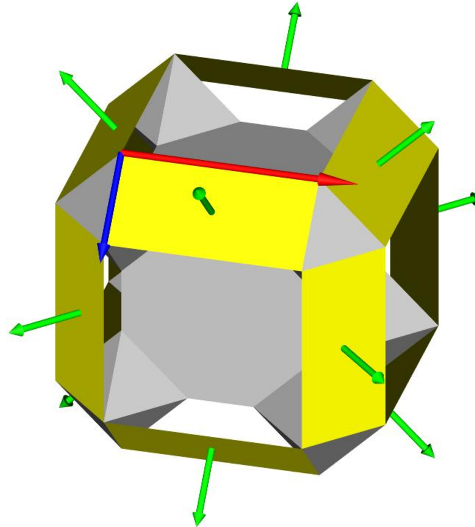

Figure 3.10. Facets formed by sweeping the edges of B along the edges of A

where V_{F_i} is the set of vertices belonging to the ith facet of the final Minkowski sum and V is the set of all vertices obtained in Step 1 and is given by Eq. 3.5.

Since the set of normals contains normals from A and B, the planes having normals drawn from N_A or N_B are guaranteed to form facets in the Minkowski sum after being translated to their respective extreme vertices. This can be seen in Figs. 3.8 and 3.9, where the normals of all facets of A and B appear as facet-normals of the Minkowski sum.

In Figure 3.10, it can be seen that additional facets are formed by pushing planes having normals drawn from the set N_{AB}. However, it is important to realize that not all of the normals in the set N_{AB} have to form facets in the final Minkowski sum. In fact, a plane having a normal from the set N_{AB}, cannot form a facet if

- A candidate plane is assigned less than 3 non-identical vertices
- If (at least 3) vertices assigned to the candidate plane are collinear

In fact, this is why the normal set of the Minkowski sum is a subset of set of all normals N. The rules above are implemented in [27, 28] using a Gaussian Map. A plane having the normal obtained by taking the cross-product of the direction vectors of two edges will contribute to the final Minkowski sum if their dual images under the Gaussian Map intersect. But in our application, since the number of normals, thus the number of the tests to be performed are very small, constructing the dual representations is a relatively time consuming process compared to our method.

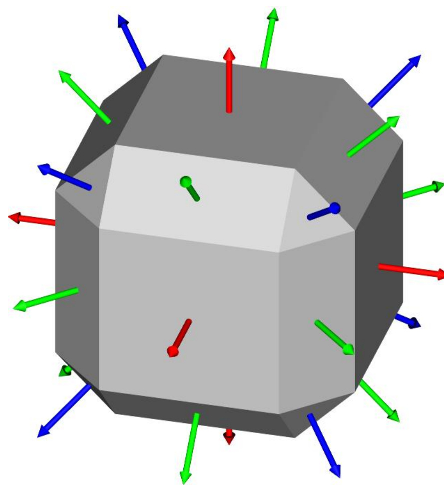

Figure 3.11. The Final Minkowski Sum

The final Minkowski sum is shown in Fig. 3.11. In this Figure, red arrows represent the normals coming from the static polyhedron A, blue arrows represent those from the pivot polyhedron B and green arrows represent the normals of facets obtained by sweeping. Note that if the visualization of the Minkowski sum is needed, some vertices need to be eliminated and only those vertices that are assigned to at least 3 distinct

incident facets need to be retained.

The pseudo-code for the geometric method as a whole is given below:

1 Generate the vertex set V

2 Generate the normal set N

3 FOR($i = 1 \ldots |N|$)

  3.1 clear *maximal_vertex_list* and *minimal_vertex_list*

  3.2 $max_dot_product = min_dot_product = v_1 \cdot n_i$

  3.3 FOR($j = 2 \ldots |V|$)

    3.3.1 $current_dot_product = v_j \cdot n_i$

    3.3.2 IF($max_dot_product = current_dot_product$)

      $maximal_vertex_list \leftarrow \mathrm{v}_j$

    3.3.3 ELSE IF($max_dot_product < current_dot_product$)

      clear *maximal_vertex_list*

      $maximal_vertex_list \leftarrow \mathrm{v}_j$

      $max_dot_product = current_dot_product$

    3.3.4 ELSE IF($min_dot_product = current_dot_product$)

      $minimal_vertex_list \leftarrow \mathrm{v}_j$

    3.3.5 ELSE IF($min_dot_product > current_dot_product$)

      clear *minimal_vertex_list*

      $minimal_vertex_list \leftarrow \mathrm{v}_j$

      $min_dot_product = current_dot_product$

    END IF (3.3.2)

  END FOR (3.3)

  3.4 IF($|maximal_vertex_list| \geq 3$)

    3.4.1 IF(vertices in *maximal_vertex_list* are *NOT* collinear)

      add the plane with normal $\mathbf{n_i}$ and support $\mathbf{v_j}$

    END IF (3.4.1)

  END IF (3.4)

  3.5 IF($|minimal_vertex_list| \geq 3$)

3.5.1 IF(vertices in *minimal_vertex_list* are *NOT* collinear)

add the plane with normal $-\mathbf{n_i}$ and support $\mathbf{v_j}$

END IF (3.5.1)

END IF (3.5)

END FOR (3)

3.3. Computational Complexity of Minkowski Sum Calculation

3D Minkowski sums can also be calculated using convex hull algorithms in 3D. Here we will compare its computational cost with the method we have used.

3.3.1. Geometric Minkowski Sum Calculation

The first step of the geometric method is to calculate the Cartesian product of the vertices of the triangle and the cube. Since the vertex set V consists of $8 \times 3 = 24$ vertices obtained by pairwise vector addition of vertices of both objects, this operation takes $24 \times 3 = 72$ flops.

In the second step, we need to calculate the set of normals that are used to define the bounding planes of the Minkowski sum. Since in our modified algorithm, we want to perform a single dot-product operation instead of two separate dot-product operations to halve the number of dot-product operations, we construct the normal set N as follows

- 1 normal comes from the triangle
- 3 normals come from 3 principal directions of the cube
- $3 \times 3 = 9$ normals come from cross-products

with a total of 13 normals. Only the last 9 normals need to be calculated, so we add the cost of taking the cross-product of the principal directions of the cube and the direction vectors of the edges of the triangle. For these operations we need 9×3 2D-determinant operations. Since the cross-product operation requires calculation of

a single 2D-determinant for each component, the result is $9 \times 3 \times (2prods + 1sub) = 81$ flops.

In the main loop of the algorithm we take dot-products of all vertices with all of the normals, so the total number of dot-products evaluated is $24 \times 13 = 312$ with the total $312 \times 5 = 1560$ flops. Thus to calculate the Minkowski sum of a triangle and a cube we need to perform $72 + 81 + 1560 = 1713$ flops in the worst-case.

3.3.2. Incremental Convex Hull Construction

Here we will roughly estimate the cost associated with constructing the Minkowski sum using the incremental Convex Hull construction method as described in [29]. Since a detailed description of the incremental method is outside the scope of the present thesis, the interested reader should refer to [29] for an introduction to convex hulls, their applications and a detailed implementation in pseudocode.

The incremental convex hull construction method starts by constructing a tetrahedron from an initial set of 4 non-coplanar points. The algorithm takes all remaining points one by one and at each step, it identifies which triangles of the current polyhedron are visible and which are not, which results in a *shadow boundary* between the visible and non-visible triangles. It then eliminates all faces visible from the current point and constructs new triangles by connecting each edge on the identified shadow boundary with the current point. It continues with the next point in the list and performs the same set of operations over and over again until no points are left in the point-list. The algorithm can be roughly sketched as follows:

1. find a set of 4 non-coplanar points and construct the tetrahedron
2. take the next point in the remaining point-list
3. identify all visible and invisible triangles from the current point
4. identify the *shadow boundary*
5. delete all visible faces
6. construct all new triangles tangent to the remaining triangles along the identified

shadow boundary

7. go to the step 2 until there are no points are left

Note that even if a single visible triangle is identified at each iteration, which is by the way the best-case, at least 3 new triangles are added to the polyhedron, because the 3 edges of that visible triangle constitutes the shadow boundary. If all triangles are invisible from a certain point, then that point must be inside the current polyhedron and in such a case the polyhedron won't be updated.

For a worst-case scenario, we assume that there are 4 non-coplanar vertices which are the vertices of the initial tetrahedron such that all of the remaining 20 points fall outside of this initial tetrahedron.

Assuming the number of triangles added at each iteration is fixed and equal to 3, the number of operations involved in the calculation of the normals of the 3 triangles is $3 \times (6prods + 3subs) = 27$ flops. An additional dot-product needs to be calculated to ensure that the normal of the each new triangle points outwards, so that we need to add $3 \times 5 = 15$ to 27 giving a total of $27 + 15 = 42$ flops per iteration.

Since there are 20 vertices left outside of the tetrahedron, we need to perform $20 \times 42 = 840$ flops. Note that we also need to perform additional operations to ensure correct connectivity among the new and old triangles as well as correct normal orientations. These are not considered here.

In the best case, in every iteration the number of triangles of the polyhedron increases by 2 (we remove 1 triangle and add 3 new triangles which results in a net increase of 2 triangles per iteration). So to add all triangles we need $4 + \ldots + 42 = 440$ dot-products ($440 \times 5 = 2200$ flops), giving a total of $840 + 2200 = 3040$ flops in the case of minimum triangle updates for a worst-case scenario where all points fall outside the initial tetrahedron. Even if half of these visibility queries are performed, which roughly corresponds to the case where half of the remaining vertices fall outside the tetrahedron and other half inside the tetrahedron, we would need to perform $840 + 2200/2 =$

$840 + 1100 = 1940$ flops, which is significantly greater than the maximum number of flops in the worst-case for the geometric method.

3.4. Octree Generation Using Minkowski Sums

An important structural property of the octree is the fact that cubes of an octree at a fixed depth are nothing but dimensionally equal copies displaced relative to each other with the same orientation. In this respect we can think of the cubes of an octree as multiple copies of a single prototype cube. Therefore it would be sufficient to construct the Minkowski sum only once, and use it multiple times to evaluate all the intersection queries for other cubes placed at different locations in the octree.

In a Minkowski sum-based intersection test, we only need to evaluate a simple point-in-polyhedron test, i.e. we need to check if the center of the cube falls inside the Minkowski-Sum, which is bounded by planes found with our method. This requires simple signed distance computations that can be performed with a single dot-product operation per bounding plane.

The additional advantage of using Minkowski sums is based on the fact that all cubes of an octree have the same orientation, i.e. their face-normals or principal directions are all the same. This means that for cubes having different depth values, the only thing that changes is the size of the cubes. Looking at Figures 3.12 and 3.13 it can be easily seen that the Minkowski sums of cubes having the same orientation but different sizes have the same topology, i.e. their facets have the same set of normals and they are essentially the same geometric entity except their facets-sizes, edge-lengths and vertex locations.

