Jan Klein

# Efficient Collision Detection for Point and Polygon Based Models

# Efficient Collision Detection for Point and Polygon Based Models

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften von

Jan Klein

Heinz Nixdorf Institut und
Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Universität Paderborn

Mai 2005

# Acknowledgements

First and foremost, I wish to thank my advisor, Prof. Dr. Friedhelm Meyer auf der Heide, for his great support. His constructive comments, ideas and optimism have always inspired me during the past years.
I feel fortunate to have met Prof. Dr. Gabriel Zachmann at VRST'02. Since then, we have had a fruitful and warmhearted cooperation, resulting in several publications which constitute the basis for this work. Moreover, I would like to thank him for many exiting and fascinating after-conference trips, e.g., climbing in the Western Alps or hiking in Yosemite.
I am grateful to Prof. Dr. Reinhard Klein and to Prof. Dr. Odej Kao for agreeing to co-referee this thesis and for their encouraging words. I am also very thankful to my other thesis committee members, Prof. Dr. Gerd Szwillus and Dr. Michael Thies.
For useful comments, ideas and for answering several questions, my thanks go to Dr. Michael Wand as well as to all my colleagues. For the excellent introduction into computer graphics during my studies, my thanks go also to Prof. Dr. Gitta Domik.
I thank my parents for always being there for me and for their very kind support over all the years.
Last but not least, I would like to thank Birgitta for all her love and care throughout my studies. This work would not have been finished without her invaluable help.

Paderborn, May 2005 Jan Klein

# Contents

# 1 Introduction

Efficient collision detection of 3D objects is needed in many highly interactive applications such as medical education, physically-based simulations, virtual prototyping, robotics, and 3D games.
In physically-based simulations, where the objects should obey the laws of physics, collision detection is used to ensure that no bodies penetrate each other. In virtual prototyping, clearances can be determined by the collision detection process, and in computer games, collision detection ensures that colliding objects bounce and slide on contact instead of passing pointlessly through each other.
Interactive 3D computer graphics requires efficient collision detection algorithms combined with object representations that provide fast answers to collision queries. This is a prerequisite in order to simulate plausible physical behavior, and in order to allow a user to interact with the virtual environment in real-time. In this thesis we are concerned with the collision detection of point-based and polygon-based models. While polygonal objects are still the standard representation, point clouds have become a popular shape representation over the past few years. This is due to two factors: first, 3D scanning devices have become affordable and thus widely available [RHHL02]; second, points are an attractive primitive for efficient rendering of complex geometry for several rea-

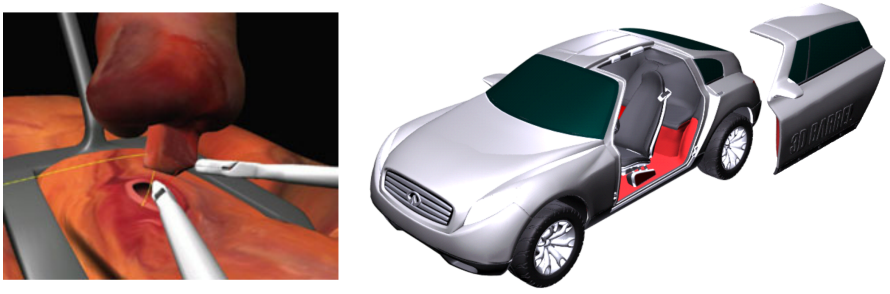


**Figure 1.1:** Collision detection is needed in many highly interactive applications, e.g., in order to model interactions between objects in a surgical simulation system or for virtual prototyping in the automotive industry (models courtesy of SimSurgery® and www.3dbarrel.com).

sons [PvBZG00, RL00, ZPvBG02, BWG03]. However, only very little work has been done on the interaction of such object representations and the inherent collision detection process.
At first glance, determining whether two objects (independent of their representation) intersect, seems quite simple. However, the demands on current applications and the constraints of modern computers complicate the situation dramatically.
First, hard, real-time requirements lead to very small time budgets for each collision query. To aggravate the situation, a scheduler interrupts the collision detection process in time-critical environments. Existing approaches however, give absolutely no hint as to the confidence in the result in such cases.
Second, there are several factors that contribute to imprecision of the collision detection because of errors in a model or the simulation. For example, there are approximations in the abstraction, inaccuracy in the data, or numerical errors in the computation. As a consequence, unnecessary errors, e.g. introduced by conversions, should be avoided if any possible.
Third, users really do not like long preprocessing times nor conversions of object representations into other ones where additional files are produced. However, no efficient algorithm has been proposed to detect collisions between point cloud models directly without any conversion that obviously introduces errors in the surface.
Last but not least, memory usage is an important factor and should always be taken into account. Small design errors during the specification can result, e.g., in memory-expensive bounding volume hierarchies[1], which — if constructed carefully — have proven to be a very efficient data structure for collision detection, even for (reduced) deformable models [JP04].
Although researchers have been concerned with collision detection for more than a quarter-century, there are several unsolved problems. From the design issues mentioned above, we would like to emphasize two problems that are very important for the real-time collision detection in interactive environments.

1. There are several areas of collision detection, e.g., medical training or games, where absolute accuracy is not needed, but where efficiency and speed is very important. However, no method has supported application-driven "levels-of-detail" of collision detection where the application can

[1] The idea of bounding volume hierarchies is to partition the set of object primitives, e.g. polygons or points, recursively until some leaf criterion is met. In most cases, each leaf contains a single primitive, but the partitioning can also be stopped when a node contains less than a fixed number of primitives. Each node in the hierarchy stores a subset of the primitives and a bounding volume that encloses this subset.

specify an allowable error rate beforehand. Moreover, theoretical foundations concerning the error incurred by an incomplete traversal have not been given by any approach. Thus, we aim for a continuous and controllable balancing between the running time and the accuracy of collision detection.

2. Prior to our work, no efficient algorithm has been proposed to detect collisions between the implicit surfaces defined by point clouds. Our goal is to never explicitly reconstruct the surface in order to avoid the additional storage overhead and additional error that would be introduced by polygon reconstruction.

In this thesis, we present algorithms and data structures for those two problems. Theoretical and experimental evidence is given to show their efficiency. Except for some few algorithms and data structures we explicitly mention, all approaches are implemented and evaluated prototypically. The main contributions can be summarized as follows.

1. We introduce a novel framework for collision detection using an average-case approach, thus extending the set of techniques for plausible simulation. To our knowledge, this is the first time that the *quality* of collision detection can be decreased in a controlled way while increasing the speed, such that a numeric *measure* of the quality of the results is obtained (which can then be related to the perceived quality). Our method utilizes bounding volume hierarchies. The main idea is to consider sets of polygons at inner nodes of such a hierarchy, and then, during traversal, we derive an estimation of the probability that there *exists* a pair of intersecting polygons based only on a small number of parameters describing the *density* of the polygon sets. This probability can be used to guide and to abort the simultaneous traversal of the hierarchies. Our approach can be applied to virtually any hierarchical collision detection algorithm.

2. We present an efficient approach for time-critical collision detection of point clouds. Based solely on the point representation, it can detect intersections of the underlying implicit surfaces without reconstructing a polygonal model. Thus, we can avoid its additional error. Moreover, it can be used to construct the intersection curves. The surfaces do not need to be closed. Our data structure, a point hierarchy, can be built efficiently. Each node stores a sufficient sample of the points plus a sphere covering of a part of the surface. These are used to derive criteria that guide our hierarchy traversal so as to increase convergence. One can be used to

prune pairs of nodes, the other one is used to prioritize pairs of nodes still to be visited. At the leaves we efficiently determine an intersection by estimating the smallest distance.

3. An algorithm for point cloud collision detection that does not need any hierarchical data structure is introduced. Our approach utilizes a proximity graph for a quick interpolation search for a common zero of the two implicit functions. If a constant number of intersection points is sufficient, our approach has a running time of $O(\log n)$, where $n$ denotes the size of the point cloud. For non-deformable point clouds, the running time can be bounded by only $O(\log \log n)$. Moreover, it can also be used to accelerate our first approach for point cloud collision detection using point cloud hierarchies.

4. We give a new definition of an implicit surface over a noisy point cloud, based on the weighted least squares approach. Our main idea is to use a different kernel function that approximates geodesic distances on the surface by utilizing a geometric proximity graph. Given a point cloud consisting of $n$ points, the implicit function can be evaluated very quickly in time $O(\log n)$, but artifacts are significantly reduced. The proximity graph allows us to estimate the local sampling density, which we utilize to automatically adapt the bandwidth of the kernel and to detect boundaries. Consequently, our method is able to handle point clouds of varying sampling densities without manual tuning. Therefore, it constitutes the optimal basis for our point cloud collision detection approaches.

5. A theoretical model to estimate the expected running time of hierarchical collision detection is proposed. We show that the average running time for the simultaneous traversal of two binary bounding volume hierarchies with $n$ leaves depends on two characteristic parameters: the overlap of the root bounding volumes and the bounding volume diminishing factor within the hierarchies. With this model, we show that the average running time is in $O(n)$ or even in $O(\log n)$ for realistic cases. This approach can be used for analyzing the running time of our average-case approach as well as for our hierarchical point cloud collision detection.

This work is organized as follows. In Chapter 2, we start by defining the problem of collision detection and give an overview of the related work. We categorize the algorithms with respect to the kind of geometry they are developed for, as well as the type of used data structure.

In Chapter 3, our average-case approach for time-critical collision detection is presented [KZ03a, KZ03b]. Moreover, we analyze the running time for any hierarchical collision detection algorithm in the average case [KZ05a].
The following Chapter 4 is concerned with our new surface definition over point clouds [KZ04a, KZ04c, KZ04d], which is based on proximity graphs. There, we show how to approximate geodesic distances by shortest path in a proximity graph, and how to compute the local kernel bandwidth automatically. Further, we analyze the running time and complexity of our algorithms and data structures, respectively.
In Chapter 5, our point cloud collision detection algorithms using a point cloud hierarchy are proposed [KZ04b]. Moreover, the idea of utilizing an interpolation search for the collision detection is shown [KZ05b]. Note that the proof of occupying the surfels[2] used to initialize the interpolation search was already given in [KKF+02, KKF+04] in a different context.
Finally, Chapter 6 concludes this thesis and describes possible avenues for further work. Note that each chapter also consists of a summary and ideas for future work. In contrast to Chapter 6, these are much more technical and detailed.

[2]Unlike classical surface discretizations, i.e., triangles or quadrilateral meshes, surfels (surface elements) are point primitives without explicit connectivity. Surfel attributes comprise depth, texture color, normal, and others [PvBZG00].

# 2 Related Work

Boyse [Boy79] presented one of the first 3D collision detection approaches for the purpose of robotics and automation. More than 25 years later, collision detection is still a very fascinating area of research and, more than ever, is a very hot topic at SIGGRAPH [JP04, GKJ+05].

In the following, we describe the literature most relevant to our problem. This chapter is organized as follows: after the problem statement, we classify polygonal collision detection algorithms with respect to their data structures, followed by an overview of point cloud collision detection approaches. Thereafter, time-critical methods for approximate collision detection are considered. Finally, we summarize and discuss the work.

## 2.1 Problem Statement

We define a *collision* as the configuration (also called constellation) of two objects whose surfaces intersect. In the following, we are interested in configurations that change over time, that means the objects are moving in space. In the real world, such motion is usually a continuous flow of the object configurations. In computer animation systems, however, this motion is often simulated by discrete updates of the objects [vdB03].

That means, the actual collision can fall between two discrete time steps where the object positions are updated. As a consequence, backtracking methods can be used to compute the time of first contact, which is often required in constraint-based dynamics simulation methods [MW88, Bar90]. Here, we are not interested in detecting such *in-between* collisions [BFA02, RKC02] which is the goal of *continuous collision detection*.

The main focus of our work is to detect efficiently whether two given objects in a certain constellation collide.

Note that we are not concerned with the broad phase of collision detection (also called $n$-body processing) that identifies smaller groups of objects that may collide and quickly excludes objects that definitively do not collide.

## 2.2 Polygonal Collision Detection

Modern graphics hardware still use polygons as the fundamental rendering primitive[1]. As a consequence, most collision detection approaches were developed only for polygonal object representations. In the case of *polygonal meshes*, the collision detection algorithm can exploit some features, e.g., it can distinguish between *inside* and *outside* of an object. However, in most cases, the polygons are given only as unordered *polygon soups*.

### 2.2.1 Bounding Volume Hierarchies

Bounding volume hierarchies (BVHs), see [ZL03, TKH+05] for an overview, have proven to be a very efficient data structure in the area of collision detection, even for (reduced) deformable models [JP04].
The idea of BVHs is to partition the set of object primitives, e.g. polygons or points, recursively until some leaf criterion is met. In most cases, each leaf contains a single primitive, but the partitioning can also be stopped when a node contains less than a fixed number of primitives. Each node in the hierarchy stores a subset of the primitives and a bounding volume (BV) that encloses this subset.
There are two conflicting constraints for choosing an appropriate BV. On the one hand, a BV-BV overlap test during the traversal should be done as fast as possible. On the other hand, BVs should enclose their subset of primitives as tight as possible so as to minimize the number of false positives with the BV-BV overlap tests. As a consequence, a wealth of BV types has been explored in the past, such as spheres [Hub96, PG95], OBBs [GLM96], DOPs [KHM+98, Zac98], Boxtrees [Zac02, AdBG+01], AABBs [vdB97, LAM01], spherical shells [KGL+98b] and convex hulls [EL01] (for a thorough discussion of "optimal" BVs and BVHs see, e.g., [Got00, Klo98, Zac00]). In order to capture these two characteristics and to estimate the time required for a collision query, the cost function

$$T = N_v C_v + N_p C_p + N_u C_u + C_i$$

was proposed, where

$$N_v, C_v = \text{ number and average costs of BV overlap tests}$$
$$N_p, C_p = \text{ number and average costs of primitive intersection tests}$$

[1] although a polygon is not a *natural* modeling primitive that often arises not before the post-processing of the designer's work.

```
traverse(A, B)
if A and B do not overlap then
    return
if A and B are leaves then
    return intersection of primitives enclosed by
    A and B
else
    for all children A_i and B_j do
        traverse(A_i, B_j)
```

Algorithm 1: Most hierarchical collision detection methods implement this algorithm, or a variation thereof, to traverse two given BVHs. (For sake of simplicity, the case of intersection between inner BV and leaf BV has been omitted.) The corresponding BV test tree can be found in Figure 3.12.

$$\begin{aligned} N_u, C_u &= \text{ number and average costs of BV updates} \\ C_i &= \text{ initialization costs.} \end{aligned}$$

An example of a BV update is the transformation of the BV into a different coordinate system. During a simultaneous traversal of two BVHs, the same BVs might be visited multiple times. However, if the BV updates are not saved, then $N_v = N_u$. This cost function was introduced by [WHG84] to analyze hierarchical methods for ray tracing and later adapted to hierarchical collision detection methods by [GLM96, KHM+98, He99].
In an asymptotic analysis, $N_v$, the number of overlap tests, defines the running time, i.e., $T(n) \sim N_v(n)$, because $N_p = \frac{1}{2}N_v$ in a binary tree and $N_u \leq N_v$. While it is obvious that $N_v = n^2$ in the worst case, the expected running time in the average case was not analyzed in the past (we show the expected running time of simultaneous BVH traversals in Section 3.7).

### Hierarchical Bounding Volume Traversal

Now, given two BV hierarchies for two objects, virtually all collision detection approaches traverse the hierarchies simultaneously by an algorithm similar to Algorithm 1. It allows to quickly zoom into areas of close proximity. The algorithm (usually) stops if an intersection is found or if the traversal has visited all relevant sub-trees[2].
A non-simultaneous traversal was introduced by [vdB97]. He tries to minimize

[2]Note that Algorithm 1 does not take into account cases where an inner node and a leaf node intersect.

the probability of an intersection as fast as possible. Therefore, it is proposed to test the node with the smaller volume against the children of the node with the larger volume.
The simultaneous traversal has the benefit of more quickly traversing to the leaves and making fewer internal node-node overlap tests [Eri04]. However, it has been shown [Chu98, Eri04] that in some special cases, this rule will not prune the search space as effectively as the traversal strategy proposed in [vdB97].
For massive data sets, hybrid approaches such as a grid of trees are commonly more effective than a single BVH [Eri04].
The characteristics of different hierarchical collision detection algorithms lie in the type of BV used, the overlap test, and the algorithm for constructing the BVH. In the case of *boolean collision detection* where no (discrete) collision points are needed but only the information whether two objects collide, the traversal criterion that decides to which pair of BV priority is given, is another very important factor.

### Constructing and Updating BVHs

There are different possibilities for constructing BVHs: insertion [GS87], bottom-up [RL85], and top-down, which is the most commonly used strategy.
Gottschalk et al. presented an approach that utilizes the principal component analysis for finding splitting planes [GLM96]. Alternatively, a balanced tree can be constructed by placing the splitting plane through the median of all points. However, it is not clear whether balanced trees lead to faster collision queries.
For AABBs, it has been shown that any splitting heuristic should try to minimize the volumes of the children [Zac02]. Let $A$ and $B$ denote the direct predecessors of the BVs $A_1$ and $B_1$ of two differen hierarchies. Then, using Minkowski sums $\oplus$ of BVs (see Figure 2.1), one can estimate the probability of a BV overlap during the traversal by

$$Pr(A_1 \cap B_1 \neq \varnothing) = \frac{\mathrm{Vol}(A_1 \oplus B_1)}{\mathrm{Vol}(A \oplus B)} \approx \frac{\mathrm{Vol}(A_1) + \mathrm{Vol}(B_1)}{\mathrm{Vol}(A) + \mathrm{Vol}(B)} \tag{2.1}$$

in the case of AABBs. Since $\mathrm{Vol}(A) + \mathrm{Vol}(B)$ has already been committed by an earlier step in the recursive construction, Equation 2.1 can be minimized only by minimizing $\mathrm{Vol}(A_1) + \mathrm{Vol}(B_1)$.
Volino and Magnenat-Thalmann [VM94, VM95] as well as Provot [Pro97] use a different kind of strategy based on the mesh topology of the objects. These connectivity-based approaches yield advantages in speeding-up self-collision of deformable objects [ZY00, BW02, FGL03, HTG04].
A different approach to reducing query times is to try to learn and model the query probability distribution either before the hierarchy construction [ACT00]

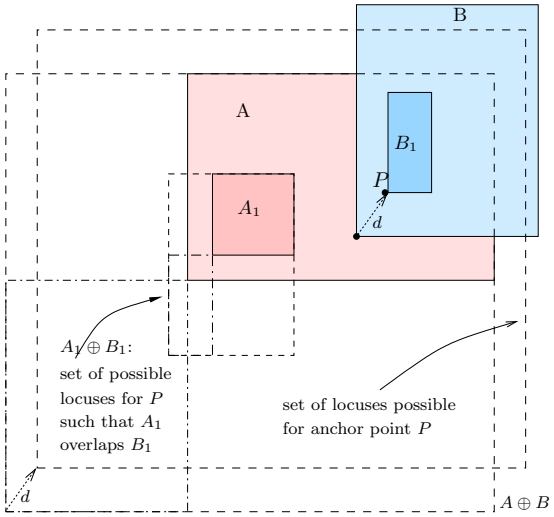



**Figure 2.1:** The probability of an overlap for $A_1$ and $B_1$ can be determined by Minkowski sums [Zac02].

or at running time [AMT02] (i.e., the construction is done on-demand). While being quite effective, their data structures and traversal algorithms are still pretty much the classical ones (besides the fact that they apply the approach only to the problem of detecting an intersection between a line segment and the environment/object).
The influence of the arity of the BVHs were examined by [LAM01, MKE03]. It turned out that there is no substantial difference for rigid objects, while a degree of 4 or 8 seems to be optimal for deformable objects [TKH+05].
In case of deformable objects (see [TKZ+04, TKH+05, ZTK+05] for a comprehensive survey), a BVH has to be updated each time step. To avoid a complete update of all corresponding BVs, different update strategies have been developed. Larsson and Akenine-Mller [LAM01] showed that if many deep nodes are reached during the traversal, the bottom-up strategy performs better, while otherwise the top-down approach is faster. Mezger et al. [MKE03] proposed a lazy hierarchy update to accelerate the update process for *slow* parts of the scene and for small time step sizes.