Having observed this so-called *topological invariance*, it is easy to see that once we calculate the Minkowski Sum of a triangle and a cube *at any depth*, then Minkowski sums of the cubes at other depths with the same triangle can quickly be calculated by keeping track of the index of the vertex (being the extreme vertex found by shifting the corresponding plane along its normal) selected as the origin of the bounding plane

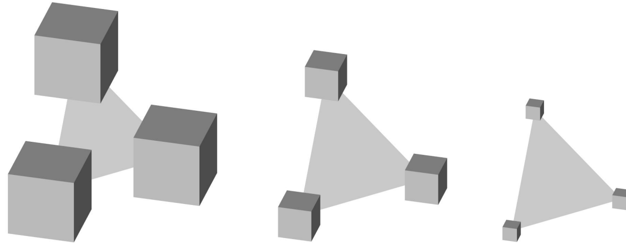

Figure 3.12. Copies of cubes of different sizes placed at triangle-vertices

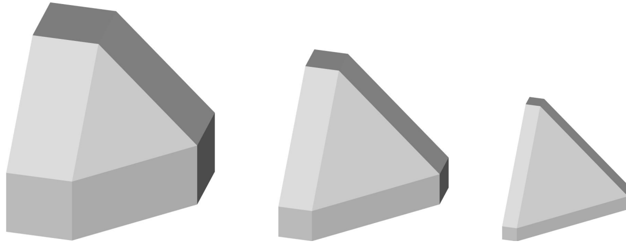

Figure 3.13. Minkowski sums showing a *topological invariance*

in the final Minkowski Sum. We will refer to this method as *Index Tracking*.

3.5. Index Tracking

Our method outputs only a set of bounding planes, that are represented by their normals and their support-points. The support of each plane originates from the set of vertices V obtained by taking the Cartesian product of the set of vertices of the cube and the triangle. Since for each depth value the vertex set V is obtained in the same manner, we label each vertex in V with a global vertex index that would indirectly encode the way the vertex is calculated. Another way to state this is as follows: let i denote the local index used to identify the ith vertex of the cube and let j denote the local index used to identify the jth vertex of the triangle. Then the vertex set $V = \{v_k\}$ consists of vertices given by

$$v_k = v_i^c + v_j^t \qquad i = 1 \dots 8,\ j = 1 \dots 3 \tag{3.11}$$

where $k = i + 8 \cdot j$, v_i^c and v_j^t are the ith and j^{th} vertices of the cube and the triangle, respectively. Since it has been shown that planes of Minkowski sums at different depths have supporting vertices that are obtained by the same labeling process and that the

supporting vertex of a particular plane is observed to have the same global index, that global index encodes how the vertex has been obtained. In other words it encodes the information which vertex of the cube should be added to which vertex of the triangle. This information can be recovered using $i = k \bmod 8$ and $j = \lfloor k/8 \rfloor$.

Figure 3.14 depicts this for the example used for explaining the geometric Minkowski sum calculation method in 2-dimensions. In the figure the vertices of the copies of the boxes at two different depths are enumerated in the *same order*. The gray edge connecting the vertices 3 and 7 in Figure 3.14(a) has the same normal as the gray edge connecting the vertices with the same index-values in Figure 3.14(b).

The support of both gray edges in Figure 3.14 is the vertex v_7, where the global index k is 7. To find out the local indices of the box and triangle, whose sum add up to v_7, using our formula we get: $i = 7 \bmod 4 = 3$ and $j = \lfloor 7/4 \rfloor = 1$. This says that to recalculate the support of the gray edge at the next depth, we need to add 3rd vertex of the box and the 1st vertex of the triangle, where these local indices start from 0.

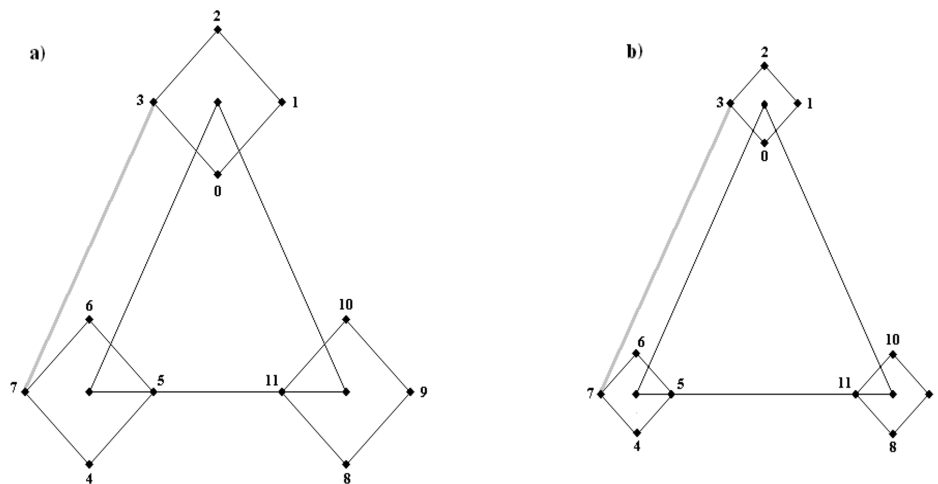

Figure 3.14. Index tracking

3.6. Computational Complexity of Intersection Detection

Here we compare the method of detecting intersections between triangles and cubes of an octree using Minkowski sums with two other methods. One is a general

non-optimized brute-force intersection test, the other is a geometry-specific optimized separating plane test.

3.6.1. Intersection detection using Minkowksi sums

Minkowski sum is essentially the intersection of closed negative half-spaces of the supporting planes of the facets of the Minkowski sum, each with the form

$$a_i x + b_i y + c_i z = d_i \tag{3.12}$$

where a_i, b_i, c_i are the components of the normal vector of the ith plane. A point p is inside the Minkowski sum M if

$$p \cdot n_i = a_i p_x + b_i p_y + c_i p_z \leq d_i \qquad \forall i \in \{1, \ldots, |P_M|\} \tag{3.13}$$

and p is outside the Minkowski sum M, then p is in the positive open half-space of one of the bounding planes, i.e. if

$$p \cdot n_i = a_i p_x + b_i p_y + c_i p_z > d_i \qquad \exists i \in \{1, \ldots, |P_M|\} \tag{3.14}$$

where p_x, p_y, p_z are the coordinates of the point p and P_M is the set of all planes bounding the Minkowski sum M. So it is sufficient to evaluate $|P_M|$ dot-products to check if a point p inside the Minkowski sum. The maximum number of dot-products to be calculated is 26. This is the size of the normal set N from where the normals of the facets of the Minkowski sum come. Since 6 normals come from the faces of the cube, 2 normals come from back and front faces of the triangle and $6 \times 3 = 18$ normals come by convolving the edges of the triangle and those of the cube. This is the total number of bounding planes of the Minkowski sum of a triangle and a cube. As a result, the maximum number of operations in terms of basic algebraic floating-point operations (*flops*) is $26 \times (3prods + 2adds) = 130$ flops in total. In fact this is the worst-case and most Minkowski sums will have less than 26 bounding planes.

3.6.2. Intersection detection using classical tests

In classic brute-force intersection detection each edge of one object needs to be checked for intersection against each face of the other object and vice versa, we have to consider the two following cases:

- edges of the *triangle* intersecting the faces of the *cube*
- edges of the *cube* intersecting the faces of the *triangle*

For the first case we need to perform $3 \times 6 = 18$ intersection tests and for the second one $12 \times 1 = 12$, totaling to a maximum of 30 intersection tests, which is the absolute maximum number of intersection tests that need to be performed in worst-case, whereas the worst-case scenario for the Minkowski-sum based intersection test is 26.

In order to find the intersection point of an edge and a face we need to consider the corresponding supporting line and plane, respectively. So we need to check the intersection of a line given by $x = p + tu$ and a plane given by $nx = d$. We define the direction vector u as the difference vector pointing from the start-vertex to the end-vertex of the corresponding edge, so that the edge corresponds to the portion of the line obtained by running the parameter t from 0 to 1. By substituting the line equation into the equation of the plane, we get

$$n(p + tu) = d \tag{3.15}$$

from which we obtain the intersection parameter given by

$$t_i = \frac{d - n \cdot p}{n \cdot u} \tag{3.16}$$

Note that even for calculating the intersection parameter 2 dot-products are required. Since the dot-product $n \cdot u$ can be pre-calculated, it will not be considered. The total number of flops for calculating the intersection parameter t_i is 1 subtraction *plus* 1

division *plus* 1 dot-product (3 prods + 2 adds), totaling to $1 + 1 + 5 = 7$ flops.

Before evaluating the intersection point we have to check if t_i is inside $(0, 1)$. We exclude the end-points due to several *degenerate cases* shown in Figure 3.15. Detection of intersection with an end-point of an edge is not sufficient to guarantee intersection of two objects. In such degenerate cases, one cannot conclude if the triangle intersects the volume (left-most case in Figure 3.15) or if it is on the boundary of one the face of the cube (middle case in Figure 3.15) or if the triangle falls outside of the cube (right-most case in Figure 3.15) without performing further evaluations other than pair-wise intersection tests.

If $t_i \in (0, 1)$, then next thing to do is to calculate the intersection point because we need to check if that intersection point is inside the boundary enclosing the face. To calculate the intersection point we use the following expression

$$x_i = p + t_i\, u \tag{3.17}$$

which involves 1 scalar product (3 prods) and 1 vector addition (3 adds) operations, totaling to 6 flops. With intersection parameter calculation we need at least $7+6 = 13$ flops. Since in the worst case we need to perform 30 intersection tests, the minimum number of flops will sum up to $13 \times 30 = 390$ flops. In order to check if the intersection

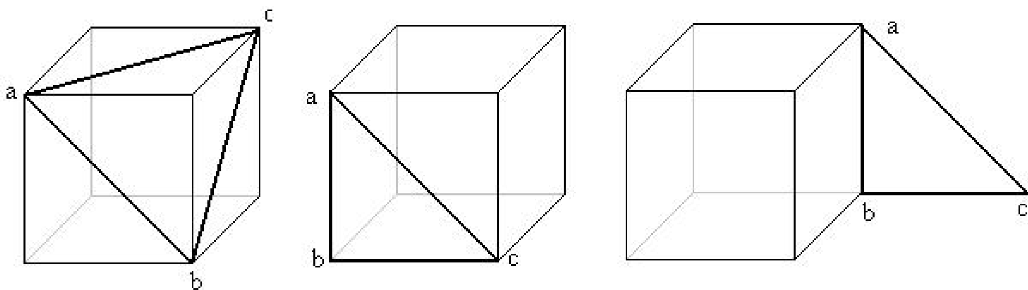

Figure 3.15. Degenerate cases in triangle and cube intersection

point is within the facet of the cube we need to perform 4 additional dot-product operations per cube-face. Perform this validity operation for all 18 intersection tests we would end up with $4 \times 18 = 72$ dot-products. A similar validity test needs to

be performed to check if the intersection point is inside the triangle, an additional $3 \times 12 = 36$ dot-products need to be evaluated, totaling to a maximum of $72 + 36 = 108$ dot-product operations corresponding to $108 \times 5 = 540$ flops. Together with the intersection calculations we get in total $390 + 540 = 930$ flops excluding the degenerate cases.

3.6.3. Intersection Detection using Separating Axis Overlap Test

Another method for detecting intersections between a cube and a triangle is based on the idea of separating axis overlap test. This method is based on the following theorem from computational geometry: Two nonintersecting convex polyhedra can be separated by a plane that is either parallel to a face of one of the polyhedra or that contains an edge from each of the polyhedra.