### 2.2.2 Space-Subdivision Approaches

In contrast to the object-partitioning methods, space-subdivision approaches are mainly used in case of deforming objects as they are independent of changes in the object's topology. For the partitioning, an octree [BT95, KSTK98], BSP tree [Mel00] or a voxel grid [Tur90, MPT99, GDO00, ZY00] can be used.

Agarwal et al. partition the space into polygonal cells with some specific properties so that the next time step of an event can directly be determined using a cinetic data structure [ABG+01]. Unfortunately, often such approaches are known only for the two dimensional case, or are very complex in 3D so that an implementation would not be worthwhile.
In [THM+03] spatial hashing with a uniform grid was proposed for deformable tetrahedral meshes. Their hash function allows for a very memory efficient data structure and for handling potentially infinite regular spatial grids with a non-uniform or sparse distribution of object primitives.

### 2.2.3 Distance Fields

In general, non-hierarchical data structures seem to be more promising for collision detection for deformable objects [ABG+01, HMB01, FL01], although some geometric data structures suggest a natural BV hierarchy [LCLL02]. Many approaches use distance fields, e.g., [FL01, VSC01, SPG03, FSG03] that are updated after a deformation.
A distance field specifies the minimum distance to a surface for all (discrete) points in the field. In literature, different data structures have been proposed for representing distance fields, e.g., octrees, BSP trees or uniform grids. The problem of uniform grids, the large memory consumption, can be solved by a hierarchical data structure called *adaptively sampled distance fields* [FPRJ00]. For the collision detection problem, special attention to the continuity between different levels of the tree has to be paid [BMF03]. In [KEHKL+99] an approach was proposed for computing generalized Voronoi diagrams that can be used to build distance meshes for each site. Sigg et al. [SPG03] have proposed an algorithm utilizing the graphics hardware to determine the distance field in linear time.
Distance fields are most powerful for collision detection between deformable and rigid objects [TKH+05]. Then, vertices of the deformable object can quickly be determined by evaluating the distance field [FSG03]. However, distance fields cannot be updated in real-time, although [VSC01] proposed an image-based approach utilizing the graphics hardware for computing distance values very efficiently. In order to reduce memory consumption, the resolution of the distance field can be decreased so that it allows to balance speed and accuracy.

### 2.2.4 Hardware-Assisted Collision Detection

In the last few years, GPU-based algorithms have become very popular. Their incredible increasing power and several new features are very interesting and

tempting to do the collision detection [SF91, MOK95, LCN99, BW03, GZ03, GLM04] and/or self-collision detection [BW02, GLM03] on the graphics hardware. Most of these methods utilize the stencil buffer or the feedback mechanism of OpenGL.
However, most of the algorithms mentioned above have restrictions to the geometry (e.g., the objects have to be convex or closed), need a lot of time to transfer the data from the graphics hardware to the CPU, and compete with the rendering module for the graphics resources (unless one spends another board just for the collision detection). Note that with modern graphics hardware, buffer readbacks are not necessary any longer [Eri04], because they can be replaced by hardware-assisted occlusion queries. A full GPU implementation for particle systems using fragment shaders was proposed by [KLRS04]. Their collision detection is based on depth maps that represent the outer shape of an object.
Note that all image-space techniques work with discretized object representations so that they do not allow for exact collision detection. That means, the accuracy and running time can be balanced by changing the resolution of the rendering.
An alternative idea is to build a special hardware architecture for a single-chip accelerator, only for the collision detection process [ZK03]. To get maximum performance, they use BVHs defined by k-DOPs.

### 2.2.5 Theoretical Results

In the past few years, some very interesting theoretical results on the collision detection problem have been shown.
One of the first results was presented by Dobkin and Kirkpatrick [DK83, DK85, DK90]. They have shown that the distance of two convex polytopes can be determined in time $O(\log^2 n)$, where $n = \max\{|A|, |B|\}$ and $|A|$ and $|B|$ are the number of faces of object $A$ or $B$, respectively. This result was obtained by using a hierarchy that can be built in time $O(n)$. If one of the objects is not convex, the intersection can be determined in time $O(n \log n)$ [DHKS93].
For two general polytopes whose motion is only along fixed trajectories, it has been shown that there is an $O(n^{\frac{5}{3}+\epsilon})$ algorithm for rotational movements, where $\epsilon$ is an arbitrary positive constant [ST95]. Moreover, they proposed an $o(n^2)$ algorithm for a more flexible motion that still has to be along fixed, known trajectories [ST96].
Suri et al. [SHH98] have proven that for a set of $n$ convex polytopes which all are well-shaped (with respect to aspect ratio and scale factor) all intersections can be computed in time $O((n + K_o) \log^2 n)$, where $K_o$ denotes the number of intersecting object pairs. They have generalized their approach [ZS99] where

| kind of objects | running time |
|---|---|
| two convex polytopes | $O(\log^2 n)$ |
| convex and non-convex polytope | $O(n \log n)$ |
| two non-convex polytoptes (motion along fixed trajectories) | $o(n^2)$ |
| two arbitrary objects (arbitrary motion) | $O(n^2)$ |

**Table 2.1:** Worst case running time for different kinds of objects.

they proposed the first *average-shape* results in computational geometry. Let $\rho = \frac{K_b}{n+K_o}$ denote the performance of the BV heuristic, where $K_b$ denotes the number of colliding bounding-box pairs and $K_o$ the number of colliding object pairs. Then, they have shown that $\rho = \Theta(\alpha_{avg}^{\frac{2}{3}} \sigma_{box}^{\frac{1}{3}} n^{\frac{1}{3}})$, where $\alpha_{avg}$ is the average aspect ratio and $\sigma_{box}$ denotes the average box scale factor.
In [ZS00a] they extend their work in one important direction: they analyzed the performance of bounding boxes when all objects in the collection are convex. Assuming the objects are in 3D, the performance $\rho$ improves in the average case: $\Theta(\alpha_{avg}^{\frac{1}{2}} \sigma_{box}^{\frac{3}{8}} n^{\frac{1}{4}})$.
Under some mild coherence assumptions, Vemuri et al. [VCC98] have shown a linear expected time complexity for the collision detection between $n$ convex objects. In the worst case, their approach has a running time of $O(n \log n)$. Their basic idea is to use known data structures, namely octrees and heaps, along with the concept of spatial coherence.
The Lin-Canny algorithm [LC91] is based on a closest-feature criterion and makes use of Voronoi regions. Let $n$ be the total number of features, the expected running time is between $O(\sqrt{n})$ and $O(n)$ depending on the shape, if no special initialization is done.
Recently, it has been shown by Agarwal et al. [AGN+04], that collision detection between deforming necklaces can be done in time $O(n^{2-\frac{2}{d}})$ using predefined BVHs ($d$ denotes the dimension).
An overview of the collision detection running time for different kinds of objects can be found in Table 2.1.

## 2.3 Point Cloud Collision Detection

From the set of non-polygonal representations, mainly parametric surfaces like NURBS, splines [Far02] or subdivision surfaces [ZS00b] were examined for the collision detection process [KM97, KGL+98a, JC98, GS01, LCLL02] (note

that parametric surfaces can become inefficient for deformations and topology changes [PKKG03]). Point clouds, however, are virtually not examined for the purpose of collision detection.
After a short overview of defining surfaces over point clouds, the existing work on point cloud collision detection is discussed.

### 2.3.1 Surface Definition

Point clouds [KB04] are a very popular object representation due to modern acquisition methods, like 3D scanning or sampling synthetic objects. As a consequence, a series of efficient rendering techniques were developed [RL00, PvBZG00, WFP+01, KV03, Wan04, BSK04]. However, only very little work has been done on the interaction of such object representations and the inherent collision detection process.
An attractive way of handling point clouds is to define the surface as the zero set of an *implicit function* that is constructed from the point cloud. Usually, this function is not given analytically but "algorithmically" [AA03, AA04a, AA04b]. This is a general method that can be used for reconstruction as well as ray-tracing or collision detection. Another very popular method is to define the surface as the set of fixed points of a projection operator based on local polynomial regression [AK04].

### 2.3.2 Algorithms for Point Cloud Collision Detection

Geometric queries on point clouds have been studied extensively. An interesting result related to our problem can be found in [CLR90, p. 908f]. They use a divide-and-conquer algorithm to find the closest pair of $n$ points in time $O(n \log n)$ which is, of course, not applicable to realtime collision detection.
However, there is very little literature on geometric queries (and the collision detection process) on the implicit surfaces defined by point clouds.
In [AD03] an algorithm to perform Boolean operations for solids is presented. Although the problem of constructing a new solid by Boolean operations and the problem of detecting an intersection in a time-critical scenario are somewhat related, there are many obvious, significant differences. In addition, [AD03] represent objects by surfels. In contrast, we consider a continuous surface defined by a set of points. Furthermore, their approach can handle only solids, because they partition space in "inside" and "outside" by an octree. Our approach is general, i.e., it can handle non-closed geometry.
Tanaka et al. [TFY00] proposed to sample an implicit function with a stochastic differential equation to detect intersections. Since it is a method for general

implicit surfaces, they do not exploit the topology of the surface (as we do by utilizing a proximity graph). In addition, our new method is much simpler and, thus, more practical.
Pauly et al. [PKKG03] use a Newton scheme to obtain the intersection curve of point-sampled geometry. First, they determine all points that are close to the intersection curve by evaluating the distance function induced by the MLS projection operator. Then, they look at all closest pairs of these points and compute a point on the intersection curve using a Newton-type iteration.

## 2.4 Time-Critical Collision Detection

It has often been noted previously, that the *perceived quality* of a virtual environment and, in fact, most interactive 3D applications, crucially depends on the real-time response to collisions [US97], and not on an exact simulation. At the same time, humans cannot distinguish between physically correct and *physically plausible* behavior of objects (at least up to some degree) [BHW96]. Analogously to rendering, a number of human factors determine whether or not the "incorrectness" of a simulation will be noticed, such as the mental load of the viewing person, cluttering of the scene, occlusions, velocity of the objects and the viewpoint, point of attention, etc. Since collision detection is still the major bottleneck of many of these simulations and interactions, it is obvious that this is where the best speedup can be achieved.
BVHs lend themselves well to time-critical collision detection, i.e., the scheduler interrupts the traversal when the time budget is exhausted. This has been observed by several researchers [DO00, Hub96]. Hubbard presented the idea of interruptible collision detection using sphere trees [Hub96]. Dingliana and O'Sullivan [DO00] are concerned with modeling contacts based on interrupted sphere tree traversals. The method described there can be applied in our framework, proposed in Section 3, too. However, they do not provide any theoretical foundations concerning the error incurred by an incomplete traversal. In addition, their methods do not support application-driven "levels-of-detail" of collision detection where the application can specify an allowable error rate beforehand.
Another approach that offers the possibility to balance the quality of the collision detection against computation time is to select random pairs of colliding features as an guess of the potential intersecting regions [RCFC03]. To identify the colliding regions when objects move or deform, temporal as well as spatial coherence can be exploited [LC92]. This stochastic approach, which was improved by Kimmerle et al. [KNF04], can be applied to several collision detection

problems [GD02, DDCB01].
Guy and Debunne [GD04] presented a Monte-Carlo based technique for collision detection. Samples are randomly generated on every object in order to discover interesting new regions. Then, the objects are efficiently tested for collision using a multiresolution layered shell representation, which is locally fitted according to the distance of the objects.

## 2.5 Summary and Discussion

There are several different approaches to the collision detection process. While BVHs are shown to be very efficient for rigid objects, space-subdivision approaches are mainly used for deformable models. For scenarios where deformable objects have to be tested against rigid objects, distance fields are a very elegant and simple solution that also provides collision information like contact normals or penetration depths. Stochastic methods are very interesting for time-critical scenarios because they allow for balancing speed versus accuracy. Hardware-assisted approaches, especially full GPU implementations, are a very promising technique. However, they do not allow for exact collision detection. Point cloud collision detection is a very new research area and not even a handful of algorithms have been proposed in the last few years.
It is impossible to compare all these techniques for several reasons. Only in a very few cases, a theoretical estimation of the running time is given. Moreover, the approaches provide different collision information, make different assumptions, or are specialized for some applications. Last but not least, the approaches require different input data so that a practical evaluation of different algorithms using all the same data set is not achievable.
Therefore, we are not interested in developing and implementing data structures and collision detection approaches that run faster for some test objects under some assumptions. We are rather interested in algorithms that allow for new applications, namely collision detection for point clouds and reducing the quality of collision detection in a controlled way.

# 3 An Average-Case Approach for Real-Time Collision Detection

In this section, we introduce a novel framework for collision detection using an average-case approach. To our knowledge, this is the first time that the *quality* of collision detection has been decreased in a controlled way (while increasing the speed), such that a numeric *measure* of the quality of the results is obtained (which can then be related to the perceived quality). Our methods can be applied to virtually any hierarchical collision detection algorithm.
It is, of course, possible to just cut off the BV traversal any time the application or scheduler deems suitable [Hub96]. The problem with this approach is that it gives absolutely no hint as to the confidence in the result.
In contrast, our novel approach enables an application to trade accuracy for speed in a controlled fashion, so that it always has a "measure of confidence" in the result reported by the collision detection algorithm.

## 3.1 Overview of our Approach

Conceptually, the main idea of the new algorithm is to consider *sets of polygons* at inner nodes of the BV hierarchy, and then, during traversal, check pairs of sets of polygons. However, we neither check pairs of polygons derived from such a

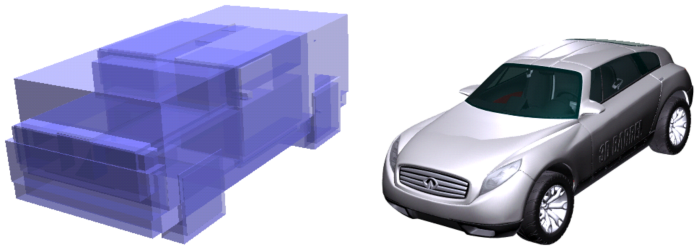


**Figure 3.1:** Bounding volume hierarchies (BVHs) allow for efficient collision detection. Our average-case approach allows one to reduce the quality for the benefit of running time in a controlled way, if assuming a uniform distribution of the polygons inside the intersection volume of two BVs.

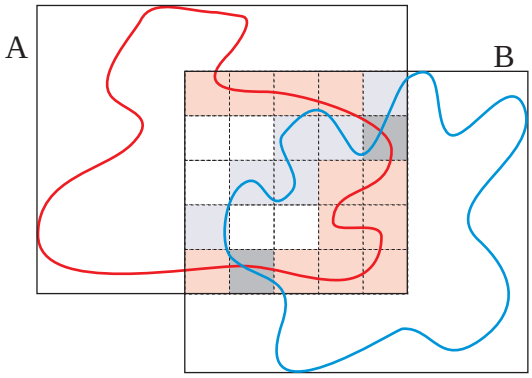



**Figure 3.2:** We partition the intersection volume with a grid. Then, we determine the probability that there are *collision cells* where polygons of different objects *can* intersect (highlighted in grey).

pair of polygon sets, nor store any polygons with the nodes. Instead, based on a small number of parameters describing the *distribution* within the polygon sets, we derive an estimation of the probability that a pair of intersecting polygons *exists*. This probability can be used to guide and to abort the simultaneous traversal of the hierarchies.
The design of our algorithm was influenced by the idea of developing an algorithm that works well and efficiently for most practical cases — in other words, that works well in the average case. Therefore, we estimate the probability of a collision within a pair of BVs using some characteristics of the average distribution of the polygons, but we do not use the exact positions of the polygons during the collision detection.
This has two advantages:

1. The application can control the running time of the algorithm by specifying the desired "quality" of the collision detection (to be defined later).

2. The probabilities can guide the algorithm to those parts of the BV hierarchies that allow for faster convergence of the estimate.

Conceptually, the intersection volume of $A$ and $B$, $A \cap B$, is partitioned into a regular grid (see Figure 3.2). If a cell contains *enough* polygons of one BV, we call it a *possible collision cell* and if a cell is a possible collision cell with respect to $A$ and also with respect to $B$, we call it a *collision cell* (a more precise definition is given in Section 3.2).
Given the total number of cells in $A \cap B$, the number of possible collision cells from $A$ and $B$, respectively, lying in $A \cap B$, we can compute the probability that there are at least $x$ collision cells in $A \cap B$.

This probability can be used to estimate the probability that the polygons from $A$ and $B$ intersect. For the computations, we assume that the probability of being a possible collision cell is evenly distributed among all cells of the partitioning because we are looking for an algorithm that works well in the average case where the polygons are uniformly distributed in the BVs or in the intersection volume, respectively.
Of course, this assumption is more realistic for smaller BVs (compared to the object size) than for larger ones. Later in Section 3.4.2 we pay special attention to uneven polygon distributions.
An outline of our traversal algorithm is given by Algorithm 2. Function computeProb estimates the probability of an intersection between the polygon sets of two BVs. By descending first into those sub-trees that have highest probability, we can quickly increase the confidence in the result and determine the end of the traversal. Basically, we are now dealing with priorities of pairs of nodes, which we maintain in a priority queue.
The priority queue contains only pairs whose corresponding polygons can intersect. It is sorted by the probability of an intersection. Instead of a recursive traversal, our algorithm just extracts the front node pair from the queue and inserts a number of child pairs.
The quality and speed of the collision detection strongly depends on the accuracy of the probability computation. Several factors contribute to that, such as the kind of partitioning and the size of the polygons relative to the size of the cells. This is discussed in more detail in Section 3.8.
There are two other important parameters in our traversal algorithm, $p_{min}$ and $k_{min}$, that affect the quality and the speed of the collision detection. Both can be specified by the application every time it performs collision detection. A *pair of collision nodes* is found if the probability of an intersection between their associated polygons is larger than $p_{min}$. A collision is reported if at least $k_{min}$ such pairs have been found. The smaller $p_{min}$ or $k_{min}$ is, the shorter the running time is and, in most cases, the more errors are made.
The remainder of this section explains this framework more precisely in a top-down manner.