In this respect it is necessary and sufficient to determine if two convex polyhedra intersect by examining the intersections of the projections of the polyhedra on lines that are perpendicular to the planes described in the theorem. If the minimal intervals containing the projections of the polyhedra onto one of these lines do not intersect, then the polyhedra themselves do not intersect, in which case that line is said to be a *separating axis* [30]. In the case of cube and triangle, the potential separating axes are formed by the following vectors:

- normal of the triangle denoted by n
- three principles directions coming from the cube denoted by a_1, a_2, a_3
- cross-products of the principles directions of the cube a_i and edges e_i of the triangle

where the edges of the triangle are given by $e_0 = v_1 - v_0$, $e_1 = v_2 - v_0$ and $e_2 = e_1 - e_0$. Counting the number of all potential separating axes we get $1 + 3 + 9 = 13$ potential separating axes. If we denote by l any of the potential separating axis, and by d the difference vector obtained by subtracting the center c of the cube from the first vertex v_0 of the triangle, i.e. $d = v_0 - c$, then the non-intersection test of triangle and cube is

guaranteed if $min(p_0, p_1, p_2) > R$ or $max(p_0, p_1, p_2) < -R$ holds. In these comparisons tests, the terms p_0, p_1, p_2, R are evaluated as shown in Table 3.1. In this table s_0, s_1, s_2 refer to the half-side lengths of the cube along the axes a_1, a_2, a_3, respectively.

l	p_0	p_1	p_2	R
n	$n \cdot d$	p_0	p_0	$s_0\|n \cdot a_0\| + s_1\|n \cdot a_1\| + s_2\|n \cdot a_2\|$
a_0	$a_0 \cdot d$	$p_0 + a_0 \cdot e_0$	$p_0 + a_0 \cdot e_1$	s_0
a_1	$a_1 \cdot d$	$p_0 + a_1 \cdot e_0$	$p_0 + a_1 \cdot e_1$	s_1
a_2	$a_2 \cdot d$	$p_0 + a_2 \cdot e_0$	$p_0 + a_2 \cdot e_1$	s_2
$a_0 \times e_0$	$a_0 \times e_0 \cdot d$	p_0	$p_0 + a_0 \cdot n$	$s_1\|a_2 \cdot e_0\| + s_2\|a_1 \cdot e_0\|$
$a_0 \times e_1$	$a_0 \times e_1 \cdot d$	$p_0 - a_0 \cdot n$	p_0	$s_1\|a_2 \cdot e_1\| + s_2\|a_1 \cdot e_1\|$
$a_0 \times e_2$	$a_0 \times e_2 \cdot d$	$p_0 - a_0 \cdot n$	$p_0 - a_0 \cdot n$	$s_1\|a_2 \cdot e_2\| + s_2\|a_1 \cdot e_2\|$
$a_1 \times e_0$	$a_1 \times e_0 \cdot d$	p_0	$p_0 + a_1 \cdot n$	$s_0\|a_2 \cdot e_0\| + s_2\|a_0 \cdot e_0\|$
$a_1 \times e_1$	$a_1 \times e_1 \cdot d$	$p_0 - a_1 \cdot n$	p_0	$s_0\|a_2 \cdot e_1\| + s_2\|a_0 \cdot e_1\|$
$a_1 \times e_2$	$a_1 \times e_2 \cdot d$	$p_0 - a_1 \cdot n$	$p_0 - a_1 \cdot n$	$s_0\|a_2 \cdot e_2\| + s_2\|a_0 \cdot e_2\|$
$a_2 \times e_0$	$a_2 \times e_0 \cdot d$	p_0	$p_0 + a_2 \cdot n$	$s_0\|a_1 \cdot e_0\| + s_1\|a_0 \cdot e_0\|$
$a_2 \times e_1$	$a_2 \times e_1 \cdot d$	$p_0 - a_2 \cdot n$	p_0	$s_0\|a_1 \cdot e_1\| + s_1\|a_0 \cdot e_1\|$
$a_2 \times e_2$	$a_2 \times e_2 \cdot d$	$p_0 - a_2 \cdot n$	$p_0 - a_2 \cdot n$	$s_0\|a_1 \cdot e_2\| + s_1\|a_0 \cdot e_2\|$

Table 3.1. Terms in separating-axis based intersection test [30]

Since all cubes of the octree have the same set of principal directions, the cubes-axes a_0, a_1, a_2 are the same for all cubes of the octree. Taking this into account we can pre-compute all cross- and dot-products and their absolute values in Table 3.1 for each triangle such that only the R terms need to be updated for each depth of the octree.

Once the constant terms are initialized, checking intersection of a cube and a triangle requires a single vector difference operation to calculate the difference vector $d = v_0 - c$ and calculating 13 dot-products to calculate the p_0 term (see the second column of the Table 3.1). All of these operations require 3 subtractions (for the difference vector), 13 dot-products (p_0 term) and 15 addition/subtraction operations (p_1 and p_2 terms) resulting in $3 + 13 \times 5 + 15 = 83$ flops in worst-case.

3.6.4. Conclusion for intersection detection

Our analysis has shown that the Minkowski sum based detection of intersections can produce performance not far from the performance of the separating axis based technique, which is optimized for the underlying cube-triangle intersection geometry. We must underline that the separating axis based method deals only with the intersection detection problem. In contrast to this, Minkowski sum can be also used to calculate both the penetration depth of the triangle into the cube or to find the closest distance of the triangle to the cube. This is a clear advantage in proximity queries.

Another advantage of using Minkowski sum based intersection test in our octree-framework is that since our geometric Minkowski sum calculation method can efficiently calculate the Minkowski sum of any simple geometric entity with the cubes of the octree, it is easier to use them in intersection detection of cubes and Voronoi regions, as shown in the next chapter. On the other hand, if method of separating axes is to be used to check intersection of Voronoi regions and cubes, then all intersection conditions must be formulated and optimized individually for each and every type of Voronoi region.

Note also that since the Minkowski sum is actually a polyhedron, when a proper set of partitioning planes are selected, the intersection tests have the potential to be performed much more faster. In fact, it is part of our future work to partition the facets of the Minkowski sum with an optimal set of partitioning planes such that the number of dot-product operations is minimized.

Table 3.2 gives a summary of the results of computational complexity analyses of intersection tests and Minkowski sum construction methods. In this table, all values correspond to the worst-case scenarios. But since the upper bound for the convex hull method is large and its algorithmic complexity along with its data structures are very complicated in comparison to the geometric method, the geometric method is a more effective method in terms of efficiency and ease of implementation when simple geometric primitives are involved.

Method	dot-products	cross-products	flops
Separating-axis-based intersection	13	-	83
Minkowski-sum-based intersection	26	-	130
Pairwise intersection	138	-	930
Geometric method	312	27	1713
Incremental convex hull method	440	60	3040

Table 3.2. Summary of computational complexities

3.7. Performance Statistics for Octree Generation

Figure 3.16 shows octrees generated for a spaceship model containing 33000 triangles. The performance figures for different methods are given in the Table 3.3. These results were obtained on an laptop computer (P4-3.2GHz) with 512 MB of RAM running under Windows XP.

Depth	Separating-Axis	Index Tracking	Minkowski Sum	Pairwise
1	1.047 sec.	27.422 sec.	27.329 sec.	200.687 sec.
2	1.672 sec.	38.328 sec.	51.672 sec.	408.204 sec.
3	2.484 sec.	49.578 sec.	77.000 sec.	642.140 sec.
4	3.296 sec.	60.953 sec.	103.266 sec.	922.171 sec.
5	4.453 sec.	74.500 sec.	129.235 sec.	1296.297 sec.

Table 3.3. Elapsed time for generating the octree

In this table, *Pairwise* employs classic intersection tests covering all degenerate cases, *Minkowski Sum* refers to our method based on Minkowski sums, *Index Tracking* refers to the optimized method with index-tracking and *Separating-Axis* refers to the seperating-axis based intersection test. The results given in Table 3.3 are in accordance with the findings of our complexity analysis.

Figure 3.16. AKIRA Spaceship [31] and its octree representations

4. PROXIMITY OCTREE

4.1. Need for proximity queries

A proximity query refers to locating the closest pair of features between two objects or an object and a point or detecting their inter-penetration along with the computation of a distance measure, which is either the separation distance between the two geometric entities in question or their penetration depth, respectively.

In comparison to locating interferences using intersection detection *at the time of or after* the objects collide, proximity queries present a more reliable method in the sense that they provide a means to estimate which objects are likely to contact *before* such a collision occurs by providing the corresponding distance measure. This is a more reliable method for detecting collisions and prevent their inter-penetration in dynamic environments where objects move at relatively high speeds or the stepping time of the application is very small (as for the case of haptic rendering) and reliable estimates for potentially colliding objects need to be made.

In virtual reality applications, one of the most commonly performed query is locating the feature of an object closest to a point of interest along with finding the distance of that point to the closest feature. Since in a brute-force method, evaluating all distance values between the point of interest and all features of an object would take a large number of calculations, there is a need for an effective culling mechanism to get rid of the features that are guaranteed to be more distant than a small set of potentially closest features.

4.2. Related Work

An extensive amount of work has been conducted on proximity queries and collision detection between polyhedral objects. In [32, 33] Lin presents a fast incremental method to compute the distance between two convex polyhedra by constructing Voronoi

regions of all features of a polyhedron to find and keep track of the closest features for a pair of convex polyhedra. In [34], Ehmann and Lin utilize a hierarchial data structure based on convex surface decomposition of the models. In [35] Larsen et. al. present a new distance computation method based on hierarchies of rectangular swept volumes. In [36] a framework for fast and accurate collision detection based on hierarchies of Oriented Bounding Boxes (OBBs) is presented. In [37] Ehmann and Lin extend the idea of Voronoi-based feature tracking presented in [32] to a multi-level-of-detail representation, in order to adapt to the variation in levels of coherence and speed up the computation. In [38] an incremental algorithm is presented that combines a hierarchical representation with a frame-to-frame computation for fast collision-detection in dynamic environments. In [39] discrete Voronoi regions are utilized to partition the Voronoi regions by voxels around a non-convex polyhedron. Each voxel is given the list of triangles which have possibility to be the closest to the points in the voxel, so that the closest triangles can be efficiently searched from this list on the voxel when a triangle on the other object intersects that voxel. Our approach is similar to this, except that in our framework the Voronoi regions are subdivided using octrees instead of voxels and all types of features are assigned individually to each node via Voronoi region construction.

4.3. Proximity Octrees

This chapter presents a method for octree-based tracking of features of convex objects (represented by triangular meshes) to a point of interest. In order to be able to quickly locate the feature of a convex object closest to a moving point, a buffer zone is created around the particular object to capture that point earlier before it touches the boundary of the object and track the closest features earlier before any interaction with the object-boundary occurs. This buffer zone is then structured by subdividing it with a special type of over-sized octree called the *proximity octree* whose boundary and outer nodes are assigned features of the convex object that are closest to its corresponding nodes.

Once the point of interest is located in a octree-node, a list of all features that

are guaranteed to be closer to that point than any other feature of the object can be obtained. This provides effective culling of many distant features at interactive rates.