## 3.2 Terms and Definitions

For the sake of accuracy and conciseness, we introduce the following terms and definitions. We treat the terms *bounding volume* (BV) and *node* of a bounding volume hierarchy (BVH) as synonym. $A$ and $B$ will always denote BVs of two different hierarchies.

```
traverse(A, B)
priorityQueue q, k:=0
q.insert(A, B, 1)
while q is not empty do
    A, B := q.pop
    for all children A[i] and B[j] do
        p := computeProb(A[i], B[j])
        if p ≥ p_min then
            k:=k+1
            if k ≥ k_min then
                return "collision"
        if p > 0 then
            q.insert(A[i], B[j], p)
return "no collision"
```

Algorithm 2: Our algorithm traverses two BV hierarchies by maintaining a priority queue of BV pairs sorted by the probability of an intersection. In case of interrupting the traversal, the maximal probability that has been determined so far can be returned.

**Definition 1.** *All polygons of the object contained in BV $A$ or intersecting $A$ are denoted as $P(A)$.*
*Let $c$ be a cell of the partitioning of $A \cap B$. The total area of all polygons in $P(A)$ clipped against cell $c$ is denoted as* $\text{area}_c(A)$.
$\text{maxArea}(c)$ *denotes the area of the largest polygon that can be contained completely in cell $c$.*

**Definition 2** (possible collision cell)**.** *Given a BV $A$ and a cell $c$. $c$ is a possible collision cell, if* $\text{area}_c(A) \geq \text{maxArea}(c)$.

**Definition 3** (collision cell)**.** *Given two intersecting BVs $A$ and $B$ as well as a partitioning of $A \cap B$. Then, $A$ and $B$ have a (common) collision cell iff* $\exists c : \text{area}_c(A) \geq \text{maxArea}(c) \wedge \text{area}_c(B) \geq \text{maxArea}(c)$ *(with suitably chosen* $\text{maxArea}(c)$*).*

Definitions 2 and 3 are actually the first steps towards computing the probability of an intersection among the polygons of a pair of BVs. In particular, Definition 3 is motivated by the following observation. Consider a cubic cell $c$ with side length $a$, containing exactly one polygon from A and B, respectively. Assuming $\text{area}_c(A) = \text{area}_c(B) = \text{maxArea}(c)$, then we must have exactly the configuration shown in Figure 3.3, i.e., an intersection, if we choose

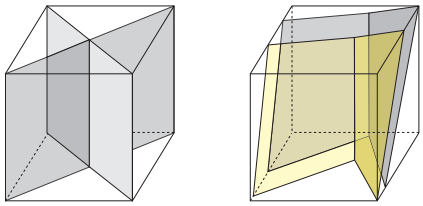


**Figure 3.3:** Left: a cubic collision cell $c$ with side length $a$. $\text{area}_c(A)$ and $\text{area}_c(B)$ must be at least $\text{maxArea}(c) = a^2\sqrt{2}$, which is exactly the area of the two quadrangles. Right: although the cell contains *enough polygons*, there is no collision inside.

$\text{maxArea}(c) = a^2\sqrt{2}$. Obviously, a set of polygons is not planar (usually), so even if $\text{area}_c(A) > \text{maxArea}(c)$ there might still not be an intersection. But since almost all practical objects have bounded curvature in most vertices, the approximation by a planar polygon fits better and better as the polygon set covers smaller and smaller a surface of the object.

**Definition 4** (lower bound $lb(A \cap B)$)**.** *Given an arbitrary collision cell $c$ from the partitioning of $A \cap B$. A lower bound for the probability that a collision occurs in $c$ is denoted as $lb(A \cap B)$.*

Let us conclude this subsection by the following important definition.

**Definition 5** ($Pr[n_{col}(A \cap B) \geq x\ ]$)**.** *The probability that at least $x$ collision cells exist in $A \cap B$ is denoted as $Pr[n_{col}(A \cap B) \geq x]$.*

Overall, given the probability $Pr[n_{col}(A \cap B) \geq 1]$, a lower bound for the probability that the polygons from $A$ and $B$ intersect is given by

$$Pr[P(A) \cap P(B) \neq \varnothing] \geq Pr[n_{col}(A \cap B) \geq 1] \cdot lb(A \cap B). \tag{3.1}$$

A better lower bound is given in Section 3.5.1, where $x > 1$ is used for the probability $Pr[n_{col}(A \cap B) \geq x]$. Section 3.5.1 will derive $Pr[n_{col}(A \cap B) \geq x]$, while Section 3.5.3 will derive $lb(A \cap B)$.

## 3.3 Data Structure

As mentioned before, our approach is applicable to virtually all BV hierarchies by augmenting them with a simple description of the distribution of the set of polygons. The resulting hierarchies are called *ADB trees* (***a**verage-**d**istri**b**ution trees*). In the following, we explicitly mention the type of BV only if necessary.

### ADB Trees

Our function computeProb$(A, B)$ needs to estimate the probability $Pr[n_{col}(A \cap B) \geq x]$ that is defined in the previous section. However, partitioning $A \cap B$ at runtime is too expensive.

Therefore, we partition each BV after the construction of the hierarchy into a fixed number of cuboidal cells, (the partitioning is discussed in more detail in Section 3.8) and then we count the number of *possible collision cells* according to Definition 2 and store it with the node. Note that, thanks to our average-case approach making the assumption that each cell of the partitioning has the same probability of being a possible collision cell, we are not interested in exactly which cells are possible collision cells, but only in their number. As a consequence, this additional parameter per node incurs only a very small increase in memory footprint of the BV hierarchy, even when utilizing very "light-weight" nodes such as spheres [Hub96] or restricted boxes [Zac02]. It is, of course, computed during preprocessing after the construction of the BV hierarchy.

With rectangular BVs at the nodes of the hierarchy, this kind of partitioning obviously causes no problems. Using other BVs, such as DOPs, some of the cells at the border of a BV are not cubic. It is clear that the total area of polygons in such a cell does not need to be as large as in other cubic cells, in order to qualify as a possible collision cell. This is why Definitions 2 and 3 depend on maxArea$(i)$.

Note that we do not need to store any polygons or pointers to polygons in leaf nodes. A possible intersection is determined solely based on the probabilities described so far.

In addition to the ADB trees, we will need a number of lookup tables in order to compute $Pr[n_{col}(A \cap B) \geq x]$ efficiently (see Section 3.5). Fortunately, they do not depend on the objects nor on the type of BV, so we need to precompute the lookup tables only once.

### Counting Possible Collision Cells

We propose two algorithms for computing the number of possible collision cells. It is convenient to do this after the BV hierarchy has already been built.

The first one is very simple and computes an exact value of the number. It partitions each node into a grid, clips the polygons associated with the node, inserts the fragments into the grid, and counts the number of possible collision cells. Assuming a complete binary BVH, the running time of this algorithm can be estimated as

$$T_1 = c_1 n \log m + c_1' m,$$

**posColCells**$(A, A')$
$pc := 0$
**if** $\text{Vol}(A') \leq \text{Vol}(c_A)$ **then**
  **if** $\text{area}(A') \geq \text{maxArea}(c_A)$ **then**
    **return** 1
**else**
  **if** $A'$ is a leaf **then**
    **if** $\text{area}(A') \geq \text{maxArea}(A')$ **then**
      **return** $1 \cdot \frac{\text{Vol}(A')}{\text{Vol}(c_A)} \cdot \frac{1}{\sqrt[3]{\frac{\text{Vol}(A')}{\text{Vol}(c_A)}}}$
  **else**
    **for all** children $A'[i]$ **do**
      $pc := pc +$ posColCells$(A, A'[i])$
**return** $pc$

Algorithm 3: The number of possible collision cells in BV $A$ can be approximated efficiently by propagating polygon areas up through the tree. $c_A$ denotes an arbitrary cell of $A$, $n_{pcol}(A) :=$ posColCells$(A, A)$.

where $n$ is the number of polygons and $m$ is the number of nodes in the hierarchy. $c_1$ is the cost of clipping and inserting one polygon, while $c_1'$ is the (average) cost of counting the number of possible collision cells of one node.
$c_1 n \log m$ time is needed because for each of the $\log m$ tree levels, $n$ polygons have to be clipped and inserted, and $c_1' m$ is needed for counting the number of possible collision cells for all nodes.
The second algorithm (Algorithm 3) approximates the number $n_{pcol}(A)$ of possible collision cells for a node $A$. Its main idea is to use the sub-tree of $A$ for the computation of $n_{pcol}(A)$. The algorithm looks for child nodes $A'$ of $A$ with $\text{Vol}(A') \leq \text{Vol}(c_A)$, which is the size of one cell of $A$. If such a child node contains *enough* polygons (in some sense), then we increase $n_{pcol}(A)$ by 1. Therefore, we do not need to partition $A$ into a grid and test each cell.
Instead, we look for "cells" that are arbitrarily positioned in $A$ and check whether they are possible collision cells.
Of course, the recursive search for such cells could end at a leaf node $A'$. Then, if this node contains *enough* polygons, we approximate the number of possible collision cells by $n_{pcol}(A') := 1 \cdot \frac{\text{Vol}(A')}{\text{Vol}(c_A)} \cdot \frac{1}{\sqrt[3]{\frac{\text{Vol}(A')}{\text{Vol}(c_A)}}}$.
We will explain this later in Section 3.4.1 (Equation 3.6). Note that the algorithm posColCells$(A, A)$ (see Figure 3) determines only $n_{pcol}(A)$, and not

$n_{pcol}(A')$, where $A'$ denotes a child of $A$. As a consequence, *posColCells* has to be started for each node of the hierarchy.
Let $c_2$ denote the cost for checking one node whether it is a possible collision cell. Then, the running time for computing $n_{pcol}(A)$ for all nodes of a complete binary BVH can be estimated by

$$T_2 = \sum_{i=0}^{d} c_2 2^i (2^{d-i+1} - 2) = c_2 2^{d+1}(d-1) + 2 \leq c_2 m \log m$$

because for a node of depth $i$, maximal $2^{d-i+1}-2$ child nodes have to be checked. Obviously, $T_2 \ll T_1$, because $c_1' \gg \log m$, and because $c_1$ is a very expensive operation compared to $c_2$. Of course, these considerations are only valid, because the construction of the BV hierarchy (which is done in the first phase) is not the dominant factor. Indeed, our experiments in Section 3.8 show that our second algorithm is substantially faster than the first one so that it can be performed at startup time (note that $m = O(n)$, so that the possible collision cells can be determined in time $O(n \log n)$). This is very important for industrial applications, because it is very difficult to establish additional data files in the workflow and data flow of design and production processes.

## 3.4 Probability Parameters

As will be explained in Section 3.5, $Pr[n_{col}(A \cap B) \geq x]$ can be computed from the following three parameters only:

$$\begin{aligned} s &= \#\text{ cells contained in } A \cap B, \\ s_A &= \#\text{ possible collision cells from } A \text{ in } A \cap B, \\ s_B &= \#\text{ possible collision cells from } B \text{ in } A \cap B. \end{aligned}$$

In this section, we explain how to determine them during the collision detection process. Algorithm 4 gives an overview of computing the probability that the polygons in $A$ and $B$ intersect.

### 3.4.1 Uniform polygon distribution

For a moment, let us assume that BVs $A$ and $B$ are of the same size and that the polygons are uniformly distributed in both of them. Later in this section, we will lift both assumptions. Let $n_c(A)$ denote the number of cells lying in $A$.

**computeProb**$(A, B)$
compute $s, s_A, s_B$
look up for $Pr[n_{col}(A \cap B) \geq x]$
using $(s, s_A, s_B)$
estimate $Pr[P(A) \cap P(B) \neq \varnothing]$ by
$Pr[n_{col}(A \cap B) \geq x]$ and $lb(A \cap B)$

Algorithm 4: This algorithm estimates the probability $Pr[P(A) \cap P(B) \neq \varnothing]$ by only three parameters, that can be computed efficiently on the fly.

Then, the number $s$ of cells in $A \cap B$ can easily be approximated by

$$s = \frac{\text{Vol}(A \cap B)}{\text{Vol}(A)} \cdot n_c(A) \tag{3.2}$$

Analogously, the parameters $s_A$ and $s_B$ are computed depending on the number of possible collision cells of $A$ and $B$ that have been determined during preprocessing.
Obviously, the cells of the preprocessing partitioning of $A$ and $B$ are not congruent with the cells of the partitioning of $A \cap B$. But congruence is not needed, because our probability computations are only based on the *number* of possible collision cells and the *number* of cells lying in $A \cap B$; they are not based on geometrical properties.

### Differently sized BVs

Now, consider the case that $A$ and $B$ are of different size. Without loss of generality, the cells in $A$ are smaller than the cells in $B$. Then, we first compute $s$ and $s_A$ as described above. If we would also compute $s_B$ this way, we would get too small a probability of collision because the number of possible collision cells would be assumed too small. In practice, the quality of the collision detection would not be affected but the performance, because the traversal would stop later than necessary.
As a remedy, we have to compute $s_B$ depending on a partitioning with a cell size equal to the cell size of the BV $A$. Therefore, we look for child nodes of $B$ whose sizes are (almost) equal to the size of $A$ and compute $s_B$ depending on these nodes. As a consequence, we have to traverse to the child nodes of $B$ and we stop the traversal at a node $B_i$ if $\text{Vol}(B_i) \leq \text{Vol}(A)$. Let $n_{pcol}(B_i)$ denote the number of possible collision cells lying in $B_i$. Then, we compute $s_{B_i}$ by

$$s_{B_i} = \frac{\text{Vol}(B_i \cap A)}{\text{Vol}(B_i)} \cdot n_{pcol}(B_i) \tag{3.3}$$

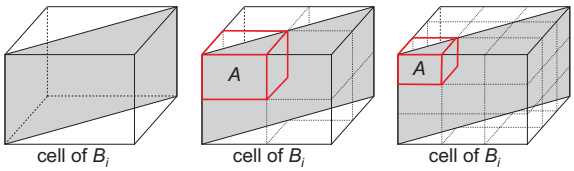



**Figure 3.4:** While the cell sizes differ by $\tau_{AB_i} = \text{Vol}(B_i)/\text{Vol}(A)$, the number of possible collision cells differs by $\tau_{AB_i} \cdot 1/\sqrt[3]{\tau_{AB_i}}$, if the polygons are aligned as shown in this figure.

and $s_B$ depending on the $n$ nodes $B_i$ where the traversal was aborted

$$s_B = \sum_{i=1}^{n} s_{B_i}. \tag{3.4}$$

In the case the traversal has reached a leaf node $B_i$, it could happen that $\text{Vol}(B_i)$ is still larger than $\text{Vol}(A)$. Then, we compute $s_{B_i}$ by Equation 3.3 and derive $s_{B_i'}$ that denotes the number of possible collision cells lying in $A \cap B_i$ depending on a partitioning, where the cell size is equal to that of $A$.
It is self-evident to compute $s_{B_i'}$ by increasing $s_{B_i}$ depending on the ratio between the sizes of $A$ and $B_i$. But if we would only increase $s_{B_i}$ by

$$\tau_{AB_i} := \text{Vol}(B_i)/\text{Vol}(A) \tag{3.5}$$

it could happen that this would be an overestimate. Figure 3.4 illustrates the problem. A single cell from the partitioning of the BV $B_i$ (left) encloses the same volume as the 8 cells in the middle and as the 27 cells in the right. The cells in the middle and on the right have the same volume as the cells of a BV $A$. That means, the cell sizes differ by a factor of 8 and 27, but the number of possible collision cells differs only by $8 \cdot 1/2 = 4$ and $27 \cdot 1/3 = 9$.
Of course, this is only correct, if the polygons are aligned as shown in the figure. But as we are dealing with leaf nodes, the assumption that the polygons can be approximated by a single plane (as shown in our figure) is realistic for real-world models. We will discuss this later in detail in Section 3.5.3. As a consequence, we compute $s_{B_i'}$ by Equation 3.6 and use this instead of $s_{B_i}$ for evaluating Equation 3.4.

$$s_{B_i'} = s_{B_i} \cdot \tau_{AB_i} \cdot \frac{1}{\sqrt[3]{\tau_{AB_i}}} = s_{B_i} \cdot \sqrt[3]{\tau_{AB_i}^2} \tag{3.6}$$

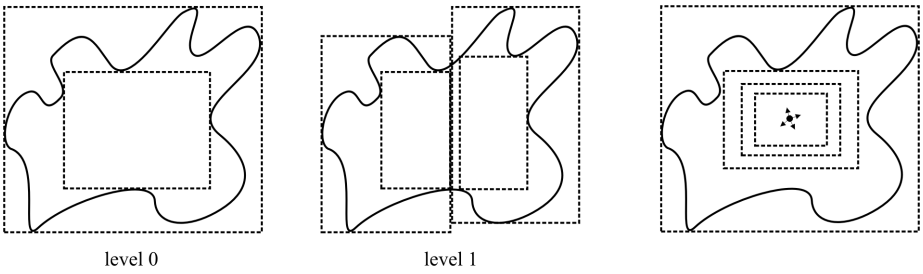



**Figure 3.5:** The assumption, the polygons are evenly distributed in a BV could be too strong. As a remedy, we propose to use two boundaries for each BV: an inner and an outer boundary. Right: an inner boundary can quickly be determined by testing some BVs centered at the point of gravity of the outer BV (or at a point in its neighborhood).

### 3.4.2 Non-uniform polygon distribution

Until now, we have assumed that the polygons are uniformly distributed in the BVs $A$ and $B$. In practice, this is obviously not always correct so that the parameters $s, s_A$, and $s_B$ could lead to probabilities that are too low or too high. The problem arises because there can be a lot of free space within a BV where no polygons are inside. Using only a tighter BV (e.g., a convex hull) would not improve the situation a lot.

As a remedy, we propose to use two boundaries for each BV: an inner and an outer boundary, as shown in Figure 3.5, where the polygons lie between those two boundaries. More precisely, the outer boundary is just the bounding volume enclosing all the polygons. The inner boundary is of the same type (AABB, OBB, DOP, ...) as the outer one. However, no polygons are allowed to lie within the inner boundary. Corresponding to the type of BV used, we denote the resulting BV consisting of two boundaries a 2-AABB, 2-OBB, 2-DOP, and so on.

In comparison with a simple BV, the assumption the polygons are evenly distributed within those boundaries is much more realistic because there is clearly less free space where no polygons are inside. During the traversal, the intersection volume of two such BVs has to be determined, which is very easy in the case of two 2-AABBs (see Figure 3.6).

The inner boundary should be computed in the preprocessing. Unfortunately, to the best of our knowledge, there is no efficient algorithm to determine an optimal inner boundary, that means a boundary that encloses the maximal possible volume. The work of Cohen-Or et al. [DCOT03] is related to that problem. However, they determine only an inner-cover of a non-convex polygon. As a consequence, we propose a simple heuristic to determine a *good* inner

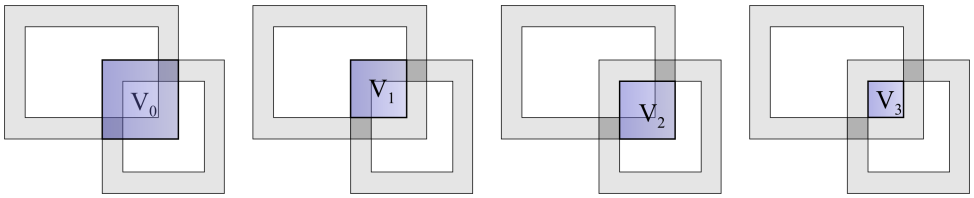



**Figure 3.6:** The intersection volume $V$ of two 2-AABBs can efficiently be determined: $V = V_0 - V_1 - V_2 + V_3$.

boundary, whose corresponding volume can be smaller than the optimal one. First, we determine the center **c** of the outer BV. Then, we test for some differently sized BVs centered at **c**, whether the polygons lie completely outside that BV. Finally, we take the BV with the largest volume as inner boundary. Note that we can move the center a little bit to test some more constellations. The situation is illustrated in Figure 3.5 (right).