4.3.1. Generation

A proximity octree differs from a standard octree in 2 respects:

a The cube at the root-node of the proximity octree is over-sized forming a buffer zone around the object it represents as opposed to the root-cube of a standard octree which tightly bounds the object it represents.

b In the standard octree some of the leaf-nodes (boundary nodes) are labeled with the geometric features they contain. In the proximity octree, in addition to such nodes there exist a set of nodes (outer nodes) which are labeled with the geometric features that they are closest to.

Thus for the proximity octree space subdivision is carried out in two steps:

1. subdivision based on the contained features of the object
2. subdivision based on the closest features (outer nodes only)

In a proximity octree, the number of outer nodes are comparatively much larger compared to a standard octree due to the second type of subdivision. When performing the subdivision based on the closest features, each outer node is labeled with every geometric feature, whose Voronoi region intersects the cube at that node. If the number of labels of a certain type assigned to a node exceed the minimum threshold value, then that node is subdivided recursively.

Since the Voronoi region of a feature is by definition the set of points closer to that feature than to any other feature, in order to find out which features the nodes of this proximity octree are closest to, we check intersection of Voronoi regions of features of the polyhedron with the nodes of the proximity octree. This procedure is shown in figure 4.1 for the case of a quadtree in 2D. In this figure the nodes closer to a feature

are marked with that features name while those that are closest to two features are marked with both features' names. Since this procedure is done in a pre-processing stage, once the proximity octree is constructed, it can be utilized over and over again to answer proximity queries efficiently. The Voronoi region and the outer nodes of the proximity octree representing a spherical object are shown in Figure 4.2.

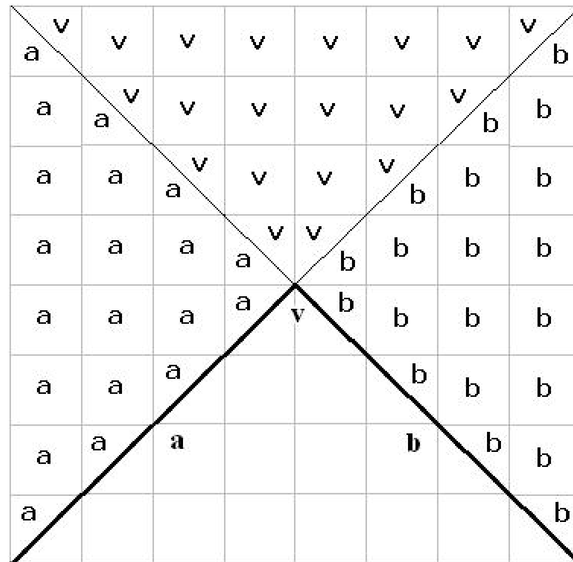

Figure 4.1. Closest features assigned to the quadtree-nodes

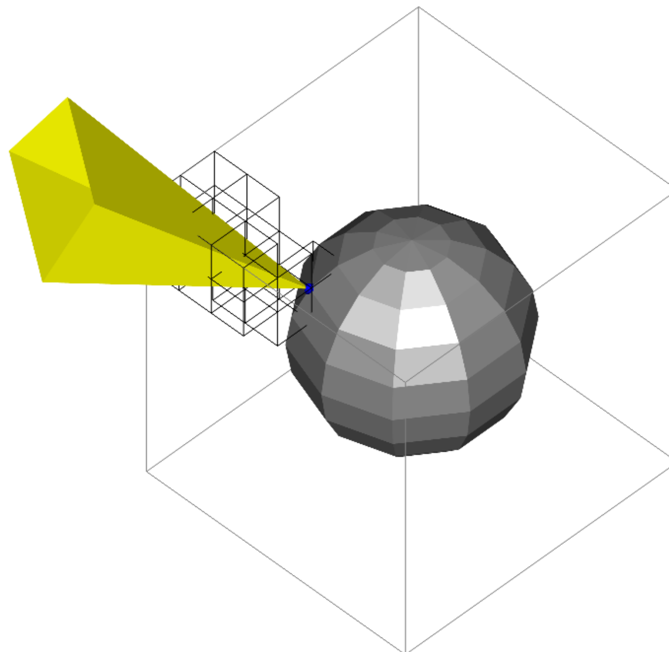

Figure 4.2. The proximity octree and a Voronoi region for a spherical object

4.3.2. Voronoi Regions

Voronoi Region of a feature (vertex, edge or face) of an object P is defined as the set of all points closer to that feature than to any other feature. This can be stated as

$$\mathcal{V}_{\mathcal{F}_i} = \{x \mid d(x, \mathcal{F}_i) \leq d(x, \mathcal{F}_j), \quad x \in R^3\} \tag{4.1}$$

for $\mathcal{F}_i, \mathcal{F}_j \in \mathcal{F}$, where $\mathcal{F}$ is the set of all features of the polyhedron P and $d(x, \mathcal{F}_i)$ gives the Euclidean distance of the point x to the feature $\mathcal{F}_i$. Since we assume all interactions take place with the outer boundary of the objects, our discussion is restricted to the exterior Voronoi regions. A convex object has three different types of features, namely vertices, edges and faces. Because we consider only triangular meshes, faces come as triangles. The geometric construction of each individual type of feature is as following:

Triangles: Voronoi region of a triangle is the set of all points, whose perpendicular projections (onto the supporting plane of that triangle) fall inside the boundaries of that triangle. Such points that lie inside the volume are obtained by extruding the triangle along its outward normal (Fig. 4.3). Thus the Voronoi region of a triangle is nothing but a prism extending to infinity along that triangle's normal, with the base being the triangle itself. Formally this is the set

$$\mathcal{V}_{T_i} = \bigcap_{j=1}^{4} H_j^i \tag{4.2}$$

where H_j^i are the closed half-spaces given by

$$H_j^i = \{x \mid (x - v_{i_j}) \cdot m_j \leq 0\} \qquad j = 1 \ldots 3 \tag{4.3}$$

$$H_4^i = \{x \mid (x - v_{i_j}) \cdot n_i \geq 0\} \tag{4.4}$$

where n_i is the normal and v_{i_j} is the jth vertex of the ith triangle T_i and m_j is the

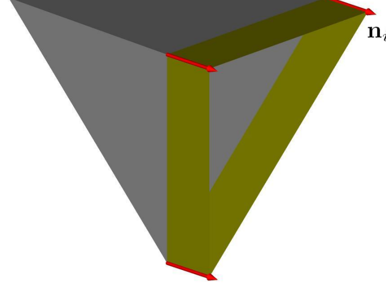

Figure 4.3. Voronoi region of a triangle of the pyramid

normal of the jth side-ways bounding plane of the Voronoi region given by

$$m_j = u_{i_j} \times n_i \tag{4.5}$$

with u_{i_j} being the direction vector of the jth edge of the triangle T_i and $\times$ denotes the vector-product operation. Note that each plane enclosing the Voronoi region is spanned by the normal of the triangle and the respective edge.

Edges: an edge is essentially the intersection of two triangles, therefore the Voronoi region of an edge is a wedge that is bounded by the planes bounding the Voronoi regions of these two triangles along that edge. At the vertices of the edge, the Voronoi region is bounded by two parallel planes spanned by the normal vectors $\mathbf{n}_{i_1}$ and $\mathbf{n}_{i_2}$ of these two triangles (Fig. 4.4). Formally, the Voronoi region $\mathcal{V}_{E_i}$ of the edge E_i is defined as

$$\mathcal{V}_{E_i} = \bigcap_{j=1}^{4} H_j^i \tag{4.6}$$

where H_j^i are the closed half-spaces given by

$$H_j^i = \{x \mid (x - v_{i_j}) \cdot m_j \leq 0\} \qquad j = 1 \dots 4 \tag{4.7}$$

where m_j is the outward normal of the jth bounding plane of the Voronoi region. These

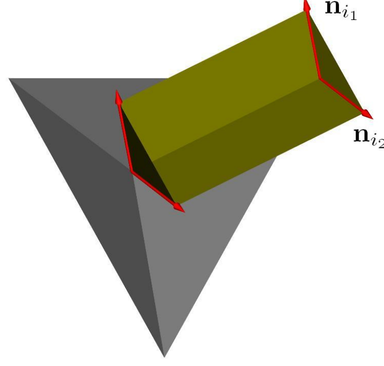

Figure 4.4. Voronoi region of an edge of the pyramid

normals are given by

$$m_1 = u_i \times n_{i_1}, \qquad m_2 = u_i \times n_{i_2} \tag{4.8}$$

$$m_3 = n_{i_1} \times n_{i_2}, \qquad m_4 = n_{i_2} \times n_{i_1} \tag{4.9}$$

where u_i is the direction vector of the ith edge and n_{i_1} and n_{i_2} are the normals of the triangles incident to that edge.

Vertices: since a vertex is a geometrical feature where the edges of the object meet, the Voronoi region of a vertex is bounded by the planes that bound the Voronoi region of each incident edge at that vertex. Thus the Voronoi region of a vertex is an unbounded pyramid whose apex is located at that vertex (Fig. 4.5) and is formally given by

$$\mathcal{V}_{V_i} = \bigcap_{j=1}^{|T(v_i)|} H_j^i \tag{4.10}$$

where $T(v_i)$ is the set of triangles incident to the ith vertex V_i and H_j^i are the closed half-spaces given by

$$H_j^i = \{x \mid (x - v_i) \cdot u_{i_j} \leq 0\} \qquad j = 1 \ldots |T(v_i)| \tag{4.11}$$

where u_{i_j} is the direction vector of the jth edge incident to the ith vertex. Note that

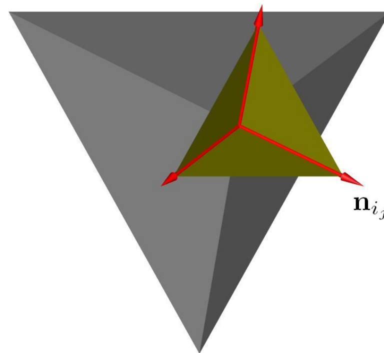

Figure 4.5. Voronoi region of an vertex of the pyramid

u_j is the vector obtained by taking the cross-product of the normals of the triangles incident to the jth edge, because the supporting line of the edge is the intersection of the supporting planes of these two incident triangles. Examples of Voronoi regions of all features for some common objects are shown in Fig. 4.6.

4.3.3. Intersection of Voronoi Regions and Cubes

Intersection detection of Voronoi Regions with the cubes of the octree can be done in a similar manner as was done in the case of intersection detection of triangles and cubes in chapter 3. All that has to be one is to construct the Minkowski sum of the Voronoi region and the cells such that all Voronoi regions are *grown* by the amount of the cells while the cells are being shrunk to their centroids. Once the desired Minkowski sums are calculated, the intersection of a Voronoi region with that cube can be detected by testing if the centroid of the cell is inside the *grown* Voronoi region, i.e. the Minkowski sum of the Voronoi region and the cube. Since the Voronoi regions of geometric features of convex polyhedra are unbounded geometric entities, geometrical computation of the Minkowski sums is achieved in two steps:

1. We compute the Minkowski sum as if the Voronoi region is a bounded polyhedron,
2. Then we remove all faces of this *intermediate* Minkowski sum that *obstruct* it along the extremal directions of the original Voronoi region.