## 3.5 Probability Computations

In this section, we explain the computation of the probability $Pr[n_{col}(A \cap B) \geq x]$ and its usage.

### 3.5.1 Probability of collision cells

Recall that computeProb$(A, B)$ estimates the probability that polygons contained in two BVs $A$ and $B$ intersect (see Figure 4).
Given a partitioning of $A \cap B$ and the numbers $s, s_A, s_B$, the question is: what is the probability that at least $x$ of the $s$ cells are possible collision cells of the $s_A$ cells and are *also* possible collision cells of the $s_B$ cells? This is the probability that at least $x$ collision cells exit.
Note that the $s_A + s_B$ possible collision cells are randomly but not independently distributed among the $s$ cells: obviously, it can never happen that two or more of the $s_A$ or $s_B$, respectively, possible collision cells are distributed on the same cell, i.e., $s_A$ possible collision cells are distributed on exactly the same number of cells of the partitioning. This problem can be stated more abstractly and generalized by the following definition.

**Definition 6** ($Pr[\#\ filled\ bins \geq x]$)**.** *Given $u$ bins, $v$ blue balls, and $w$ red balls. The balls are randomly thrown into the $u$ bins, whereby a bin never gets two or more red or two or more blue balls. The probability that at least $x$ of the $u$ bins get a red and a blue ball is denoted as $Pr[\#\ filled\ bins\ \geq x]$.*

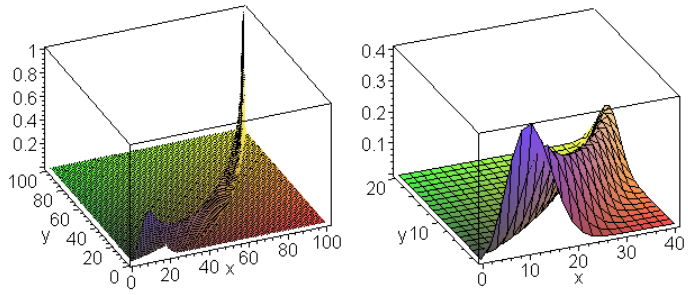



**Figure 3.7:** The probability, that exactly $t$ of the $u$ bins get a red and a blue ball can be determined by the hypergeometric distribution. Here, the probability is shown for a fixed $u = 100$ depending on $x = v = w$ and $y = t$.

If $u = s, v = s_A$ and $w = s_B$, this definition is related to our original problem by the following observation, because we assume that each cell of the partitioning has the same probability of being a possible collision cell.

**Observation 1.**
$Pr[n_{col}(A \cap B) \geq x] \approx Pr[\#\ \textit{filled bins}\ \geq x]$.

Now, let us determine $Pr[\#$ filled bins $\geq x]$. The probability, that exactly $t$ of the $u$ bins get a red and a blue ball, is

$$\frac{\binom{w}{t}\binom{u-w}{v-t}}{\binom{u}{v}}.$$

Figure 3.7 illustrates this probability[1].
*Explanation:* Let us assume, the $w$ red balls have already been thrown into the $u$ bins. Now, the question is: what is the probability that $t$ of the $v$ blue balls are thrown into bins containing a red ball? $\binom{u}{v}$ is the number of possibilities to distribute the $v$ blue balls to the $u$ bins. The number of possibilities that $t$ of the $v$ blue balls are distributed to bins already filled with a red ball is $\binom{w}{t}$. And the remaining $v - t$ blue balls have to be distributed simultaneously to the $u - w$ empty bins.
Thus, the probability that at least $x$ of the $u$ bins get a red and a blue ball, is

$$Pr[\#\ \text{filled bins}\ \geq x] = 1 - \sum_{t=0}^{x-1} \frac{\binom{w}{t}\binom{u-w}{v-t}}{\binom{u}{v}} \tag{3.7}$$

[1] The corresponding probability space $(\Omega, \Sigma, P)$ is defined as: $\Omega = \{\omega_t\}$, where $\omega_t$ denotes the event, that exactly $t$ of the $v$ blue balls are thrown into bins containing a red ball; $\Sigma = \mathcal{P}(\Omega)$, the power set of $\Omega$, denotes a $\sigma$-algebra on $\Omega$; the probability measure $P$ is defined by a hypergeometric distribution, $P(\omega_t) = \frac{\binom{w}{t}\binom{u-w}{v-t}}{\binom{u}{v}}$.

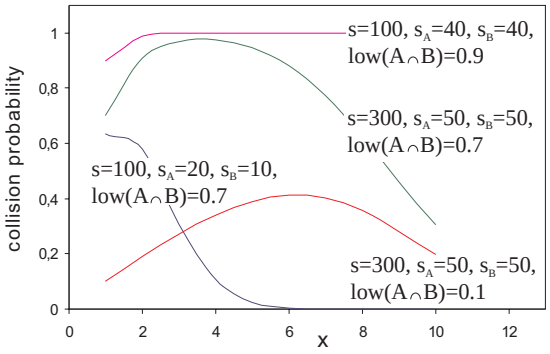



**Figure 3.8:** Usually, $\left(1-(1-lb(A\cap B))^x\right)\cdot Pr[n_{col}(A\cap B)\geq x]$ is maximal for $x\leq 10$.

In the case $x=1$, this equation can be simplified yielding

$$\begin{aligned} Pr[\#\text{ filled bins } \geq 1] &= 1-\frac{\binom{u-w}{v}}{\binom{u}{v}} \\ &\overset{v\geq 0}{=} 1-\prod_{i=1}^{v}\left(1-\frac{w}{u-v+i}\right). \end{aligned} \tag{3.8}$$

It is obvious that in Equation 3.8, $u-v$ has to be computed only once. Therefore, $4v+1$ operations are necessary to compute $Pr[\#\text{ filled bins } \geq 1]$ and in the worst case (where $v=8^3\wedge w=8^3$) $4\times 8^3+1=2049$ operations are necessary to calculate $Pr[\#\text{ filled bins } \geq 1]$ for a single pair of nodes. Of course, this would be too expensive during running time. Therefore, we compute a complete lookup table for $Pr[\#\text{ filled bins } \geq x]$, which has to be done only once for our algorithm. Note that $Pr[\#\text{ filled bins } \geq x]$ is completely independent of the type of BV or model.

### 3.5.2 Probability of collision

Until now, for computing a lower bound for $Pr[P(A)\cap P(B)\neq\varnothing]$ (see Equation 3.1) we have only used the probability that at least *one* collision cell exists in $A\cap B$. Although the algorithm achieves very good quality using only that probability, we can improve the lower bound by using the probability that *several* collision cells are in the intersection, i.e., by using $Pr[n_{col}(A\cap B)\geq x]$, $x>1$.
Obviously, $Pr[n_{col}(A\cap B)\geq x]$ decreases as $x$ increases. But the more collision cells (with high probability) in the intersection volume are, the higher the probability is that a collision really takes place in the pair of BVs.

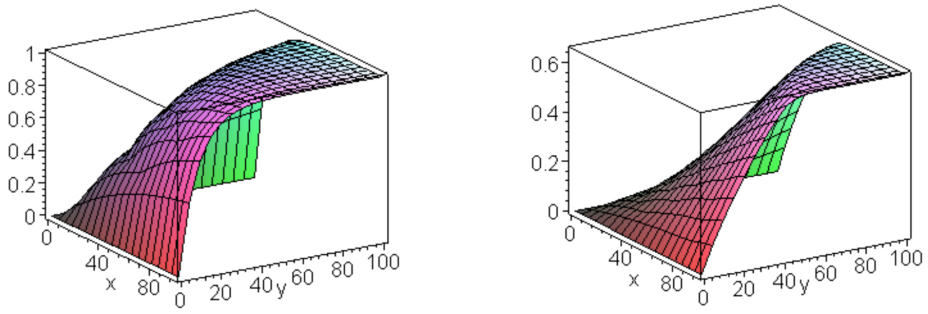



**Figure 3.9:** Probability $Pr[P(A) \cap P(B) \neq \varnothing]$ for a fixed $s$ (=300). Here, $x = s_A$ and $y = s_B$. Left: $lb(A \cap B) = 0.5$, right: $lb(A \cap B) = 0.1$.

Let a partitioning of $A \cap B$ be given. Then, a lower bound for the probability $Pr[P(A) \cap P(B) \neq \varnothing]$ can be computed by

$$Pr[P(A) \cap P(B) \neq \varnothing] \geq \max_{x \leq \min\{s_A, s_B\}} \left\{ Pr[n_{col}(A \cap B) \geq x] \cdot \left(1 - (1 - lb(A \cap B))^x\right) \right\} \quad (3.9)$$

because $\left(1 - (1 - lb(A \cap B))^x\right)$ denotes a lower bound for the probability that in at least one of the $x$ collision cells a collision takes place. Note that, if we use the approximation shown in Observation 1, this is not a lower bound any longer, but *only* a good estimation of it.
In practice, it is sufficient to evaluate Equation 3.9 for small $x$, because for realistic values of $s, s_A, s_B$, and $lb(A \cap B)$, it assumes the maximum at a small $x$ (see Figure 3.8). Consequently, we bound $x$ by a small number (e.g., 10) in Equation 3.9.
To give an overview of the behaviour of $Pr[P(A) \cap P(B) \neq \varnothing]$, Figure 3.9 visualizes Equation 3.9 for different $s_A$ and $s_B$ (x is bounded as described above). Summarizing this section, in order to get a better lower bound for the collision probability, $Pr[P(A) \cap P(B) \neq \varnothing]$ can be computed by Equation 3.9 instead of Equation 3.1.

### 3.5.3 Probability of intersection in a cell

It remains to derive $lb(A \cap B)$, which denotes a lower bound for the probability of an intersection in an arbitrary collision cell from the partitioning of $A \cap B$ (see Definition 4).
For most real-world models, we can assume that the curvature of the surfaces is bounded (possibly except in a finite number of curves on the surface). Now

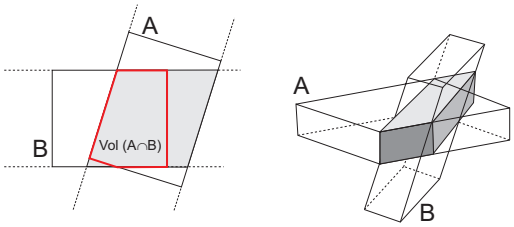



**Figure 3.10:** The intersection volume (highlighted area) could be approximated by taking the three best slabs [BH99].

consider a collision cell $c \subset A \cap B$, i.e., it contains two sets of polygons, $P(A)$ and $P(B)$. Because we are looking for a lower bound, we can only assume that $\text{area}_c(A)$ and $\text{area}_c(B)$ are equal to $\text{maxArea}(c)$.
Suppose the cell $c$ is large compared to the size of the objects. Then, the possible curvature of the surface parts in $P(A)$ and $P(B)$ can lead to convex hulls of $P(A)$ and $P(B)$ that are small with respect to $c$. Therefore, the probability of an intersection is small.
On the other hand, as the traversal of the BVHs reaches lower levels, $c$ becomes small compared to the size of the objects. Then, because of the bounded curvature, $P(A)$ and $P(B)$ can be approximated better and better by two plane polygons. In the extreme, we reach the situation shown in Figure 3.3. Therefore, we estimate the lower bound $lb(A \cap B)$ by

$$lb(A \cap B) \approx \frac{d_A + d_B}{d_{max_A} + d_{max_B}},$$

where $d_A, d_B$ are the depth of node $A, B$ in their respective BV hierarchies, and $d_{max_A}, d_{max_B}$ are the maximum depths. In other words, the larger the depth of the nodes of $A$ and $B$, the smaller the BVs, and the larger is the probability that the polygons in a collision cell intersect.
Note that if we approximate $lb(A \cap B)$ as described, the lower bound given in Equation 3.9 is not a lower bound any longer, but *only* a good estimation of it.

## 3.6 Intersection Volume

Since Equation 3.2 has to be computed once per node pair during the hierarchy traversal, we need a fast way to compute $\text{Vol}(A \cap B)$. However, an exact computation is prohibitively expensive for most BVs (except spheres), even for cubes, because they are not aligned with each other. So we need to resort to approximations. A simple and efficient one is to enclose the BVs by spheres and compute their intersection volume. Unfortunately, this is also a very inaccurate

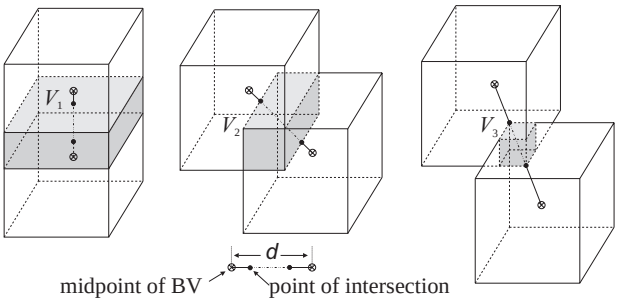



**Figure 3.11:** We estimate the intersection volume for two not necessarily aligned BVs by the maximum of three corresponding aligned cases, $\max\{V_1, V_2, V_3\}$, which is an upper bound. Note that, also for non-cubic BVs, the line through the midpoints of the two BVs has to intersect the vertices and/or edges of the BV of the intersection volume as shown.

one. In the case of DOPs (note that cubes are special 6-DOPs), we could also enclose one of $A$ or $B$ by an aligned DOP efficiently [Zac98]. Another method that works only for boxes (and similarly for DOPs) is to choose three slabs out of the $2 \times 3$ slabs defined by two boxes $A$ and $B$ [BH99]. Figure 3.10 illustrates this idea. The volume of the resulting parallelepiped can be computed by the scalar triple product.
Since the above mentioned methods can all produce fairly gross overestimations, and since the latter two methods are also quite expensive (for our purpose), we propose a different method. The idea is shown in Figure 3.11.
Given two bounding boxes of (nearly) the same size at a specific distance $d$ that are not necessarily aligned with each other. Then, an upper bound of their intersection volume is given by two BVs of the same size with the same distance $d$, that are aligned as one of the 3 cases shown in Figure 3.11. As a consequence, we only need to tentatively compute the intersection volume $V_i$ for each of them. Then, $\max\{V_1, V_2, V_3\}$ is an upper bound of the intersection volume. In the following, let $a$, $b$, and $c$ denote the side lengths of the BVs, where $a \geq b \geq c$. Then

$$
\begin{aligned}
V_1 &= a \cdot b \cdot c \cdot (1 - \frac{d}{\sqrt{a^2}})^1 = (a - d) \cdot b \cdot c \\
V_2 &= a \cdot b \cdot c \cdot (1 - \frac{d}{\sqrt{a^2 + b^2}})^2 \\
V_3 &= a \cdot b \cdot c \cdot (1 - \frac{d}{\sqrt{a^2 + b^2 + c^2}})^3
\end{aligned}
$$

To prove this claim, one has to perform two steps. First of all, assume that the boxes are axis-aligned. Without loss of generality, let one box be centered at the

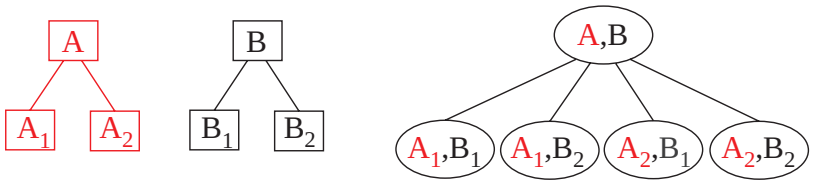



**Figure 3.12:** The BV test tree (BVTT) shows all possible pairs of BVs that might need to be tested for overlap. All hierarchical collision detection algorithms basically perform a traversal through this (conceptual) tree.

origin and the other centered at $P = (x, y, z)$. Then, the intersection volume is $V = (a - x)(b - x)(c - x)$, which has to be maximized under the constraint that $x^2 + y^2 + z^2 = d^2$. Then, one has to show that $V \leq \max\{V_1, V_2, V_3\}$ always holds. In the second step, one has to prove that for a rotated BV the intersection volume is smaller or equal than when aligning that BV while keeping the same distance. If the difference of the size of the two bounding boxes is above a certain threshold, we use the intersection volume of the two bounding spheres as an estimate. In our experience this seems to work well.

## 3.7 Expected Running Time of Hierarchical Collision Detection

In this section, we will derive a model that allows to estimate the number $N_v$, the number of BV overlap tests (see Section 2.2.1). This is equivalent to the number of nodes in the BVTT (see Figure 3.12) that are visited during the traversal. The order and, thus, the exact traversal algorithm are irrelevant.
For the most part of this section, we will deal with 2-dimensional BVHs, for sake of illustration. At the end, we extend these considerations to 3D, which is fairly trivial.
Let $\tilde{N}_v^{(l)}$ denote the number of nodes on level $l$ in the BVTT where a BV pair overlap could possibly occur. We define the (conditional) probability that the BV pair $(A_i^{(l)}, B_j^{(l)})$ on level $l$ overlaps as

$$p^{(l)} := Pr[A_i^{(l)} \cap B_j^{(l)} \neq \varnothing \mid A \cap B \neq \varnothing \wedge o_{ij}^{(l)} > 0],$$

where $(A, B)$ is the parents node of $(A_i^{(l)}, B_j^{(l)})$ in the BVTT. In principle, we would need a different probability for each individual pair of BVs, but we will see that this is not necessary. Let $X_l$ denote the number of nodes we visit on

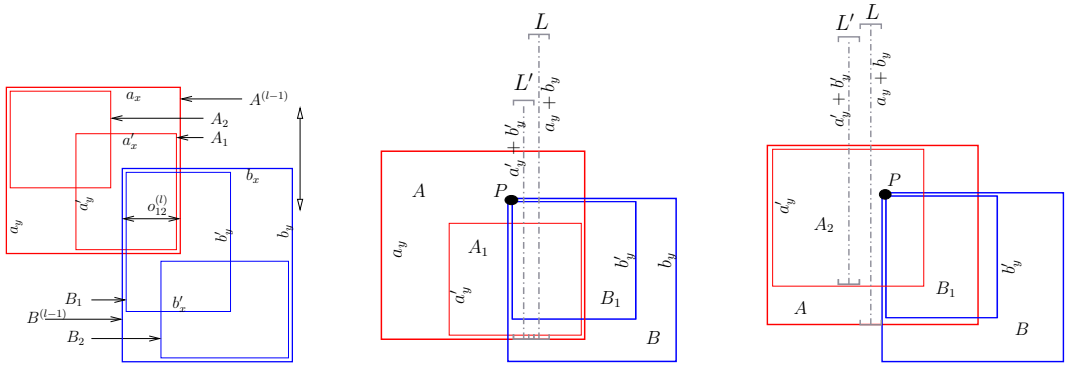



**Figure 3.13:** Left: general configuration of the boxes, assumed throughout our probability derivations. For sake of clarity, boxes are *not* placed flush with each other. Middle: The ratio of the length of segments $L$ and $L'$ equals the probability of $A_1$ overlapping $B_1$. Right: ditto for $p_{21}$.

level $l$ in the BVTT[2]. Then, its expected number $E(X_l)$ can be determined by

$$E(X_l) = \tilde{N}_v^{(l)} \cdot \prod_{i=1}^{l} p^{(l)},$$

where we only assume that the two root BVs $A^{(0)}$ and $B^{(0)}$ overlap (i.e., $p^{(0)} = 1$). Overall, the expected total number of nodes we visit in the BVTT is

$$\tilde{N}_v(n) = E(\sum_{l=1}^{d} X_l) = \sum_{l=1}^{d} \tilde{N}_v^{(l)} \cdot \prod_{i=1}^{l} p^{(l)}, \tag{3.10}$$

where $d = \log_4(n^2) = \lg(n)$ is the depth of the BVTT (equaling the depth of the BVHs).
In the following, we will first derive an estimate for $p^{(l)}$, then for $\tilde{N}_v^{(l)}$.