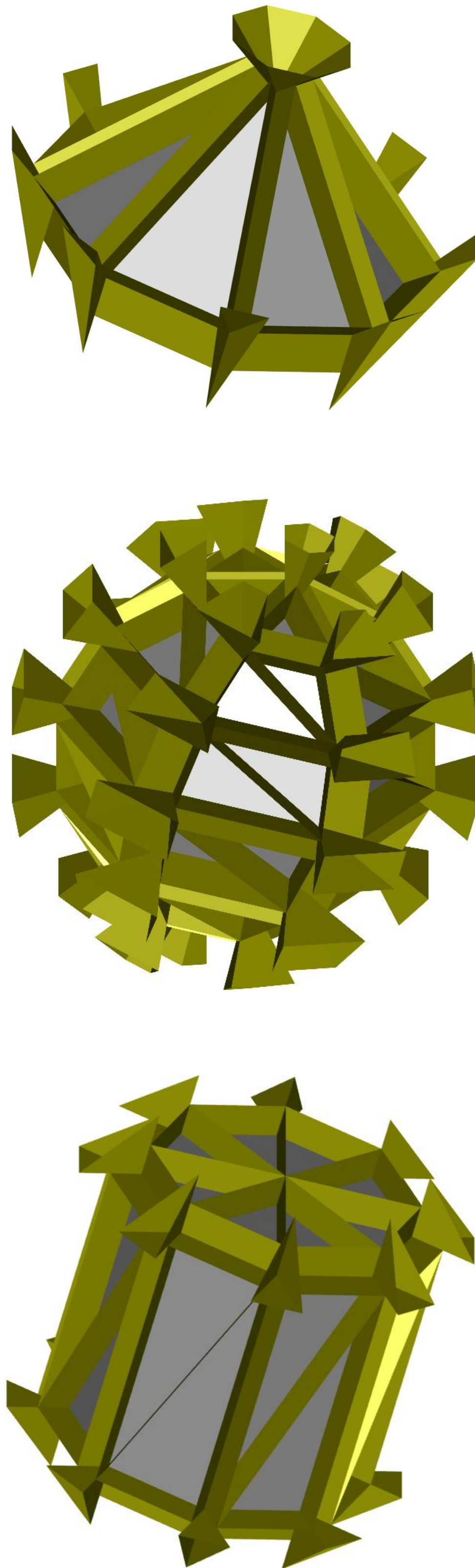

Figure 4.6. Voronoi regions of some convex 3D objects

In order to make the Voronoi region of finite dimensions, we assign to its side-faces arbitrary dimensions by extending the extremal directions by an arbitrary amount, say unit-length. Then we place copies of the cube at each of the vertices of the corresponding Voronoi region, as shown in the Fig. 4.8 for each feature-type.

The same process is shown in Figure 4.7 for the Voronoi region of a vertex in 2 dimensions. Figure 4.7(a) represents the unbounded Voronoi region, which is made finite dimensional by cutting it with an arbitrary line (plane in 3D). In Figure 4.7(b), the copies of the cube are placed at all of the corners and the Minkowski sum of this finite-dimensional Voronoi region and the cube is obtained by the geometrical method. In Figure 4.7(c), the edges (facets in 3D) obstructing the Minkowski sum to extend to infinity along the extreme directions are marked with their normals drawn in red. These are the normals that form a positive dot-product with *any* one of these extreme directions. Finally, in Figure 4.7(d), these edges are removed. In both 2- and 3-dimensions, the set of facets that need to be removed from the intermediate Minkowski sum is given by

$$\{ \, F_i \mid n_{F_i} \cdot e_j > 0 \, , \quad \forall \, F_i \in M_{int}, \;\; \forall \, e_j \in E_{\mathcal{V}} \, \} \tag{4.12}$$

where F_i is the ith facet of the intermediate Minkowski sum M_{int}, n_{F_i} is the normal of F_i and e_j is the jth extreme direction belonging to the set of extreme directions $E_{\mathcal{V}}$ of the Voronoi region $\mathcal{V}$. In these figures the dimensions chosen arbitrarily (to make the unbounded Voronoi region a finite dimensional polyhedron) will have no impact on the final result, as long as these arbitrary dimensions do not cause numerical problems due to finite precision arithmetic. Three different types of Minkowski sums are shown in Figure 4.9. In this figure, extremal directions are shown as yellow arrows, whereas the normals of the obstructing faces are shown as red arrows. The red-normals belong to faces that form positive dot-products with the extremal directions of the respective Voronoi region. In order to obtain the unbounded Minkowski sum, these faces are removed. Since the Voronoi region is a simple convex object, to calculate its intersection with the nodes of an octree the same principles we have outlines earlier are exploited.

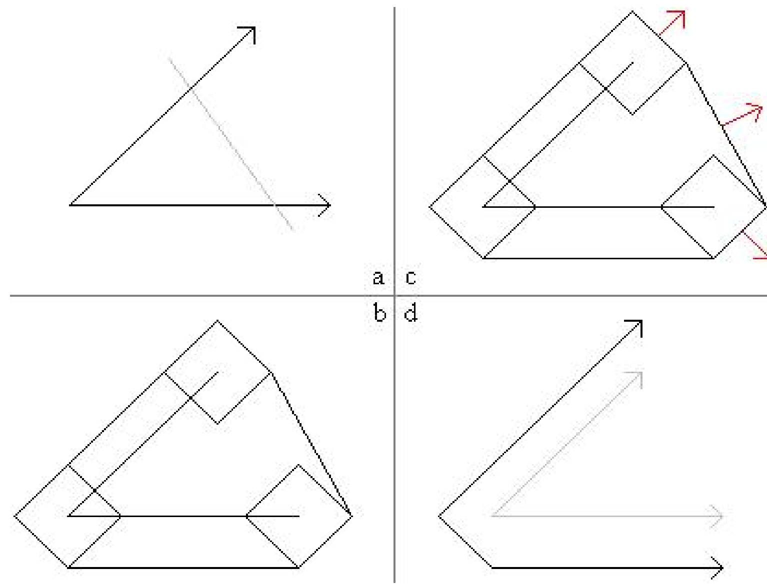

Figure 4.7. Unbounded Minkowski sum in 2D

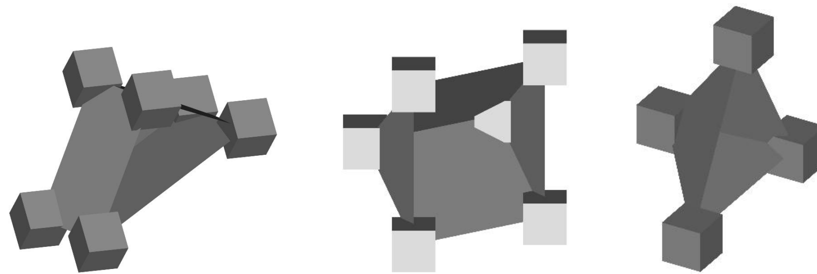

Figure 4.8. Copies of the cube placed at all vertices of each Voronoi region

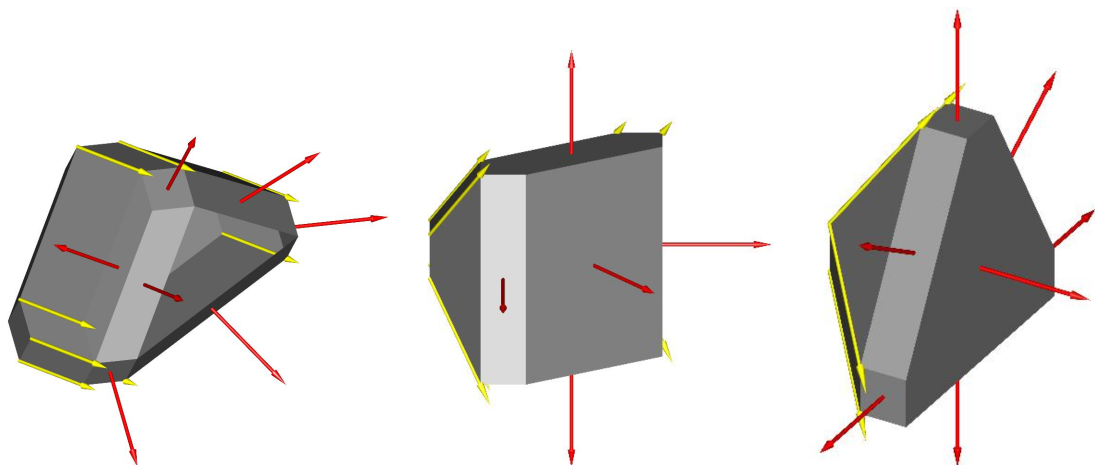

Figure 4.9. Intermediate Minkowski sums of Voronoi regions

5. APPLICATIONS AND PERFORMANCE EVALUATIONS

In this chapter we will present sample applications in OpenGL environment to evaluate the performance of methods developed within our framework.

5.1. Haptic Interaction

Haptic interaction with a virtual environment enables a user to feel forces that originate from interaction with that environment via an electromechanical device called the *haptic device*. Haptic interaction is used to augment the sense of vision with the sense of touch so that the user is able to interact with the virtual environment in a more realistic manner.

The forces reflected to the user by the haptic device are computed by a process called *haptic rendering*. Haptic rendering consists of two major steps that are performed repetitively at high update rates:

1. Determination of contact or proximity between an object and a point controlled by the haptic device, also called the *haptic cursor*.
2. Calculation of forces based on a distance measure, which is either penetration depth into the contacted surface or the distance to the closest feature of the object.

There are numerous methods proposed in the literature how these interaction forces should be calculated. The interaction forces do not depend only on the distance measure used, but also on the material properties like static and dynamic surface friction coefficients, damping factors, surface stiffness as well as texture. Since in our framework our primary concern is efficient proximity queries, for haptic rendering the interested reader should look at references such as [40, 41, 42, 43, 44, 45, 46, 47]

Haptic rendering must be performed at update rates as high as 1000 Hz to guar-

antee a continuity in the kinesthetic perception of the human-sense of touch. At frequencies below this threshold value, there is a high chance that discontinuities in forces reflected via the haptic device are felt. But this high update rate is in contrast with the relatively much lower update rates needed for graphic rendering. Therefore a virtual reality application involving haptic interaction usually consists of at least 2 different threads:

- Graphic Rendering thread (running at approximately 50 Hz)
- Haptic Rendering thread (running at approximately 1000 Hz)

For our haptic interaction application we use the *Phantom Omni* haptic device from SensAble Technologies (Figure 5.1). This haptic device is capable of reading 3 Cartesian coordinates and 3 orientation angles at its tip (6-DOF input) and can provide the user with force-feedback without torque-feedback (3-DOF force-output). This device comes with a Software Development Kit (SDK) called the Open Haptics Toolkit (OHT) to enable application developers implement their own haptic rendering methods.

Figure 5.1. Phantom Omni haptic device [48]

OHT consists of two Application Programming Interfaces (APIs), a low-level library called HDAPI and a high-level library called HLAPI. HLAPI provides an easy way to interface the haptic device using OpenGL instructions to enable haptic rendering

of objects consisting of polygonal primitives, which has been our choice for implementation to avoid the more detailed issues related to force calculations and thread synchronization. When linked to an application, HLAPI registers two additional threads at the background:

- *servo-thread* running at 1000 Hz for sending force-updates to the haptic device
- *collision-thread* running at 100 Hz for detecting collisions between the haptic cursor and the geometric model defined via OpenGL instructions.