### Preliminaries

For sake of simplification, we assume that the child boxes of each BV sit in opposite corners within their respective parent boxes, so that $A_1$ and $B_1$ overlap before $A_2$ and $B_1$ do (if at all), see Figure 3.13. In addition, we assume that

[2]The probability space $(\Omega, \Sigma, P)$ is defined as: $\Omega = \{\omega_i^{(l)}\}$, where $\omega_i^{(l)}$ denotes the event, that $i$ nodes of the BVTT at level $l$ are visited; $\Sigma = \mathcal{P}(\Omega)$, the power set of $\Omega$, denotes a $\sigma$-algebra on $\Omega$; the probability measure $P$ is defined as $P \sim Bi(\tilde{N}_v^{(l)}, \prod_{i=1}^{l} p^{(l)})$, because we consider a $\tilde{N}_v^{(l)}$-step Bernoulli experiment ($\prod_{i=1}^{l} p^{(l)}$ denotes the probability that a BV pair on level $l$ overlaps, assuming the two root BVs $A^{(0)}$ and $B^{(0)}$ overlap).

| probability $p^{(l)}$ | $T(n)$ |
|---|---|
| $< 1/4$ | $O(1)$ |
| $1/4$ | $O(\lg n)$ |
| $\sqrt{1/8} \approx 0.35$ | $O(\sqrt{n})$ |
| $1/2$ | $O(n)$ |
| $3/4$ | $O(n^{1.58})$ |
| $1$ | $O(n^2)$ |

**Table 3.1:** Effect of the probability $p^{(l)}$ on the running time of a simultaneous hierarchy traversal.

there is a constant *BV diminishing factor* throughout the hierarchy, i.e.,

$$a'_x = \alpha_x a_x, \qquad a'_y = \alpha_y a_y, \qquad \text{etc.}$$

Only for sake of clarity, we assume that the scale of the boxes is about the same, i.e.,

$$b_x = a_x, \qquad b'_x = a'_x, \qquad \text{etc.}$$

This assumption allows us some nice simplifications in Equation 3.11 and 3.14 as well as the use of a single $\omega$ in Equation 3.13, but it is not necessary at all.

### Probability of Overlap

Similarly to [Zac02], we could estimate $p^{(l)}$ by the volume of Minkowski sums. However, in that model all positions of $B$ with respect to $A$ would be equally likely, which is not true in practice.
Instead we assume, without loss of generality, that the root box $B^{(0)}$ penetrates $A^{(0)}$ by some known distance $o_{ij}^{(0)} := \delta$ along the x axis from the right. Our analysis considers all configurations as depicted in Figure 3.13, where $\delta$ is fixed but $B$ is free to move vertically, under the condition that $A$ and $B$ overlap.
First, let us consider $p_{11}$ (in the following, we will drop the superscript $(l)$) (see Figure 3.13). By precondition, $A$ overlaps $B$, so the point $P$ (defined as the upper left (common) corner of $B$ and $B_1$) must be on a certain segment $L$, which has the same x component as the point $P$ and which length is defined by $a_y + b_y$. (The BV $B$ and, thus, $P$ can be chosen arbitrarily, under the condition that $A$ and $B$ still overlap. $L$ would be shifted accordingly, but its length would be the same.) Note that for sake of illustration, segment $L$ has been shifted slightly to the right from its true position in Figure 3.13 (center). If we restrict the position of $B$ such that $A_1$ and $B_1$ overlap, too, then $P$ remains on segment

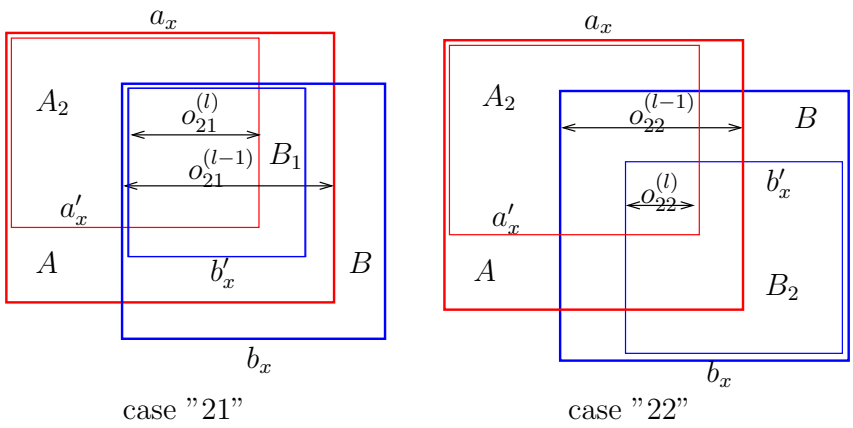



**Figure 3.14:** Denotations for computing $o_{ij}^{(l)}$ for a child pair. The case "12" is symmetric to "21", and the case "11" is trivial. Here, $\omega = a_x^{(0)}\alpha_x^0(1-\alpha_x) = a_x^{(0)}(1-\alpha_x)$.

$L'$. Thus,

$$p_{11} = \frac{\text{Length}(L')}{\text{Length}(L)} = \frac{a'_y + b'_y}{a_y + b_y} = \alpha_y. \tag{3.11}$$

Next, let us consider $p_{21}$ (see Figure 3.13; for sake of clarity, we re-use some symbols, such as $a'_x$). For the moment, let us assume $o_{ij}^{(l)} > 0$; in Section 3.7 we estimate the likelihood of that condition. Analogously as above, $P$ must be anywhere on segment $L'$, so $p_{21} = \alpha_y = p_{11}$ and, by symmetry, $p_{12} = p_{21}$. Very similarly, we get $p_{22} = \alpha_y$. At this point, we have shown that $p^{(l)} \equiv \alpha_y$ in our model.

### Positive x-Overlap

We can trivially bound $\tilde{N}_v^{(l)}$, the number of nodes whose x-overlap is $> 0$, by $4^l$. Plugging this in Equation 3.10 yields

$$\begin{aligned} \tilde{N}_v(n) &\leq \sum_{l=1}^{d} 4^l \cdot \alpha_y^l = \frac{(4\alpha_y)^{d+1} - 1}{4\alpha_y - 1} \qquad (4\alpha_y \neq 1) \\ &\in O((4\alpha_y)^d) \;=\; O(n^{\lg 4\alpha_y}). \end{aligned} \tag{3.12}$$

The corresponding running time for different $p^{(l)}$ can be found in Table 3.1. For $p^{(l)} > 1/4$, the running time is in $O(n^c)$, $0 < c \leq 2$.
Clearly, $\tilde{N}_v^{(l)} \leq 4^l$ is not a tight bound. So, in the following, we derive a better one.

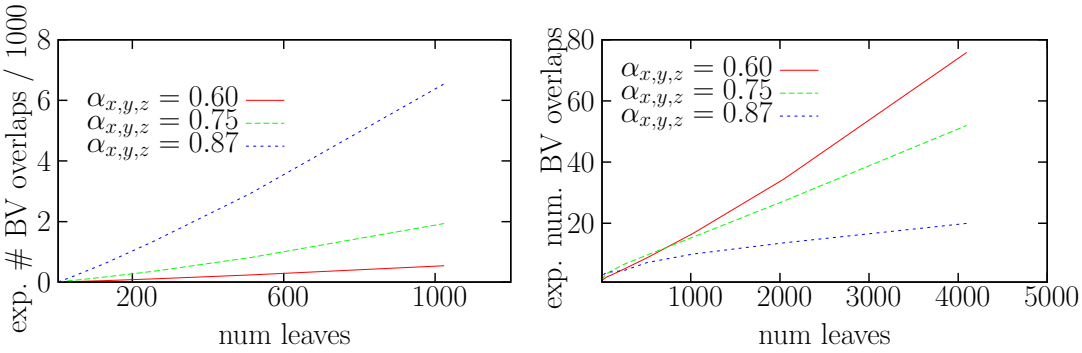



**Figure 3.15:** Overall expected number of overlapping BV pairs $\tilde{N}_v(n)$ (= nodes in the BVTT), which is proportional to the expected running time. Left: $\delta = 0.4$; right: $\delta = 0.15$ (note that we chose $\alpha_x = \alpha_y = \alpha_z$).

Each pair $(A_i^{(l)}, B_j^{(l)})$, when projected onto the x axis, has a specific amount of overlap, $o_{ij}^{(l)}$. As mentioned before, only pairs with $o_{ij}^{(l)} > 0$ can possibly intersect (remember that $\tilde{N}_v^{(l)}$ denotes the number of these pairs).
Given $o_{ij}^{(l-1)}$, the x-overlap of $(A^{(l-1)}, B^{(l-1)})$, we can easily compute $o_{ij}^{(l)}$ for each case (see Figure 3.14):

$$\begin{aligned} &\text{case } ij = 11: \quad o_{11}^{(l)} = o_{11}^{(l-1)}, \\ &\text{case } ij = 21: \quad o_{21}^{(l)} = o_{21}^{(l-1)} - \omega, \\ &\text{case } ij = 12: \quad o_{12}^{(l)} = o_{12}^{(l-1)} - \omega, \\ &\text{case } ij = 22: \quad o_{22}^{(l)} = o_{22}^{(l-1)} - 2\omega. \end{aligned} \tag{3.13}$$

with $\omega = a_x^{(0)} \alpha_x^{l-1}(1 - \alpha_x)$ and $a_x^{(0)}$ = the extent of the root BV.
So, in order to determine whether or not a pair of boxes $A_i^{(l)}, B_j^{(l)}$ on level $l$ could possibly overlap, we need to determine whether its $o_{ij}^{(l)} > 0$. We can do this by starting at the root of the BVTT, initializing $o_{ij}^{(0)} := \delta$, the overlap of the root boxes when projected onto the x axis. Then we proceed down through the BVTT along the path to $A_i^{(l)}, B_j^{(l)}$, decrementing $o_{ij}^{(0)}$ with each step by 0, $\omega$, or $2\omega$ according to Equation 3.13, depending on which child we go into.
Clearly, $\tilde{N}_v^{(l)}$ depends on the parameters $\alpha_x$ and $\delta$. So, for a range of parameter values, we could tabulate $\tilde{N}_v^{(l)}$, and then compute a much better estimate for the number of BV overlap tests using Equation 3.10.

### 3D Collision Detection

As mentioned, our considerations can simply be extended to 3D. Then, $L$ and $L'$ of Equation 3.11 are not line segments any longer, but 2D rectangles in 3D

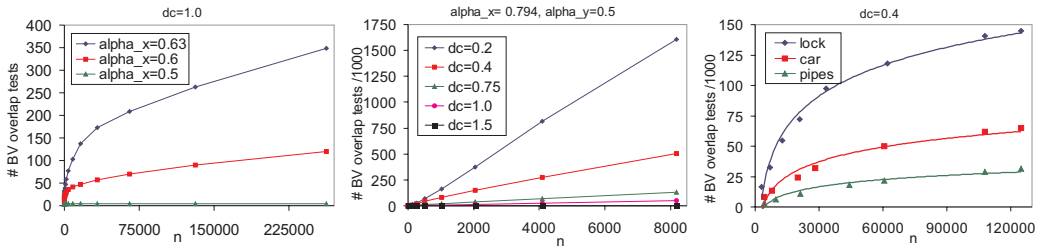



**Figure 3.16:** Number of BV overlap tests depending on the number $n$ of leaves (left, center: artificial BVHs, right: BVHs for models shown in Figure 3.18). Left: the distance $dc$ between centers of the root BVs is set to 1. Center: if $\alpha_x$ is chosen so that $\alpha_y = 0.5$, the number of BV overlap tests is linear in $n$, independent of the distance $dc$. Right: plots for our models at distance $dc = 0.4$.

which are perpendicular to the x axis lying in the y/z plane. The area of $L'$ can be determined by $(a'_y + b'_y)(a'_z + b'_z)$ and the area of $L$ by $(a_y + b_y)(a_z + b_z)$. Thus,

$$p_{11} = \frac{\text{area}(L')}{\text{area}(L)} = \frac{(a'_y + b'_y)(a'_z + b'_z)}{(a_y + b_y)(a_z + b_z)} = \frac{4a'_y a'_z}{4a_y a_z} = \alpha_y \alpha_z. \tag{3.14}$$

The other probabilities $p_{ij}$ can be determined analogously as above, so that $p_{11} = p_{12} = p_{21} = p_{22} = \alpha_y \alpha_z$. Then, we can estimate the number of BV overlap tests by

$$\tilde{N}_v(n) = \sum_{l=1}^{d} \tilde{N}_v^{(l)} \cdot \alpha_y^l \cdot \alpha_z^l, \tag{3.15}$$

where $d = \log_4(n^2) = \lg(n)$. Figure 3.15 shows this number for different parameters $\alpha_{x,y,z}$ (that means $\alpha_x, \alpha_y, \alpha_z$) and $\delta$.
Note that Table 3.1 is still valid in the 3D case.

### Experimental Support

We implemented (using C++) the traversal algorithm shown in Figure 1 using AABBs as BVs. As we are only interested in the number of visited nodes in the BVTT, we switched off the intersection tests at the leaf nodes.
We used a set of CAD objects, each of them with varying complexities with respect to the number of polygons (Figure 3.18). Benchmarking is performed by the procedure of [Zac02], which can compute the average number of overlapping BVs for a distance $dc$ between two identical objects (for more detail see Section 3.8.1).

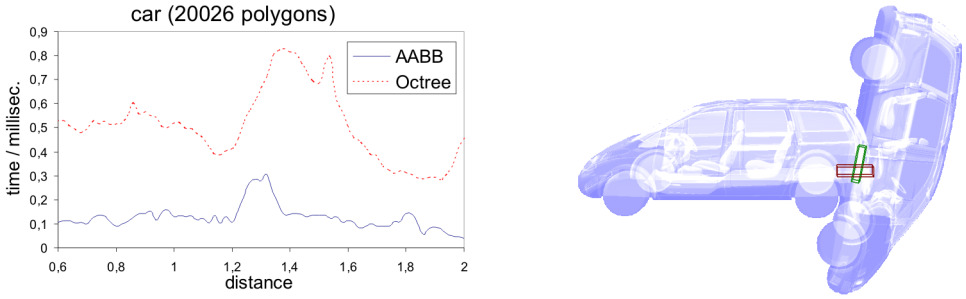



**Figure 3.17:** Left: comparison of our ADB trees based on an octree and on a binary AABB tree. Here, the results for the car body with 20,000 polygons are shown, but we obtained similar ones for all other objects of our suite. Right: A snapshot of the benchmarking procedure (objects courtesy of VW). The two boxes show one pair of BVs that contain at least one collision cell with high probability.

Figure 3.16 (right) shows the number of BV overlap tests for our models depending on their complexities for a fixed distance $dc = 0.4$. Clearly, the average number of BV overlap tests behaves logarithmically for all our models.
We also examined, whether $p^{(l)}$ can be estimated by the volume of Minkowski sums where all positions of $B$ with respect to $A$ are equally likely [Zac02]. In this case,

$$p^{(l)} = \alpha_x \alpha_y \alpha_z.$$

To measure the the number of BV overlap tests, we used artificial BVHs where we can adjust $\alpha_{x,y,z}$, and thus $p^{(l)}$. There, the child pairs of each node are placed in its opposite corners. Fig. 3.16 (left) shows the number of BV overlap tests depending on $n$ for different choices of $\alpha_{x,y,z}$. If $\alpha_{x,y,z} = 0.5$, then $p^{(l)} = 0.125$ and, as also shown by Table 3.1, the running time seems to be constant. Moreover, the plot shows that, if $\alpha_{x,y,z} = 0.63$ so that $p^{(l)} = 0.25$, the number behaves logarithmically.
If $\alpha_{x,y,z}$ is chosen so that $p^{(l)} = 0.5$, then the number of overlaps behaves linear as one can see in Fig. 3.16 (center). Overall, $p^{(l)}$ can be approximated by the volume of Minkowski sums.

## 3.8 Results

Because our approach is applicable to most hierarchical BVHs, we decided to implement two basic data structures, namely an octree and a binary AABB tree, that are used in many VR applications and that can easily be turned into

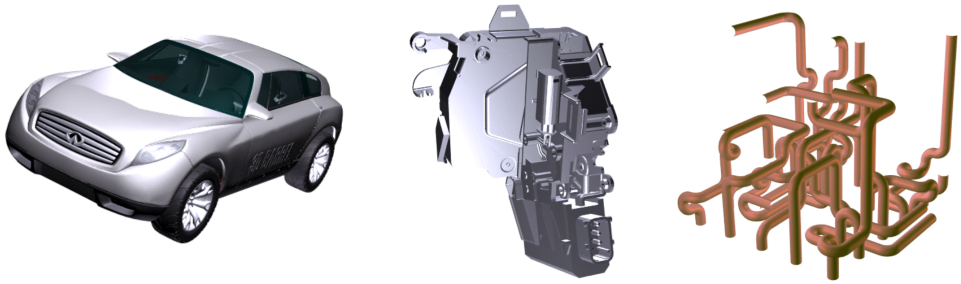


**Figure 3.18:** Some models of our test suite: Infinity Triant (www.3dbarrel.com), lock (courtesy of BMW) and pipes.

ADB trees. The construction heuristic of the AABB tree is the same as that used for the restricted boxtree [Zac02], so that the corresponding ADB tree can be built in time $O(n \log n)$ (note that the possible collision cells can be determined in time $O(n \log n)$ as shown in Section 3.3).
Figure 3.17 shows the timing results of a comparison between our ADB trees based on the octree and the AABB tree. Obviously, the AABB tree performs better by a factor $> 3$ and we obtained similar results with all other models. Therefore, all the following benchmarks were performed using a binary ADB tree based on AABBs.
We implemented our new algorithm in C++. To date, the implementation is not fully optimized. Note that we have not implemented the idea of inner boundaries as proposed in Section 3.4.2. In the following, all results were obtained on a 2.4 GHz Pentium-IV with 1 GB main memory.