With the above threads running at the back-ground, the only thing that is left to the *main thread* is to send the geometry information via OpenGL instructions. Since the priority of the main thread is lower than those registered by HLAPI, its update rate is guaranteed not to be higher than the collision-thread so that any geometry sent from the main thread is captured by the collision-thread. The collision information is then sent to the servo-thread to ensure stable haptic rendering.

5.1.1. Use of Proximity Octree for Haptic Interaction

The primary need of the proximity octree has been the need to estimate the set of primitives that have a high probability to collide with the haptic cursor very quickly. This need in turn is due to the high complexity of the objects consisting of many primitives. In various tests with the haptic device we have observed that sending all of the primitives to the haptic rendering pipeline at every frame is costly and results in unstable haptic feedback due to the heavy overload on the haptic rendering pipeline.

Since contact regions are guaranteed to be in close proximity of the haptic cursor, finding out and sending only the closest features of the object to be rendered reduces the load on the haptic rendering pipeline and results in stable haptic rendering. Thus the primary use of proximity octrees in haptic rendering is to find a set of features of a convex object that are guaranteed to be closer to the haptic cursor than any other feature of that object. Once such a set is found, it is much easier to find out the individual feature of the object closest to the haptic cursor. All triangles in the

neighborhood of that feature (upto the desired *adjacency depth*) are then sent down to the haptic rendering pipeline. To guarantee that fall-through of the haptic cursor due to missing surface patches caused by asynchronous and/or delayed running of threads, the size of the neighborhood can be adjusted.

Figure 5.2 shows a snapshot of our haptic interaction application, where the feature closest to the haptic cursor is drawn in red and all triangles adjacent to that feature are in blue. For the position of the haptic cursor at the time of taking the snapshot, the closest feature is a triangle and the leaf node containing the cursor is shown in green.

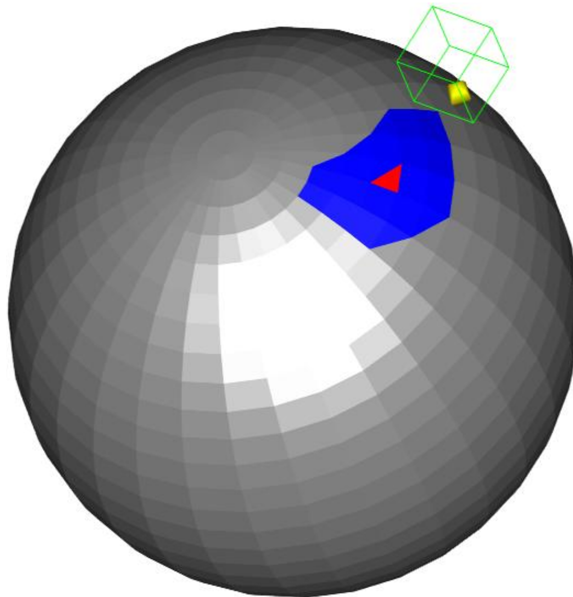

Figure 5.2. Snapshot of the haptic interaction application

5.1.2. Culling Performance of the Proximity Octree

To demonstrate the culling performance of the proximity octree, a simple evaluation is performed on a sample object obtained by subdividing a sphere into 50 stacks and 50 slices, which results in 2452 vertices, 4850 non-degenerate edges and 4900 triangles. Effective culling is observed starting from the 3rd depth. Figure 5.3 shows the distribution of the closest feature assignments to outer nodes of the proximity octree. Note how the histogram shifts to left as the depth of the octree is increased. This confirms the expected result that as the depth of the proximity octree is increased,

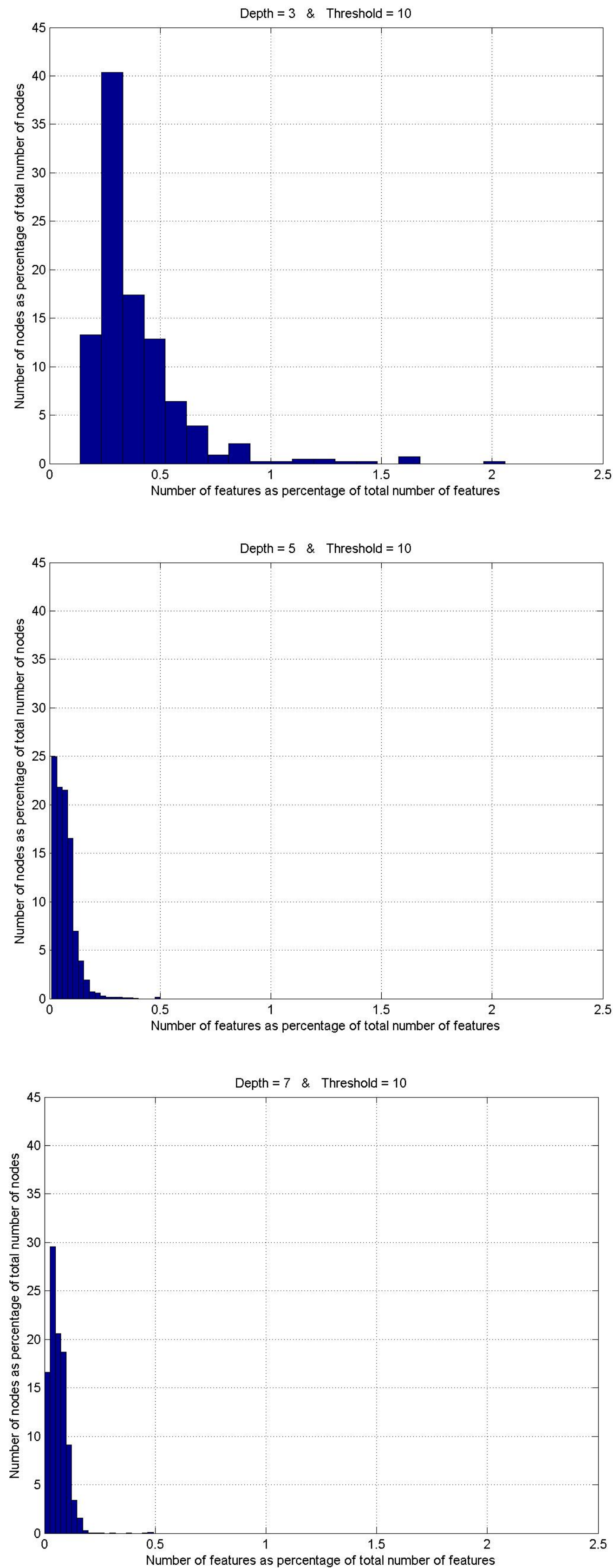

Figure 5.3. Distribution of features for various depths

an *increasing* percentage of nodes are assigned with *smaller* number of features. This is due to reduced size of the outer nodes, so that each node is intersected by a small number of Voronoi regions as the depth increases.

5.2. Object-Object Intersections

Intersection and collision detection plays an important role in many virtual reality applications. Being able to detect collision among objects or calculate intersection among them at interactive rates is an important property of many applications. In this section a simple demonstration of using octrees in intersection calculations is given.

Finding the intersection of two general 3D polygonal objects is a computationally challenging problem, especially if complex objects consisting of large number of geometric primitives are involved. A crucial problem in intersection detection and calculation is to be able to locate the sets of primitives that have a high potential to intersect. Since octrees, due to their inherent coherence, provide a means to group such primitives inside a particular structured region in space, regions of potential contact or intersection can be quickly located by checking intersections among the hierarchies of cubes rather than individual primitives.

To perform intersection queries between the cubes of two different octrees each having different spatial dimensions and orientations, the same principles described earlier are used. Since all cubes of each octree are translated and scaled copies of a prototype cube for each octree, once the Minkowski sum of two prototype cubes each one from each octree are calculated, the Minkowski sums between prototype cubes at different depths of both octrees can be calculated by taking advantage of the topological invariance property.

Consider two octrees each having a depth of 10 having different dimensions and orientations in 3D space. In order to find out which cubes of both octrees intersect in an efficient manner, we have to calculate the Minkowski sums of all pairs of cubes coming from each octree, leading to $10 \times 10 = 100$ Minkowski sum calculations. But

since calculating so many Minkowski sums requires a prohibitively large number of computations regarding the interactive rate requirements of many virtual reality applications, taking advantage of topological invariance is an important step for performing these intersection queries much more efficiently. This means that once we calculate the Minkowski sum of two cubes at any depth each from one octree, the remaining 99 Minkowski sums can be calculated much more rapidly, which lead us to achieve interactive intersection detection rates.

Once all of the Minkowski sums are calculated, the next problem to be solved is how to find the set of intersecting nodes from both octrees. Since there are in general a huge number of nodes in an octree, the number of intersection tests to be performed can be prohibitively large if a poor recursive strategy is employed. An effective solution to this problem is to eliminate large number of nodes at stages as early as possible. To realize such an elimination, the depth values of both octrees must be increased one by one, i.e. the order of the intersection queries to be performed should be as follows

- depth 0 (root) of the octree-1 and depth 0 (root) of the octree-2
- depth 1 of the octree-1 and depth 0 of the octree-2
- depth 1 of the octree-1 and depth 1 of the octree-2
- depth 1 of the octree-1 and depth 2 of the octree-2
- depth 2 of the octree-1 and depth 2 of the octree-2
- ...

This type of recursion can be thought of as *double recursion* in the sense that nodes intersecting each other at a certain depth value of one of the octrees, call the intersection detection routine recursively for their children with the same node from the other octree and vice-and-versa. Since there is no point in checking the intersection for a pair of nodes, one of which does not contain any primitives, a check is made at the very first entry to the recursive function to exit from if this is the case. Performing intersection detection queries with such a recursive scheme results in significant performance gains. The algorithm we employ for node-node intersection detection can be summarized as follows:

find_intersecting_nodes(*node1*, *node2*, *node_list*)

1 IF(*node1* or *node2* does NOT contain any primitives)

1.1 return;

END IF (1)

2 IF(*node1* and *node2* do NOT intersect)

2.1 return;

END IF (2)

3 IF(*node1* has any children)

3.1 FOR($i = 1 \ldots 8$)

3.1.1 find_intersecting_nodes(*node2*, i^{th} child of *node1*, *node_list*)

END FOR (3.1)

END IF (3)

4 IF(*node1* and *node2* are leaf-nodes)

4.1 add *node1* and *node2* to *node_list* ;

END IF (4)

The Figure 5.4 shows a snapshot of a simple demo developed to show the effectiveness of our approach in finding the list of all potentially intersecting triangles of two spherical objects. Each sphere has 1740 triangles. In this snapshot, the triangles rendered in yellow color represent those triangles that are assigned to the nodes of one octree that are intersecting those of the other octree. The intersecting nodes are rendered as red and green boxes where each color shows, which octree the node belongs to.

As can be seen in Figure 5.4, it is sufficient to evaluate the intersection of a small fraction of triangles of both objects, if the aim is to evaluate the intersection boundary of these objects. Note that, if the only aim is to *detect* intersection, then a much less number of intersection tests will be sufficient to detect the intersection, since in such a case locating a single pair of intersecting triangles in a pair of intersecting nodes in the very early stages would be sufficient to cancel all the very large number of further intersection tests in our example.