### 3.8.1 Benchmark Scenario

For timing the performance and measuring the quality of our algorithm, we used a set of CAD objects, each of them with varying complexities (see Figure 3.18). Benchmarking is performed using the procedure proposed in [Zac02], which computes average collision detection times for a range of distances between two identical objects.
Two identical objects are positioned at a specific distance from each other. The distance is measured between the centers of the BVs and both objects are scaled uniformly so that they fit in a cube of size $2^3$. Then, one of them performs a full tumbling turn about the $z$ and $x$ axes in a fixed, large number of small steps (5000). With each step, a collision query is done, and the average collision detection time for a complete revolution at that distance is computed. Then, the distance between the objects is decreased, and a new average collision

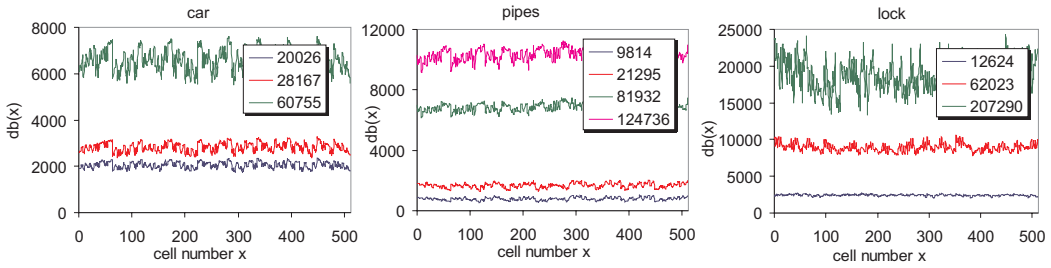



**Figure 3.19:** Histogram of the number of times, $db(x)$, cell $x$ occurred as a possible collision cell in the ADB tree. The number in parentheses in the legend gives the number of polygons.

detection time is determined.
Figure 3.17 (right) shows a snapshot of this benchmarking procedure, where one pair of BVs has been highlighted which contains at least one collision cell with high probability.

### 3.8.2 Distribution of Possible Collision Cells

One premise of our average-case approach is the assumption that the probability of being a possible collision cell is evenly distributed among all cells of the partitioning (see Section 3.1). Here, we give some empirical results suggesting that in practical cases this assumption is actually valid.
The basic idea is to show that the possible collision cells are evenly distributed throughout the hierarchy. Assuming each cell has the same probability of lying inside the intersection volume, then the probability of being a possible collision cell is also evenly distributed among all cells of the partitioning.
Given an ADB tree, we can identify corresponding cells of all nodes by a number $x \in \{1, \ldots, 512\}$. Thus, for all $x$ we can count for all nodes how often that cell is a possible collision cell throughout the tree (this number $db(x) \leq n$).
Figure 3.19 shows the distribution of the possible collision cells for different models with varying complexities. We have obtained similar plots for all other models of our benchmark suite.
Obviously, our assumption seems to be met by almost all objects occurring in practice. An exception might be the door-lock model with 207 290 polygons, where $\max\{db(x)\}$ and $\min\{db(x)\}$ are about 30% larger and smaller than the average.
Note that the distribution of possible collision cells inside the intersection vol-

| BV hierarchy | Bytes |
|---|---|
| ADB tree based on AABBs | 36 |
| sphere tree | 16 |
| AABB tree | 28 |
| OBB tree | 64 |
| 24-DOP tree | 100 |

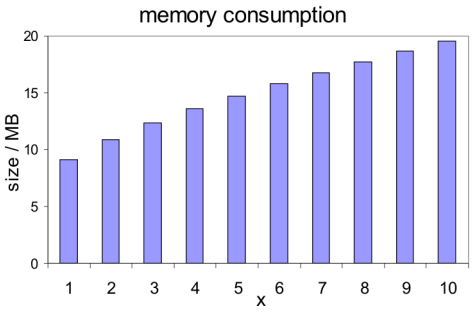



**Figure 3.20:** Left: the table compares the amount of memory per node for our ADB tree with some traditional ones. Right: memory usage of our ten lookup tables.

ume depends on the trajectories of the objects and their rotations which are not considered here. Therefore, these experiments are only a hint that our assumption can be valid for most practical cases. In Section 3.10 we propose some ideas for future work on how to analyze the distribution in a more general way.

### 3.8.3 Preprocessing

#### Memory requirements

Figure 3.20 (left) summarizes the number of bytes per node for different BVHs. Note that we do not need to store any polygons or pointers to polygons in leaf nodes. Therefore, we need exactly the same number of bytes for each node in our hierarchy. We use 24 bytes for storing two vectors that define the BV, 4 bytes for a pointer to child nodes, 4 bytes for storing possible collision cells, and 4 bytes for storing the volume of the BV.

#### Partitioning of BVs

As mentioned in Section 3.1, the quality of the collision detection depends, to some extent, on the number of cells a BV is partitioned into.
The finer the partitioning, the more possible collision cells are stored at each node and the larger $s_A$ and $s_B$ are, but also the larger $s$ is. Nevertheless, the probability of an intersection increases if the partitioning gets finer (assuming the ratio of $s$ and $s_{A,B}$ remains constant). The situation is illustrated in Figure 3.21.
Our experiments have shown that $8^3$ cells per node is a good choice: if the nodes are partitioned into remarkably more or fewer cells, the collision quality (in the sense of the error rate) decreases.

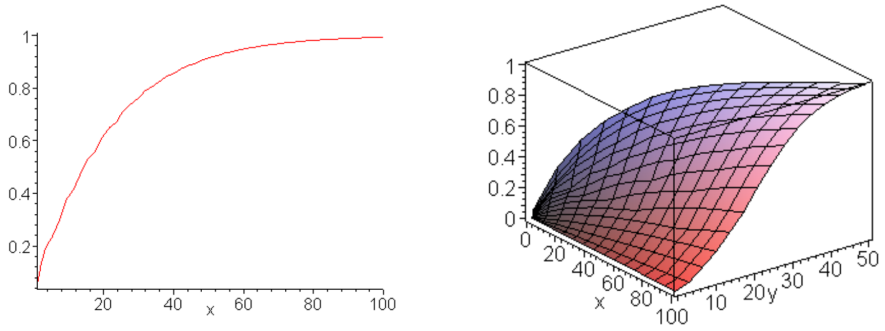



**Figure 3.21:** Probability $p = Pr[P(A) \cap P(B) \neq \varnothing]$ of an intersection depending on the number $x \leq 100$ of cells lying in the intersection volume. In the left plot, $s_A$ and $s_B$ are chosen so that $s_A = s_B = \frac{x}{2}$. The right plot shows $p$, also depending on $y = s_A = s_B \leq \frac{x}{2}$. Here, we have chosen $lb(A \cap B) = 0.25$.

An other reason for restricting the number of cells by $8^3$ is that the finer the subdivision into cells, the larger the lookup tables are (and the higher the memory usage is), because the values $s$, $s_A$ and $s_B$ directly depend on the partitioning.
The number of possible collision cells can be computed by our algorithms shown in Section 3.3. For our models, the exact algorithm needs about 2 minutes on average for the computation of possible collision cells for one complete BVH, while the approximate algorithm only needs less than 2 seconds for our most complex model of 200,000 polygons. All our measurements were performed using the exact algorithm, but the approximate one reduces the quality of the collision detection only by about 0.2% points on average.

### Lookup Tables

As mentioned, we need a lookup table for the probabilities $Pr[\#\text{ filled bins } \geq x]$ with $u, v, w \in \{1, \ldots, 512\}$ and $x \leq 10$ (see Section 3.5.1). By exploiting the fact that $Pr[\#\text{ filled bins } \geq x]$ for $u = u', v = v', w = w'$ is equal to $Pr[\#\text{ filled bins } \geq x]$ for $u = u', v = w', w = v'$, we can reduce memory usage of that lookup table by a factor 2. Still, for each $x$, the lookup table would contain $512 \cdot \sum_{i=1}^{512} i$ entries amounting to about 256 MB. Fortunately, we can reduce the number of entries significantly by exploiting the monotonicity of $Pr[\#\text{ filled bins } \geq x]$ in the variables $v$ and $w$. If the probability for $u = u', v = v', w = w'$ is close to 1 (e.g., $\geq 1 - 0.9 \cdot 10^{-5}$), then we do not compute the probabilities for $u = u', v = v', w \geq w' + 1$; instead, we continue at $u = u', v = v' + 1, w = v' + 1$. On average, this reduces the number of probabilities for one lookup table by a factor of 17. Figure 3.20 (right) gives an overview of

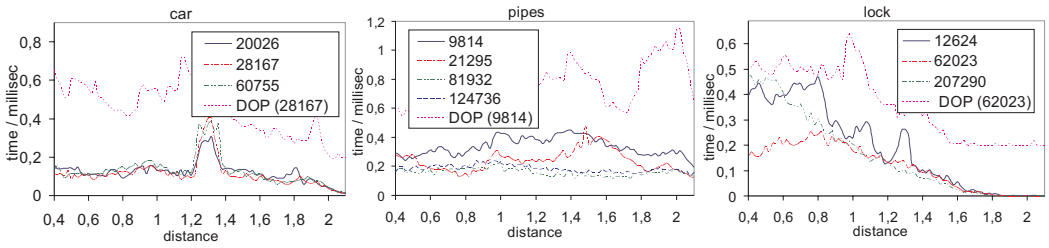



**Figure 3.22:** Timings for different models and different polygon counts ($k_{min} = 10$ and $p_{min} = 0.99$). Also, a running time comparison to a DOP tree is shown.

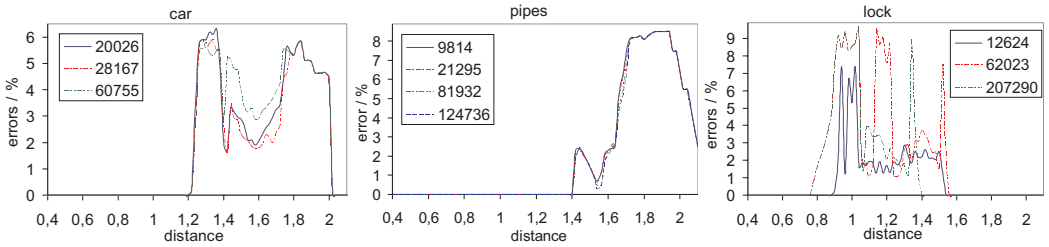



**Figure 3.23:** Error rates corresponding to the timings in Figure 3.22.

memory usage for storing the lookup tables.
Note that when computing $Pr[\#\text{ filled bins } \geq x]$, the binomial coefficients can become very large. Therefore, in order to compute the probabilities as accurately as possible, one should use arbitrary precision for numeric calculations. Using Maple, the time for computing all ten tables took about 93 hours. As mentioned earlier, these lookup tables are independent of the models, so that they have to be computed only once and can be stored in a file.

## 3.8.4 Performance and Quality

### Time and Quality versus Complexity

Each plot in Figure 3.22 shows the running time for a model of varying complexity (the legend gives the number of polygons per object). In most cases, the running time is fairly independent of the complexity.
There are exceptions, for instance, the pipes with 9,814 polygons even take slightly longer than the pipes modelled with 124,736 polygons. We conjecture

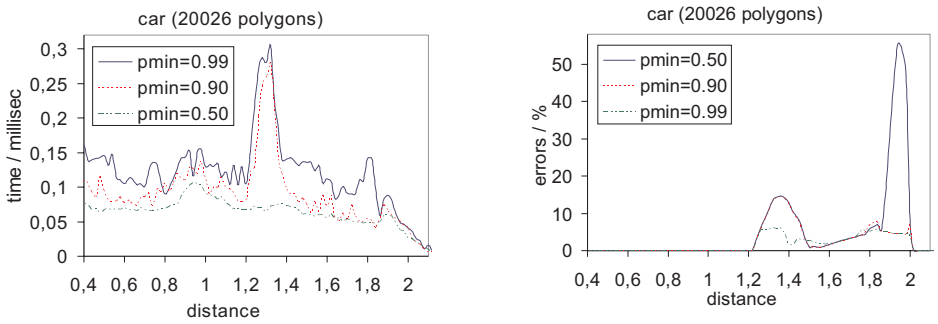



**Figure 3.24:** Running time and error comparisons for the car body with 20,000 polygons for different $p_{min}$ ($k_{min} = 10$).

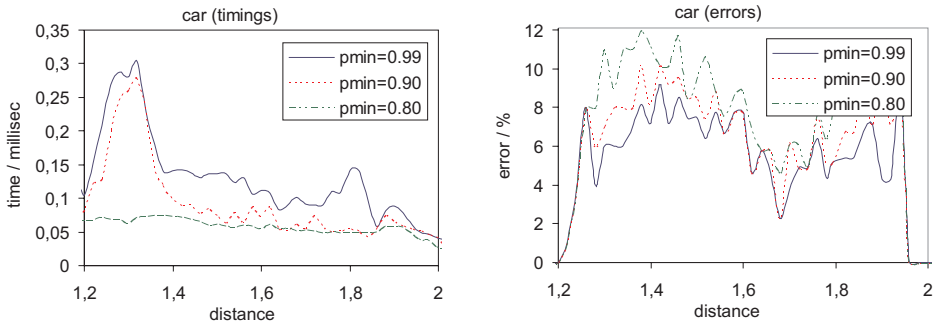



**Figure 3.25:** Controllable balancing between the running time and the accuracy of collision detection. However, we cannot guarantee an upper bound for the error rate because the assumption of uniformly distributed polygons is not always met in practice.

that this is caused by larger polygons in the coarser resolution of the model which allows the construction heuristic fewer possibilities to construct optimal BVHs.

Figure 3.23 shows the error rates corresponding to the timings in Figure 3.22. Here, the error is defined as the percentage of wrong detections. To measure them, we compared our results with an exact approach. Only collision tests are considered where at least the outer BVs, which enclose the whole objects, intersect. Apparently, the error rates are always relatively low and mostly independent of the complexities: on average, only 1.89% (car), 1.54% (door lock), and 2.10% (pipes) wrong collisions are reported if the objects have a distance between 0.4 and 2.1, and about 3.19% (car), 1.71% (door lock), and 3.15% (pipes) wrong collisions are reported for distances between 1 and 2.

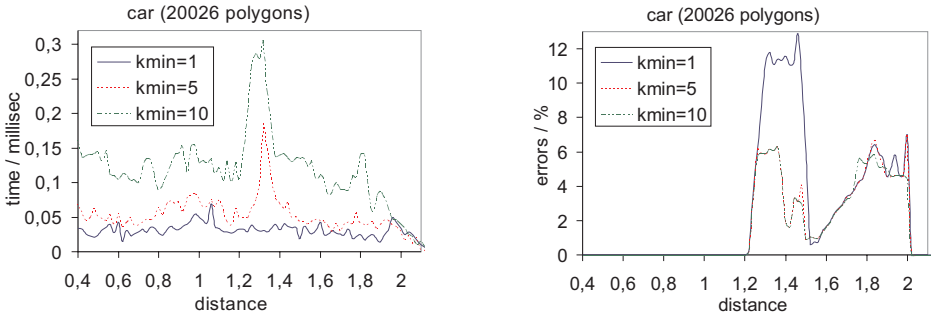



**Figure 3.26:** Timings and error comparisons for different $k_{min}$ ($p_{min} = 0.99$).

### Time versus Quality

In this section, we examine how the running time depends on the quality of the collision detection.

As mentioned in Section 3.1, the running time and the quality can be influenced by the values of $p_{min}$ and $k_{min}$ (see Algorithm 2): the smaller $p_{min}$ or $k_{min}$ is, the shorter the running time is and, usually, the more errors are made.

Figure 3.24 on the left shows the correlation between the running time and $p_{min}$ (car, 20026 polygons). The corresponding error rates are shown on the right. Obviously, as $p_{min}$ increases, the error rate decreases. There are a few exceptions, where more errors are made when using a larger $p_{min}$. We conjecture that this is caused by pairs of BVs where corresponding polygons (that do intersect) have a low probability of intersection. For $p_{min} = 0.9$ and $p_{min} = 0.99$ the errors differ only by less than 2% points while for $p_{min} = 0.9$ and $p_{min} = 0.5$ the errors differ by about 5% points on average (object distances between 1.2 and 2).

In Figure 3.26 (left), the timings and the corresponding errors for different $k_{min}$ (the number of pairs of collision nodes that have to be found before the traversal stops) are compared (car, 20026 polygons).

Only about 0.2% points fewer errors are made if $k_{min}$ increases from 5 to 10, while 2% points fewer errors occur if $k_{min}$ is changed from 1 to 5 (object distances between 1.2 and 2). Comparing the timings for $k_{min} = 5$ and $k_{min} = 10$, it is questionable whether an increase in accuracy of 0.2% points justifies a decrease in speed by a factor $\approx 2.3$.

### Performance comparison

A running time comparison between our approach and a DOP tree implementation, where no probabilities are utilized, can be found in Figure 3.22. We implemented the DOP tree with the same care as for our new approach. The

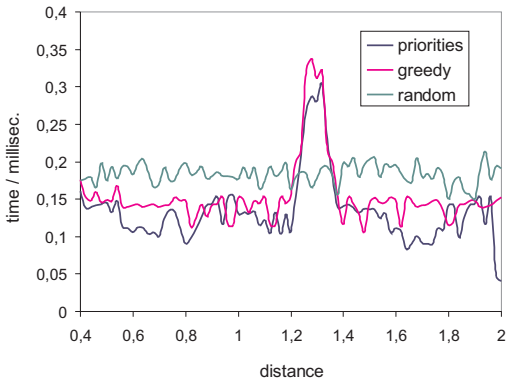



**Figure 3.27:** On average, our traversal strategy (priorities) has a running time of 0.13 millisec. The greedy traversal needs about 0.16 and the random strategy about 0.18 millisec.

running time for the DOP tree is only shown for a single resolution at which the highest performance was achieved using the DOP tree approach. As you can notice, our algorithm is always remarkably faster, e.g., in the case of the car body, our new algorithm is $\approx 3$ times faster on average ($k_{min} = 10, p_{min} = 0.99$) and $> 6$ times faster, if the error rate is permitted to increase by only 0.2% points ($k_{min} = 5, p_{min} = 0.99$).

#### Traversal-Criterion

We proposed to give priority to those pairs of BVs with the highest probability of intersection. We compared this traversal criterion with two other strategies. The *random strategy* always takes a random pair during the traversal, while the *greedy strategy* gives priority to those pairs with the highest intersection volume.
The results for the car model can be found in Figure 3.27. Our strategy is about 20% faster than the greedy strategy and about 28% faster than the random strategy.

## 3.9 Summary and Discussion

We have presented a general method to turn a conventional hierarchical collision detection algorithm into one that uses probability estimations to decrease the quality of collision detection in a controlled way. To our knowledge, this is the first such approach to this problem.
Our algorithm can be utilized to increase the perceived quality of simulations

and interactions by increasing the performance without noticeably decreasing the accuracy. More precisely, our novel framework can be useful in several ways:

- It allows applications that do not need precise collision detection to take advantage of that opportunity by specifying a desired *quality* threshold, thus decreasing the collision detection time significantly.

- It allows a scheduler to *interrupt* the collision detection while still allowing the collision detection to make a better effort than in the traditional traversal schemes *and* return a kind of measure of *confidence* in the result.

Note that in both cases, the application can still obtain meaningful contact information in order to handle the collision.
Our approach is made possible by augmenting traditional BV hierarchies with just a few additional parameters per node, which are utilized during traversal to efficiently compute the probability of a collision occurring among the polygons of a pair of BVs. These probabilities are then used as priorities to direct the traversal into those parts of the BV trees with higher probability.
We implemented our ADB trees (average-distribution trees) based on AABBs (that can be constructed in time $O(n \log n)$ and present performance measurements and comparisons with a fast traditional algorithm, namely the DOP tree. The results show a speedup by about a factor 3 to 6 with only about 4% error on average. Furthermore, the error rates are almost independent of the number of polygons.
We compared our traversal strategies with two others (greedy and random) and found out that our strategy performs up to 30% better on average.
Moreover, we have presented an analysis for simultaneous hierarchical BV traversals that provides a better understanding of the performance of hierarchical collision detection that has been observed in the past. Our analysis is independent of the order of the traversal. We performed several experiments to support the correctness of our model, and have shown that the running time behaves logarithmically for real world models, even for a large overlap between the root BVs.

## 3.10 Future Work

We believe that our algorithms, data structures and techniques presented in this section open up a number of additional avenues for future work.
Until now, our approach has assumed that the polygons are evenly distributed within the BVs. This assumption becomes more and more incorrect, the deeper

the nodes are stored in the hierarchy. Then, the locus of a polygon depends on the other polygons because of their restricted curvature. As a consequence, the polygons are not independently and randomly distributed in space. Therefore, one should improve our approach so that it does not need this assumption any longer.

The technique of "machine learning" could possibly be used to solve the problem. Its idea is to learn from examples a (functional) context. That means that the assumption of a uniform distribution can incrementally be adapted for two objects and their special collision behavior. This technique allows us to determine a probability distribution for some training examples $x_1, ..., x_m \in X$ that corresponds to the distribution of $x_1, ..., x_m$. For our problem, $x_i$ could denote the event of a collision occurring in cell $i$. Then, during the collision detection, it should be determined which cell has what probability of collision, and to use this information in order to decide whether the assumption of a uniform distribution was suitable.

Of course, the resulting dependencies should be updated in the corresponding BVs. That means, if the distribution within a leaf changes, how the distribution changes in the parent nodes should be examined. Moreover, we should think about an efficient description for those uneven distributions.

Another very interesting point that should be examined in the future, is to follow up our idea of 2-AABBs, 2-OBBs and 2-DOPs. While for 2-AABBs the intersection volume can easily be determined, an efficient way to do this for the other types of BVs should be considered. We believe, provided an efficient overlap test exists, that these BVs are very well suited for all other hierarchical collision detection approaches and perform better than the classical ones. As a *side effect*, the error rate of our approach should clearly be improved.

Moreover, adapting our approach to simple BVHs, in particular DOP trees and Restricted Boxtrees, would be very promising. In addition, an examination can be made to see if it can be applied to non-hierarchical data structures for deformable collision detection.

A very useful extension of our new algorithm would be the modeling of contacts and an estimation of the separation distance (if non-colliding) or penetration depth (in the case of collision). Both are extremely helpful for speeding up physically-based simulations, and they are even essential for force-feedback.

We believe that our approach can easily be adapted to point-based geometry. Assuming the models can be approximated by surfels, it should be quite easy to determine (possible) collision cells (the surfel radius can be estimated as proposed in the next section). Moreover, cell sizes adapted to the local sampling density could improve the quality of the collision detection process.

There are many areas in computer graphics that utilize BVHs, such as ray

tracing, occlusion culling, or shadow rendering. Thus, a natural question is whether an average-case approach can be applied there too.
Several existing methods for hierarchical collision detection may benefit from our analysis and our model for estimating the expected running time. Especially in time-critical environments or real-time applications it could be very helpful to predict the running-time of the collision detection process only with the help of two parameters that can be determined on the fly.
Furthermore, it would be very interesting to apply our technique to other areas, such as ray tracing. And, finally, we believe one could exploit these ideas to obtain better bounding volume hierarchies.

# 4 Point Cloud Surfaces using Geometric Proximity Graphs

A point cloud sampled from a 3D model can be used to approximate and reconstruct the original surface. So, a point cloud can be seen as a representation of an implicit function $f(\mathbf{x})$, whose zero set $S$

$$S = \left\{\mathbf{x} | f(\mathbf{x}) = 0, \mathbf{x} \in \mathbb{R}^3\right\}$$

approximates the original surface.
The weighted least squares (WLS) approach [LS81, Lev98] for defining such implicit surfaces, that was originally introduced by McLain [McL74] in the context of contouring, is quite attractive and can be evaluated very quickly. As we are aiming at interactive collision detection between point clouds (without an explicit reconstruction of the surface), this surface definition seems to be very suitable for our problem. Moreover, in contrast to parametric representations, surfaces with highly complex topology can be represented easily and the global consistency of the surface is guaranteed by construction [PKKG03]. Extreme geometric deformations and even topology changes can be achieved by simply modifying the weight coefficients of the respective basis functions [PKKG03].
However, there are some problems, like

- the distance in the weighting kernel is not adapted to the "topology" of the original surface (Section 4.1.2),

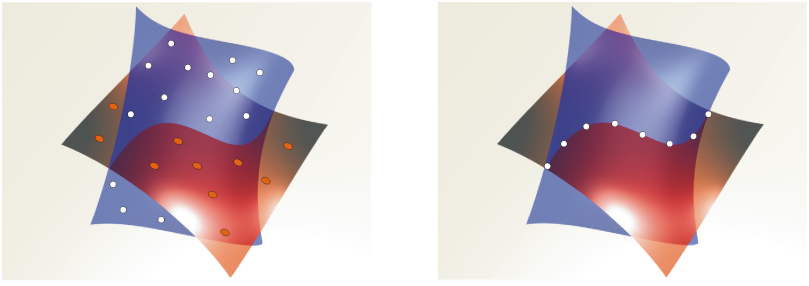


**Figure 4.1:** Left: two implicit surfaces that are defined over point clouds. Right: (discrete) points lying on the intersection curve.

- the (local) kernel bandwidth determination (Section 4.2.5), and
- the boundary detection (Section 4.2.6)

which we explain in detail in the following sections.
As a consequence, it can suffer from artifacts in the surface. Therefore, we propose a new definition of a surface over a possibly noisy point cloud. It builds on an implicit function which is defined using weighted least squares regression. For weighting the points, we propose to use geodesic distances along the surface so that artificial zero sets can be avoided. The basic idea for approximating geodesic distances is to utilize a geometric proximity graph which also allows for easy bandwidth and boundary detection.
Our techniques can be applied to other surface definitions as well, such as Levin's popular projection operator [Lev03] which is based on moving least squares approximation. Note that we are not concerned with actually rendering the implicit surface. This can be done with ray tracing [AA04a], sphere tracing [OBA+03, Har96], or tesselation [Blo94].
This section is organized as follows. Section 4.1 introduces the WLS surface definition and its problems with the Euclidean kernel. In the next Section, our new surface definition based on geometric proximity graphs is explained. We examine the Delaunay graph and the sphere-of-influence graph in detail, and propose several extensions for the sphere-of-influence graph to adapt it optimally to our problem. Moreover, our new automatic bandwidth and boundary detections are described. Section 4.3 gives an overview of the complexity of our data structures and the time for evaluating the implicit function using our new geodesic kernel. We conclude the section with a short summary and discussion, followed by some ideas for future work.

## 4.1 Implicit Surface Model

In this section, we first give a quick recap and then explain the problem with the conventional weighted least squares (WLS) method (see [Nea04] for an overview of different least squares methods). In the following, we assume that points are in $\mathbb{R}^3$, but all methods work, of course, in any dimension.
For the sake of accuracy, we introduce the following definition.

**Definition 7** (Cloud point)**.** *Each point $\mathbf{p}_i$ of a given point cloud $\mathcal{P}$ is denoted as a cloud point.*

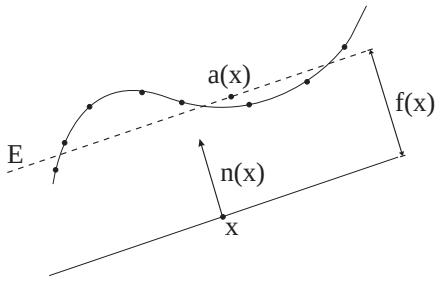



**Figure 4.2:** A normal $\mathbf{n}(\mathbf{x})$ is estimated in each $\mathbf{x}$ that minimizes the weighted squared distances to the plane $E : \mathbf{n}(\mathbf{x}) \cdot (\mathbf{a}(\mathbf{x}) - \mathbf{x}) = 0$. $f(\mathbf{x})$ is the distance of $\mathbf{x}$ to that plane.

### 4.1.1 WLS Surface Definition

Let a point cloud $\mathcal{P}$ with $n$ points $\mathbf{p}_i \in \mathbb{R}^3$ be given. Then, the surface from $\mathcal{P}$ is the zero set

$$S = \big\{\mathbf{x} | f(\mathbf{x}) = 0, \mathbf{x} \in \mathbb{R}^3\big\}$$

of an implicit function [AA03]

$$f(\mathbf{x}) = \mathbf{n}(\mathbf{x}) \cdot (\mathbf{a}(\mathbf{x}) - \mathbf{x}), \tag{4.1}$$

where $\mathbf{a}(\mathbf{x})$ is the weighted average of all points $\mathcal{P}$

$$\mathbf{a}(\mathbf{x}) = \frac{\sum_{i=1}^{n} \theta(\mathbf{x}, \mathbf{p}_i)\mathbf{p}_i}{\sum_{i=1}^{n} \theta(\mathbf{x}, \mathbf{p}_i)}. \tag{4.2}$$

The situation is illustrated in Figure 4.2. Usually, a Gaussian kernel (weight function)

$$\theta(\mathbf{x}, \mathbf{p}) = e^{-d(\mathbf{x},\mathbf{p})^2/h^2}, \quad d(\mathbf{x}, \mathbf{p}) = \|\mathbf{x} - \mathbf{p}\| \tag{4.3}$$

is used, but other kernels work as well (see below).
The bandwidth of the kernel, $h$, allows us to tune the decay of the influence of the points. It should be chosen such that no holes appear.
Theoretically, $\theta$'s support is unbounded. However, it can be safely limited to the extent where it falls below the machine's precision, or some other, suitably small threshold $\theta_\varepsilon$. Alternatively, one could use the cubic polynomial [Lee00]

$$\theta(\mathbf{x}, \mathbf{p}) = 2\Big(\frac{d(\mathbf{x}, \mathbf{p})}{h}\Big)^3 - 3\Big(\frac{d(\mathbf{x}, \mathbf{p})}{h}\Big)^2 + 1,$$

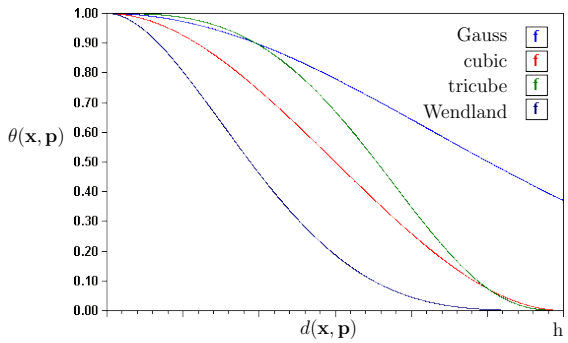



**Figure 4.3:** Our method is independent of the mapping from distances to weights, so different weight functions can be used.

or the tricube weight function [CL95]

$$\theta(\mathbf{x},\mathbf{p}) = \Big(1 - \big|\frac{d(\mathbf{x},\mathbf{p})}{h}\big|^3\Big)^3,$$

or the Wendland function [Wen95]

$$\theta(\mathbf{x},\mathbf{p}) = \Big(1 - \frac{d(\mathbf{x},\mathbf{p})}{h}\Big)^4 \big(4\frac{d(\mathbf{x},\mathbf{p})}{h} + 1\big),$$

all of which are set to 0 for $d(\mathbf{x},\mathbf{p}) > h$ and, thus, have compact support (see Figure 4.3 for a comparison). However, the choice of kernel function is not critical [Här90].
The normal $\mathbf{n}(\mathbf{x})$ is determined by weighted least squares. It is defined as the direction of smallest weighted covariance, i.e., it minimizes

$$\sum_{i=1}^{n} \big(\mathbf{n}(\mathbf{x}) \cdot (\mathbf{a}(\mathbf{x}) - \mathbf{p}_i)\big)^2 \theta(\mathbf{x},\mathbf{p}_i) \tag{4.4}$$

for fixed $\mathbf{x}$ and under the constraint $\|\mathbf{n}(\mathbf{x})\| = 1$.
Note that, unlike [AA03], we use $\mathbf{a}(\mathbf{x})$ as the center of the PCA, which seems to make $f(\mathbf{x})$ much more well-behaved (see Figure 4.4). Also, we do not solve a minimization problem like [Lev03, ABCO$^+$03], because we are aiming at an extremely fast method.
The normal $\mathbf{n}(\mathbf{x})$ defined by (4.4) is the smallest eigenvector (that means, the vector corresponding to the smallest eigenvalue $\lambda_0$) of the centered covariance matrix $\mathbf{B}(\mathbf{x}) = \{b_{ij}(\mathbf{x})\}$ with

$$b_{ij}(\mathbf{x}) = \sum_{k=1}^{n} \theta(\mathbf{x},\mathbf{p}_k)(\mathbf{e}_i(\mathbf{p}_k - \mathbf{a}(\mathbf{x})))(\mathbf{e}_j(\mathbf{p}_k - \mathbf{a}(\mathbf{x}))), \tag{4.5}$$

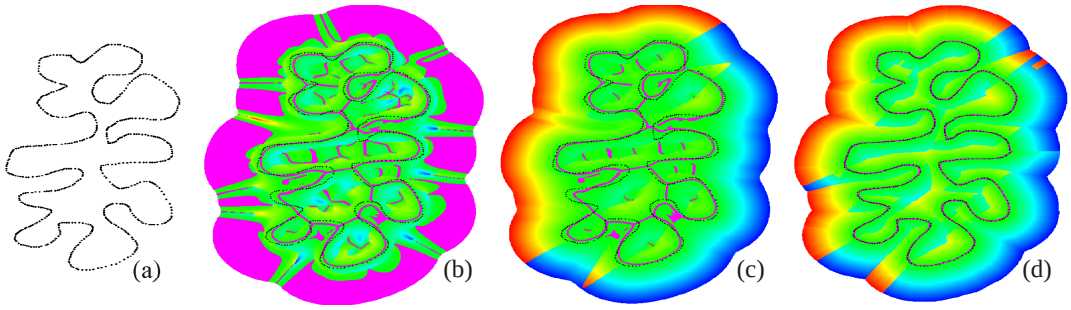



**Figure 4.4:** Visualization of the implicit function $f(\mathbf{x})$ over a 2D point cloud. Points $\mathbf{x} \in \mathbb{R}^2$ with $f(\mathbf{x}) \approx 0$, i.e., points on or close to the surface, are shown magenta. Red denotes $f(\mathbf{x}) \gg 0$ and blue denotes $f(\mathbf{x}) \ll 0$. (a) point cloud; (b) reconstructed surface using the definition of [AA03]; (c) utilizing the centered covariance matrix produces a better surface, but it still has several artifacts; (d) surface and function $f(\mathbf{x})$ based on our more geodesic kernel using the sphere-of-influence graph.

where $\mathbf{e}_i, i \in \{0, 1, 2\}$ is a basis of $\mathbb{R}^3$.
If we assume that $\mathbf{a}(\mathbf{x})$ and $\mathbf{n}(\mathbf{x})$ are continuously differentiable functions (and that $\mathbf{n}(\mathbf{x})$ is unique), then the surface $S$ is a 2D surface.
There are several variations of this simple definition, but for sake of clarity, we will stay with this basic one. Our new method can be applied to more elaborated ones as well.

### 4.1.2 Problems with the Euclidean Kernel

The main problem with the above definition, and virtually all of its variants, is that the Euclidean distance function $d(\mathbf{x}, \mathbf{p}) = \|\mathbf{x} - \mathbf{p}\|$ in Equations 4.3 is not adapted to the topology of the surface and, therefore, makes points "close" to query points[1] $\mathbf{x}$ that are really topologically far away.
As a consequence, such points are weighted too heavily and artifacts in the surface $S$ can appear (see Figure 4.4 and Figure 4.5).
There are two typical cases. First, assume $\mathbf{x}$ is halfway between two (possibly unconnected) components of the point cloud; then it is still influenced by *both* parts of the point cloud, which have similar weights in Equation 4.2 and 4.4. This can lead to an *artificial* zero subset $\subset S$ where there are no points from $\mathcal{P}$ at all.

[1] For each query point $\mathbf{x}$, the approximate distance to the surface can be seen as the value of the implicit function $f(\mathbf{x})$.

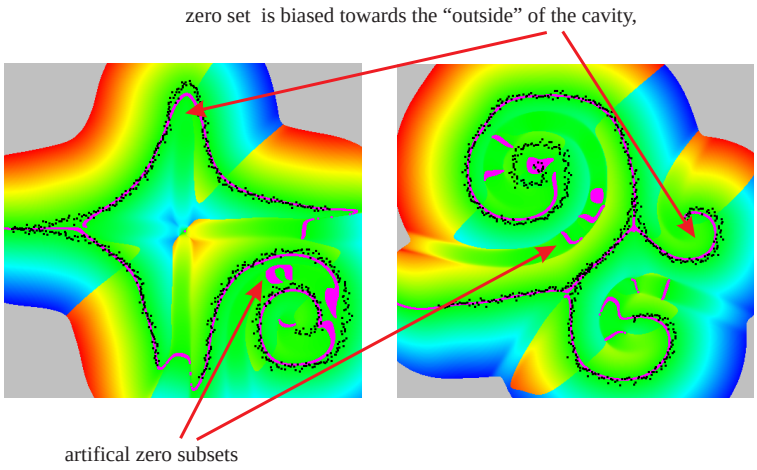



**Figure 4.5:** The Euclidean kernel can produces artifacts as shown in the figure.

Second, let us assume that $\mathbf{x}$ is inside a cavity of the point cloud. Then, $\mathbf{a}(\mathbf{x})$ gets “drawn” closer to $\mathbf{x}$ than if the point cloud was flat. This makes the zero set *biased* towards the “outside” of the cavity, away from the true surface. In the extreme, this can lead to cancellation near the center of a spherical point cloud where all points on the sphere have a similar weight.
This thwarts algorithms based solely on the point cloud representation, such as ray-tracing [AA04a] or our collision detection algorithm for point clouds described in the next Chapter 5. In all of these cases, the problem is caused by the following deficiency in the kernel (4.3). The Euclidean distance $\|\mathbf{x} - \mathbf{p}\|$, $\mathbf{p} \in \mathcal{P}$, can be small, while the distance from $\mathbf{x}$ to the closest point on $S$ and then along the shortest path to $\mathbf{p}$ on $S$ (the geodesic) is quite large.
The problems mentioned above could be alleviated somewhat by restricting the surface to the region $\{\mathbf{x} : ||\mathbf{x} - \mathbf{a}(\mathbf{x})|| < c\}$ (since $\mathbf{a}(\mathbf{x})$ must stay within the convex hull of $\mathcal{P}$). However, this does not help in many cases involving cavities.