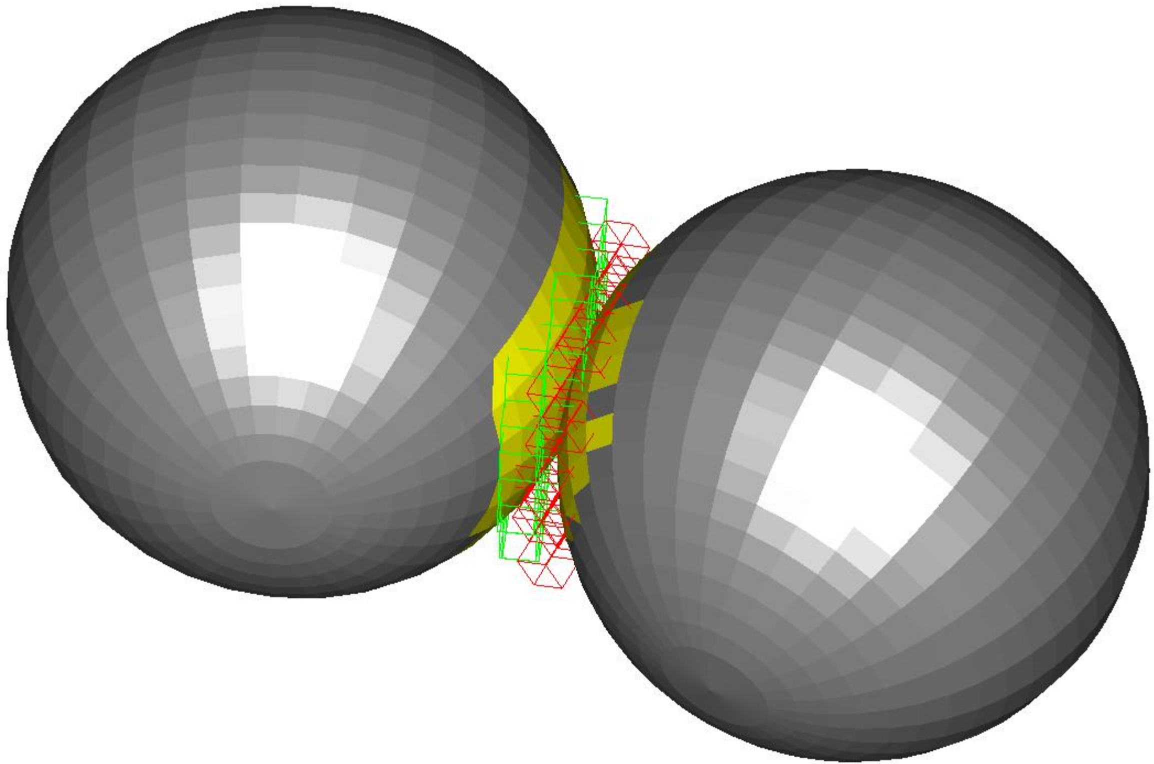

Figure 5.4. Intersection of 2 sphere-like objects

5.3. Ray Casting

Ray-casting is an important technique widely used in volume rendering applications as well as in visibility calculations involved in global illumination methods such as ray tracing and radiosity. Many methods and hierarchial representations have been proposed in the literature to speed-up the ray-casting process [12] with different complexities.

When developing our framework, we have realized that the methods we have developed, which take advantage of topological invariance of Minkowski sums when using octrees, are as well applicable to intersection testing between rays and cubes of an octree.

Another advantage of using Minkowski sums in ray casting is that, while performing the intersection test of a ray with the cube of the octree, the amount of penetration of the ray inside the cube can be calculated at the same time, which is a very important

advantage compared to other methods used for such intersection tests.

The *penetration depth* is an important quantity in ray-casting for volume rendering since it gives a measure of how much an intersected cube (voxel) should contribute to the final intensity produced at a specific location of the screen. For voxels, whose penetration depth is small (where the ray is passing very close to one of the corners of the voxel), the contribution of that voxel to the final intensity should be small as well.

Figures 5.5 and 5.6 show snapshots of a simple demonstration of ray casting with calculated penetration depth. A ray is cast into the octree and all voxels that are intersected by this ray are highlighted with a color ranging between yellow and black. The intensity of the intersected voxels show the contribution of that voxel to the final intensity, i.e. brighter voxels are those where the penetration is high, while darker voxels represent those where the penetration is low.

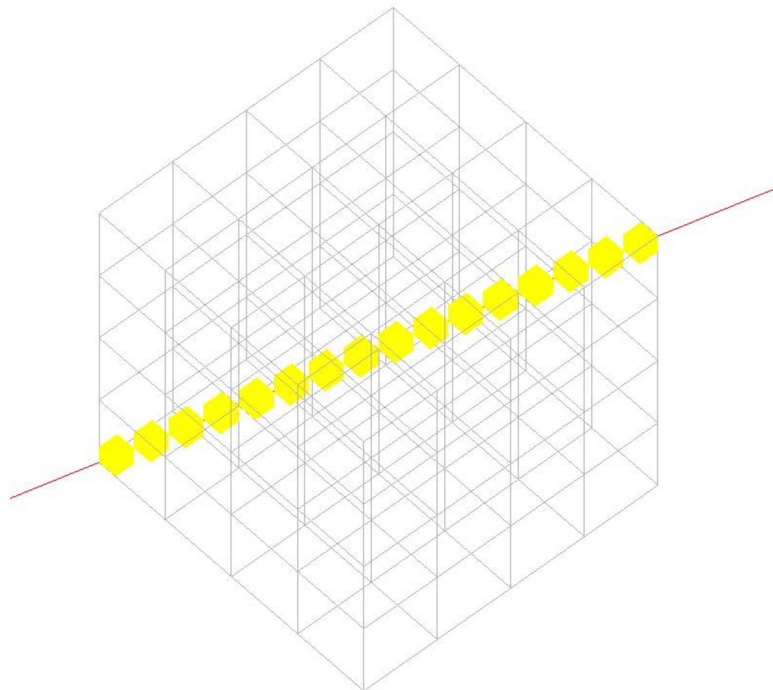

Figure 5.5. Ray intersecting voxels with full intensity contribution

Calculation of the penetration for an intersected voxel is based on the following

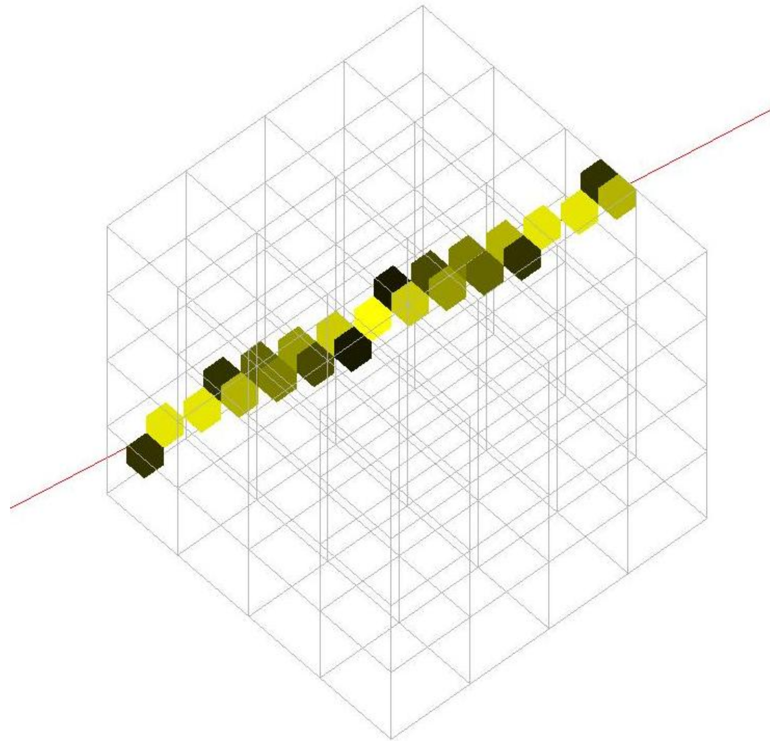

Figure 5.6. Ray intersecting voxels with various intensity contributions

idea: For each ray having a certain direction, the Minkowski sum of the voxel and that particular ray produce a prism extending to infinity along the ray-direction. The minimum distance from the ray to the sides of this prism gives the maximum penetration depth possible for that ray. Once this value has been found, the contribution of the voxel can be found by dividing the actual penetration depth into the voxel in question by this maximum penetration depth for the given ray.

5.4. View Volume Culling

In graphics rendering, culling of a large portion of a scene that is guaranteed to be out of view provides an important optimization. Such an elimination of primitives outside the so-called view volume is called View Volume Culling, or also View Frustum Culling. Once a scene is structured using an octree, view volume culling can be done by performing intersection tests between the cubes of the octree and the frustum representing the view volume, which is a box for orthographic projection or a truncated pyramid for perspective projection. The intersection tests between the view volume

and cubes of the octree can be done efficiently by taking advantage of the spatial coherence inherent in the octree. Figure 5.7 and 5.8 illustrates a simple view frustum culling example. Only those primitives are rendered, which are contained in nodes that are intersected by the view frustum.

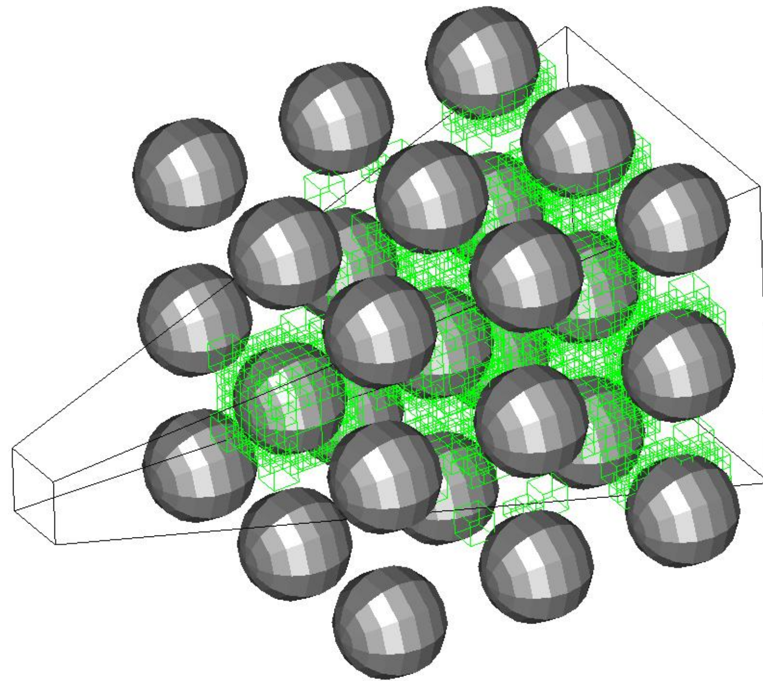

Figure 5.7. View volume culling example (isometric view)

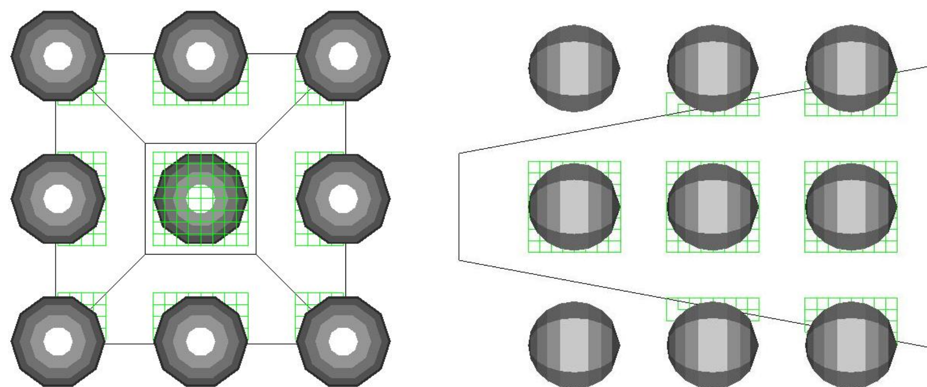

Figure 5.8. View volume culling example (axis views)

6. CONCLUSIONS

The methods we have developed for this thesis for octree generation and its utilization are observed to present a unified framework for efficient generation of octrees and evaluation of proximity queries. The initial results obtained for the sample octree-generation application show the potential of our approach employing Minkowski sums which takes advantage of both translational and topological invariance of the Minkowski sums of simple geometric entities and cubes of an octree.

The performance of the proximity octree promises to be an effective tool in culling a very large of portion of distant features of a convex object to a point of interest. It has been successfully used in our haptic rendering demo and the haptic interaction is observed to be stable although our processing thread has a much lower priority than the collision detection and servo threads, pointing out to the real-time performance of the proximity octree.

The simple concept of detecting the intersection of simple geometric primitives with the cubes of the octree using Minkowski sums is further applied to ray-casting, view volume culling and object-object intersections. Although no temporal measurements have been conducted, using our approach we achieved real-time performance, even without index-tracking to speed-up Minkowski sum calculations.

In the future we plan to speed-up the octree generation process by speeding up the calculation of the Minkowski sums. To achieve this we plan to exploit the special structure of the dual images of the simple geometric primitives under the Gaussian Map and implement these simple algorithms on Graphical Processing Units (GPUs). Also, we want to investigate the potential of the proximity octrees in distance computations between two non-convex objects. In addition to this, we want to extend the use of octrees to achieve faster evaluation of some important queries.

REFERENCES

1. Hubbard, P.M., “Interactive collision detection”, In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993

2. Quinlan, S., “Efficient distance computation between non-convex objects”, In *Proceedings of Internetional Conference on Robotics and Automation*, pp. 3324–3329, 1994.

3. Beckmann, N., H. Kriegel, R. Schneider and B. Seeger, “The r-tree: An efficient and robust access method points and rectangles”, *Proc. SIGMOD Conf. on Management of Data*, pp. 322–331, 1990.

4. Held, M., J.T. Klosowski and J.S.B. Mitchell, “Evaluation of collision detection methods for virtual reality fly-throughs”, In *Canadian Conference on Computational Geometry*, 1995.

5. Barequet, G., B. Chazelle, L. Guibas, J. Mitchell and A. Tal, “Boxtree: A hierarchial representation of surfaces in 3d”, In *Proc. of Eurographics'96*, 1996.

6. Gottschalk, S., M. Lin and D. Manocha, “Obb-tree: A hierarchial structure for rapid interference detection”, *Proc. of ACM Siggraph'96*, pp. 171–180, 1996.

7. Naylor, B., J. Amanatides and W. Thibault, “Merging bsp-trees yield polyhedral modelling results”, *Proc. of ACM Siggraph*, pp. 115–124, 1990.

8. Samet, H. *Spatial Data Structures: Quadtree, Octrees and Other Hierarchial Methods.* Addison Wesley, 1989.

9. Naylor, B., *A Tutorial On Binary Space Partitioning Trees*, 1993.

10. Greene, N., M. Kass and G. Miller, “Hierarchical Z-buffer visibility”, *Computer Graphics*, Vol. 27, pp. 231–240, 1993.

11. Zhang, H., D. Manocha, T. Hudson and K.E. Hoff III, “Visibility culling using hierarchical occlusion maps”, *Computer Graphics*, Vol. 31, pp. 77–88, 1997.

12. Watt, A. and M. Watt, *Advanced Animation and Rendering Techniques*, Addison-Wesley, 1992.

13. Endl, R. and M. Sommer, “Classification of Ray-Generators in Uniform Subdivisions and Octrees for Ray Tracing”, *Computer Graphics Forum*, Vol. 13, No. 1, pp. 3–19, 1994.

14. Kuzmin, Y.P., “Ray Traversal of Spatial Structures”, *Computer Graphics Forum*, Vol 13, No. 4, pp. 223–227, 1994.

15. Noborio, H., S. Fukuda, and S. Arimoto, “Fast interference check method using octree representation”, *Advanced Robotics*, Vol. 15, pp. 193–212, 1989.

16. Yu, Y., M. Wu and J. Zhou, “An octree algorithm for dynamic interference detection using space partitioning”, In *Proc. Of The 1996 ASME Design Engineering Technical Conference and Computers in Engineering Conference*, Irvine, CA, Paper Number 96, 1996.

17. Sung, R.C.W., J.R. Corney and D.E.R. Clark, ’‘Octree based recognition of assembly features”, In *Proceedings of the ASME 2000 Design Engineering Technical Conferences and Computers and Information in Engineering Conference, 10-13 September, ASME, Baltimore*, 2000.

18. Roy, U. and Y. Xu, “3-D object decomposition with extended octree model and its application in geometric simulation of NC machining”, *Robotics and Computer-Integrated Manufacturing*, Vol. 14, pp. 317–327, 1998.

19. Medellin, H. and et. al., “Algorithms for the physical rendering and assembly of octree models”, *Computer-Aided Design*, Vol. 38, pp. 69–85, 2006.

20. Ding, S., M.A. Mannan and A.N. Poo, “Oriented bounding box and octree based

global interference detection in 5-axis machining of free-form surfaces", *Computer-Aided Design*, Vol. 36, pp. 1281-1294, 2004.

21. Agarwal, P. K., E. Flato and D. Halperin, "Polygon decomposition for efficient construction of Minkowski sums", *Comput. Geom. Theory Appl.*, Vol. 21, pp. 39-61, 2002.

22. Guibas, L. J., L. Ramshaw and J. Stolfi, "A kinetic framework for computational geometry", *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 100-111, 1983.

23. Guibas, L.J. and R. Seidel, "Computing convolutions by reciprocal search", *Disc. Comp. Geom.*, Vol. 2, pp. 175-193, 1987.

24. Gritzmann, P. and B. Sturmfels, "Minkowski addition of polytopes: Computational complexity and applications to Grobner bases", *SIAM J. Disc. Math*, Vol. 6, No. 2, pp. 246-269, 1993.

25. Fukuda, K., "From the zonotope construction to the Minkowski addition of convex polytopes", *Journal of Symbolic Computation*, Vol. 38, No. 4, pp. 1261-1272, 2004.

26. Ghosh, P.K., "A unified computational framework for Minkowski operations", *Comp. Graph.*, Vol. 17, No. 4, pp. 357-378, 1993.

27. Bekker, H. and J.B.T.M. Roerdink, "An efficient algorithm to calculate the Minkowski sum of convex 3d polyhedra", *Proc. of the Int. Conf. on Comput. Sci.-Part I*, Springer-Verlag, pp. 619-628, 2001.

28. Fogel, E. and D. Halperin, "Exact and Efficient Construction of Minkowski Sums of Convex Polyhedra with Applications", *To appear in ALENEX 2006, Miami, Florida*, 2006.

29. O'Rourke, J., *Computational Geometry In C*, Cambridge University Press, 1994.

30. Eberly, D., *Dynamic Collision Detection using Oriented Bounding Boxes*,

http://www.geometrictools.com/Documentation/DynamicCollisionDetection.pdf

31. http://www.trekmeshes.ch/

32. Lin, M.C. and J.F. Canny, "Efficient algorithms for incremental distance computation", In *IEEE Conference on Robotics and Automation*, 1991

33. Lin, M.C., *Efficient Collision Detection for Robotics and Animation*, PhD Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993

34. Ehmann, S.A. and M.C. Lin, "Accurate and Fast Proximity Queries between Polyhedra Using Convex Surface Decomposition", *Eurographics 2001*, Vol. 20, No. 3, pp. 500–510, 2001.

35. Larsen, E., S. Gottschalk, M. Lin and D. Manocha, "Fast Distance Queries with Rectangular Swept Sphere Volumes", *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, Vol. 4, pp. 24–48, April 2000.

36. Gregory, A., M.C. Lin, S. Gottschalk and R. Taylor, "A Framework for Fast and Accurate Collision Detection for Haptic Interaction", In *Proc. IEEE Virtual Reality 1999*, pp. 38–45, 1999.

37. Ehmann, S. and M. Lin, *Accelerated proximity queries between convex polyhedra by multi-level Voronoi Marching*, Technical report, Computer Science Department, University of North Carolina at Chapel Hill, 2000.

38. Ponamgi, M., D. Manocha and M.C. Lin, "Incremental Algorithms for Collision Detection between Solid Models", *Proc. ACM/ SIGGRAPH Symp. Solid Modeling*, pp. 293–304, 1995.

39. Kawachi, K. and H. Suzuki, "Distance Computation between Non-convex Polyhedra at Short Range Based on Discrete Voronoi Regions" *Proceedings in Geometric Modeling and Processing 2000*, pp. 123–128, 2000.

40. Baraff, D., "Fast Contact Force Computation for Nonpenetrating Rigid Bodies", *Computer Graphics, Ann. Conf. Series*, Vol. 28, pp. 23–34, 1994.

41. Salisbury, K., D. Brock, T. Massie, N. Swarup and C. Zilles, "Haptic Rendering: Programming Touch Interaction with Virtual Objects", *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics (I3D)*, pp. 123–130, Apr. 1995.

42. Ruspini, D., K. Kolarov and O. Khatib, "The Haptic Display of Complex Graphical Environments", *Proc. Computer Graphics and Interactive Techniques (SIGGRAPH 1997)*, pp. 345–352, Aug. 1997.

43. Salisbury, K. and C. Tarr, "Haptic Rendering of Surfaces Defined by Implicit Functions", *ASME Dynamic Systems and Control Division*, Vol. 61, pp. 61–67, 1997.

44. Ho, C., C. Basdogan and M. Srinivasan, "An Efficient Haptic Rendering Technique for Displaying 3D Polyhedral Objects and their Surface Details In Virtual Environments", *PRESENCE: Teleoperators and Virtual Environments*, Vol. 8, No. 5, pp. 477–491, 1999.

45. Thompson II, T.V. and E. Cohen, "Direct Haptic Rendering of Complex Trimmed Nurbs Models", In *8th Annual Symp. Haptic Interfaces for Virtual Environment and Teleoperator Systems*, 1999.

46. Otaduy, M.A. and M.C. Lin, "Sensation Preserving Simplification for Haptic Rendering", *ACM Trans. Graphics (TOG): SIGGRAPH 2003*, Vol. 22, No. 3, pp. 543–553, 2003.

47. Salisbury, K., F. Conti and F. Barbagli, "Haptic Rendering: Introductory Concepts", *IEEE Computer Graphics and Applications*, Vol. 24, No. 2, pp. 24–32, 2004.

48. http://www.sensable.com/