## 4.2 Geodesic Distance Approximation

As mentioned above, we propose to use a different distance function that is based on geodesic distances on the surface $S$. Unfortunately, we do not have an explicit reconstruction of $S$, and in many applications, we do not even want to construct one.
Therefore, the idea of our method is to utilize (conceptually) a Voronoi diagram to find the nearest neighbor of a query point $\mathbf{x}$, and then traverse the Voronoi diagram breadth-first to compute approximate geodesic distances between the

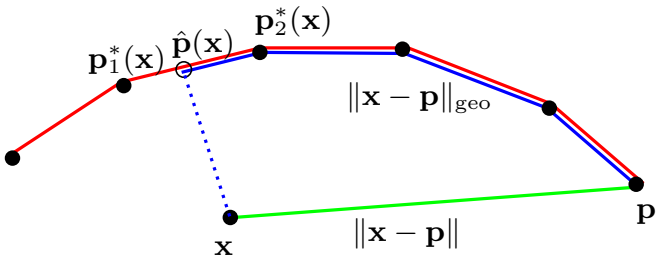



**Figure 4.6:** Instead of the Euclidean distance, we use an approximate geodesic distance based on the close-pairs shortest-paths matrix over a proximity graph.

query point and the cloud points. Since the Voronoi diagram, in this context, basically provides just an adjacency relation based on some notion of proximity, we can also use other *proximity graphs* where the nodes are points $\in \mathcal{P}$.[2]
In such proximity graphs, nodes $\mathbf{p}$ and $\mathbf{q}$ are connected by an edge if some geometric proximity predicate holds. So, it is obvious that geodesic distances between the points can be approximated by shortest paths on the edges of the graph.
There is a whole spectrum of different proximity graphs over a set $\mathcal{P}$, for instance the Delaunay graph $DG(\mathcal{P})$, the Gabriel graph, the relative neighbor graph, and the nearest neighbor graph [JT92]. These are all subgraphs of the DG, with different densities, so we choose to investigate the DG. Another interesting proximity graph seems to be the sphere-of-influence graph $SIG(\mathcal{P})$ introduced by Toussaint [Tou88], because it is not a subgraph of the DG, and because it seems to capture the notion of *sampling density* fairly well (see below).
In the following, the length of an edge is the Euclidean distance $\|\mathbf{p}-\mathbf{q}\|$ (or any other metric).

### 4.2.1 Geodesic Kernel

We define our new distance function $d_{\text{geo}}(\mathbf{x},\mathbf{p})$ as follows. Given some location $\mathbf{x}$, we compute its nearest neighbor $\mathbf{p}_1^*(\mathbf{x}) \in \mathcal{P}$. Then, we compute the closest point $\hat{\mathbf{p}}(\mathbf{x})$ to $\mathbf{x}$ that lies on an edge adjacent to $\mathbf{p}_1^*(\mathbf{x})$. Let $\mathbf{p}_2^*(\mathbf{x})$ denote the other point adjacent to that edge (see Figure 4.6) and let $l(\mathbf{p}^*,\mathbf{p})$ denote the accumulated length of the shortest path from $\mathbf{p}^*$ to $\mathbf{p}$ for any $\mathbf{p} \in \mathcal{P}$.

[2] Another way to encode approximate geodesic distances is a triangle mesh. However, here we would have a "bootstrapping" problem, because the mesh would have to be created by some kind of tesselation, which would need to evaluate the implicit function at many points in space.

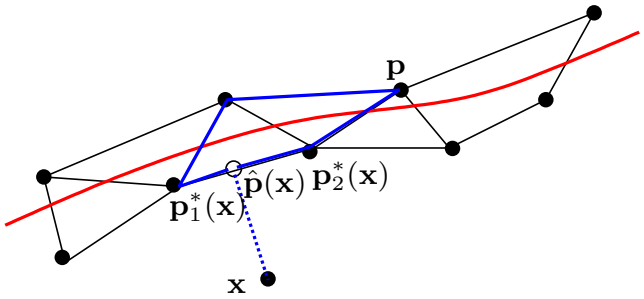



**Figure 4.7:** Interpolating geodesic distances between $p_1^*(\mathbf{x})$ and $p_2^*(\mathbf{x})$ allows for defining the distance function with as few discontinuities as possible.

Now, conceptually, the distance from $\mathbf{x}$ to any $\mathbf{p} \in \mathcal{P}$ could be defined as

$$\tilde{d}_{\text{geo}}(\mathbf{x}, \mathbf{p}) = \min_{\mathbf{p}^* \in \{\mathbf{p}_1^*(\mathbf{x}), \mathbf{p}_2^*(\mathbf{x})\}} l(\mathbf{p}^*, \mathbf{p}) + \|\hat{\mathbf{p}}(\mathbf{x}) - \mathbf{p}^*\|.$$

The accumulated length $l(\mathbf{p}^*, \mathbf{p})$ can be multiplied by the number of "hops" along the path: if an (indirect) neighbor $\mathbf{p}$ is reached by a shortest path with many hops, then there are many points in $\mathcal{P}$ that should be weighted much more than $\mathbf{p}$, even if the accumulated length of the shortest path from $\mathbf{p}^*$ to $\mathbf{p}$ is small. This is independent of the concrete proximity graph used for computing the shortest paths. That means, we can weight two points differently, although their distances along shortest paths to point $\mathbf{x}$ is the same. Especially in very irregular sampled point clouds, this seems to be very promising.
However, it is not obvious that this is always the desired distance. In addition, it is desirable to define the distance function with as few discontinuities as possible. Therefore, we just take the weighted average (see Figure 4.7)

$$\begin{aligned} d_{\text{geo}}(\mathbf{x}, \mathbf{p}) = & \frac{\|\hat{\mathbf{p}}(\mathbf{x}) - \mathbf{p}_2^*(\mathbf{x})\|}{\|\mathbf{p}_1^*(\mathbf{x}) - \mathbf{p}_2^*(\mathbf{x})\|} \left[ l(\mathbf{p}_1^*(\mathbf{x}), \mathbf{p}) + \|\hat{\mathbf{p}}(\mathbf{x}) - \mathbf{p}_1^*(\mathbf{x})\| \right] \\ & + \frac{\|\hat{\mathbf{p}}(\mathbf{x}) - \mathbf{p}_1^*(\mathbf{x})\|}{\|\mathbf{p}_1^*(\mathbf{x}) - \mathbf{p}_2^*(\mathbf{x})\|} \left[ l(\mathbf{p}_2^*(\mathbf{x}), \mathbf{p}) + \|\hat{\mathbf{p}}(\mathbf{x}) - \mathbf{p}_2^*(\mathbf{x})\| \right]. \end{aligned} \tag{4.6}$$

Note that we do *not* add $\|\mathbf{x} - \hat{\mathbf{p}}(\mathbf{x})\|$. The effect is that $f(\mathbf{x})$ is non-zero everywhere far away from the point cloud.
Overall, we use the following *geodesic kernel*

$$\theta(\mathbf{x}, \mathbf{p}) = e^{-d_{\text{geo}}(\mathbf{x}, \mathbf{p})^2 / h^2} \tag{4.7}$$

when computing $f$ by (4.1)–(4.5).

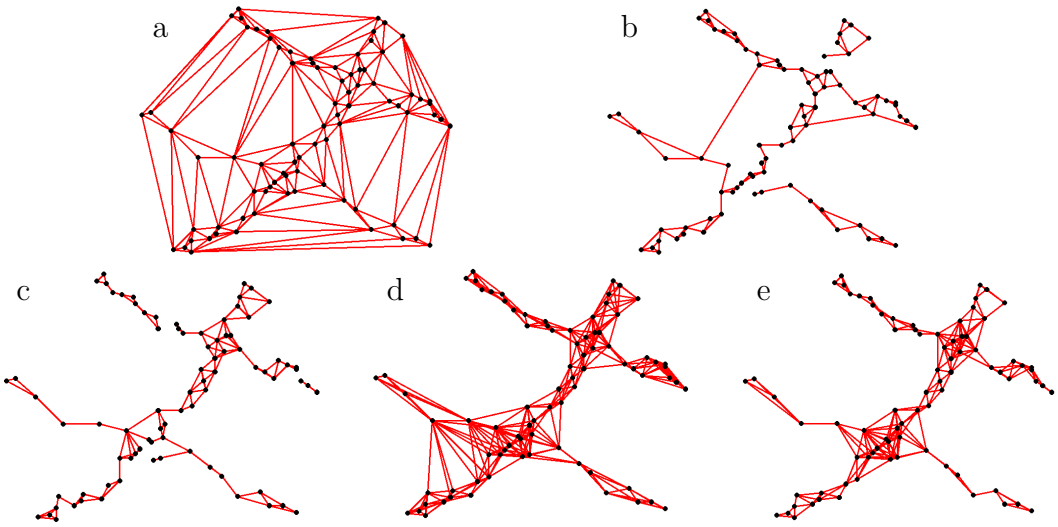



**Figure 4.8:** Different proximity graphs. (a) $DG(\mathcal{P})$, (b) $DG(\mathcal{P})$ where edges are pruned according to $Q_3 + IQR$, (c) $1 - SIG(\mathcal{P})$, (d) $3 - SIG(\mathcal{P})$, (e) $3 - SIG(\mathcal{P})$ with pruning.

### 4.2.2 Proximity by Delaunay Graph

It is very intuitive to use the Delaunay graph $DG(\mathcal{P})$ for our problem, because [ASCL02] described an intriguing algorithm for reconstructing a polygonal surface over a point cloud without noise from its Voronoi diagram (which is the dual of the Delaunay graph). Later, [DG04] extended this to provable reconstruction from noisy models.

So we investigated the possible use of the $DG(\mathcal{P})$ as a proximity graph. Since it induces a neighborhood relation that also includes "long distance" neighborhoods, some shortest paths can "tunnel" through space that should really be a gap in the model (see Figure 4.8, left). Therefore, we prune edges from $DG(\mathcal{P})$ based on criteria that involve an estimation of the local spatial density of the point cloud (see below). However, this can make the $DG(\mathcal{P})$ too sparse, while at the same time it does not always prune all "long" edges. This will cause artifacts in the surface (see Figure 4.9, left).

If our point cloud is well-sampled in the sense of [ASCL02], then we could prune all edges incident to a point $\mathbf{p} \in \mathcal{P}$ that are longer than the distance of $\mathbf{p}$ from the medial axis of $S$ — *provided* we knew that distance for each $\mathbf{p}$. This is, of course, not feasible.

Therefore, we propose to utilize a statistical outlier detection method to prune edges. This is motivated by the observation that most of the unwanted "long distance" edges are local outliers, or form a cluster of outliers. In the following,

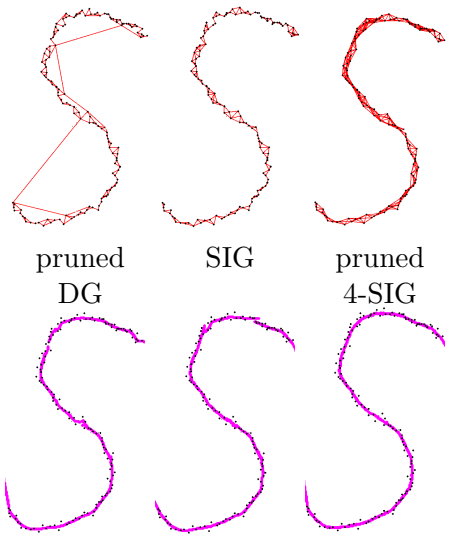



**Figure 4.9:** Surface induced by different proximity graphs. Clearly, the pruned DG and the plain SIG incur artifacts, due to sparsity and disconnected components. In our experience, the 4- or 5-SIG, with pruning, seems to work well in all cases.

we describe a simple outlier detection algorithm that seems to perform well in our case, but, of course, other outlier detection algorithms [VB94] should work as well.
In statistics, an outlier is a single observation which is far away from the rest of the data. One definition of "far away" in this context is "greater than $Q_3 + 1.5 \cdot IQR$", where $Q_3$ is the third quartile, and $IQR$ is the interquartile range $Q_3 - Q_1$. Our experiments showed that best results are achieved by pruning edges with length of at least $Q_3 + IQR$.

### 4.2.3 Proximity by Sphere-of-Influence Graph

The sphere-of-influence graph (denoted as SIG) is a fairly little known proximity graph [Tou88, BLS00, MQ03]. The idea is to connect points if their "spheres of influence" intersect. More precisely, for each point $\mathbf{p}_i$ the distance $d_i$ to its nearest neighbor is determined and two points $\mathbf{p}_i$ and $\mathbf{p}_j$ are connected by an edge, if $\|\mathbf{p}_i - \mathbf{p}_j\| \leq d_i + d_j$.
As a consequence, the SIG tends to connect points that are "close" to each other relative to the local point density. In contrast to the $DG(\mathcal{P})$, no "long distance" neighbor relations are created (see Figure 4.8 c), except for some pathological cases when the surface is very irregularly sampled.

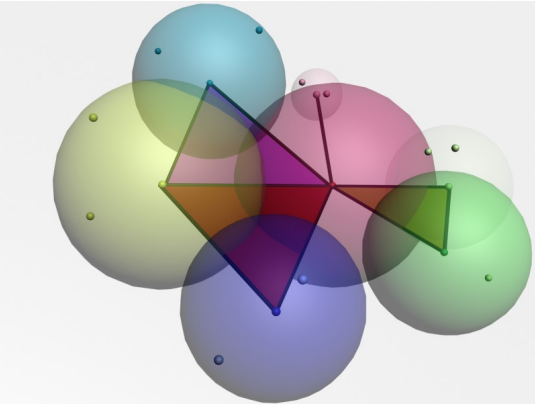


**Figure 4.10:** The sphere-of-influence graph can be extended to a sphere-of-influence complex in 3D, which allows better geodesic approximations than the sphere-of-influence graph (in 3D). Note that only a few of all simplices of the complex for this point set are shown.

### 4.2.4 Extensions of the SIG

In the following, we propose several extensions to the plain SIG.

#### The $r$-th order SIG

In noisy or irregularly sampled point clouds, there can be several pairs of points that are placed much farther apart from each other than their inter-pair separation. In such situations, the $SIG(\mathcal{P})$ would consist of a lot of isolated "mini-clusters", even though there are no holes in the original surface (see Figure 4.9, middle). Consequently, the corresponding surface could not be reconstructed correctly, because the approximated geodesic distances are too imprecise: on the one hand, they are too large because points close together can only indirectly be accessed through the graph by visiting other nodes; on the other hand — in the case of unconnected components — for some points in space, too few cloud points are considered for the reconstruction.

To overcome this problem, we propose to use the $r$-SIG [GPS92]: instead of computing the distance to the nearest neighbor for each node, we compute the distance to the $r$-nearest neighbor and then proceed as in the case of $r = 1$. It is obvious that the larger $r$, the more nodes are directly connected by an edge, and that too large $r$ can result in "long distance" edges as in the case of the $DG(\mathcal{P})$ (see Figure 4.8 d).

In our experience, it seems best to choose $r = 3$ or $r = 4$, and then prune away all "long" edges by an outlier detection algorithm as described in Section 4.2.2 (see Figure 4.8 e), which yields almost always a nice surface (see Figure 4.9,

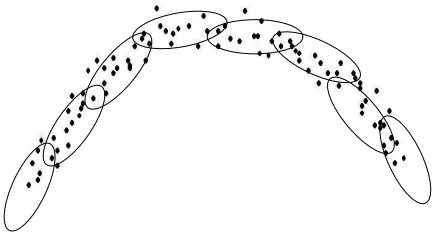


**Figure 4.11:** We propose the anisotropic SIG to adapt the "spheres"-of-influence better to the neighborhood of the points, thus yielding a better proximity graph for our purposes.

right).

### Sphere-of-Influence Complex

In 3D, we can either stay with the $r$-SIG presented so far, or we can extend the definition analogously to contain triangles as well. Thus, we obtain a complex that consists of the vertices, the edges as defined before, and triangles. The latter are formed among all triples of points $(\mathbf{p}, \mathbf{q}, \mathbf{r})$ that are close to each other: $\mathbf{p}, \mathbf{q}, \mathbf{r}$ are considered close iff $B_p \cap B_q \cap B_r \neq \varnothing$, where $B_x$ is the sphere of influence around $\mathbf{x}$ with radius $r_x$.[3] Figure 4.10 shows a few triangles and points from such a complex.

The advantage of such a complex is that we can compute much better approximations to geodesic paths than if the paths were restricted to edges only [KS00, CH90, AHPK00]. We propose the algorithm of Kanai and Suzuki [KS00] with some slight modifications to determine efficiently approximate shortest path on the triangles [Sch04].

### Anisotropic SIG

Sometimes, the "post-processing" of the $r$-SIG, as described above, can prune away too many or too few edges. In order to reduce the susceptibility of the proximity graph to the pruning threshold, and in order to adapt it better to our problem, we propose the *anisotropic SIG*. The idea is to use ellipsoids instead of spheres around the points, where the axes of the ellipsoids are the principal components of a suitable number of neighboring points (see Figure 4.11).

For each point $\mathbf{p} \in \mathcal{P}$, we construct its "ellipsoid"-of-influence as follows. We start with its $k$ nearest neighbors, $k$ being small (3 or 4).[4] Then, we compute

[3] This is somewhat related to the Czech complex. However, in the Czech complex all spheres have the same radius.

[4] Should these, including $\mathbf{p}$, be (almost) coplanar, we take more until we get a non-coplanar set.

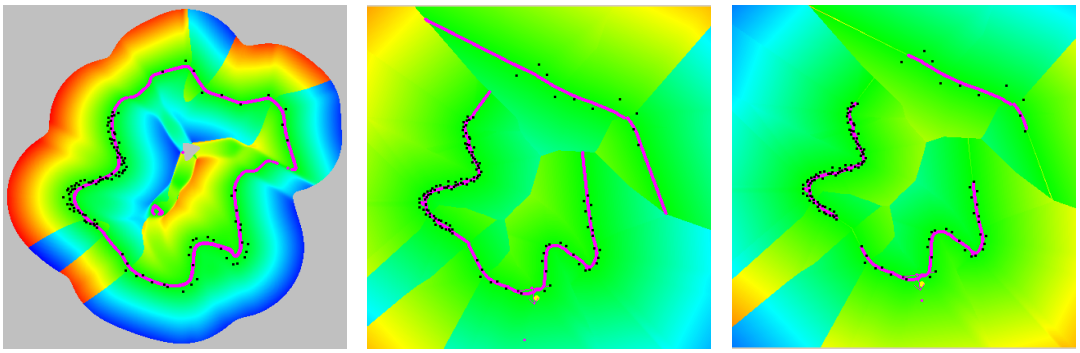
the principal axes of this set and determine radii so that the ellipsoid contains the set.[5] Then, we (conceptually) scale the ellipsoid until it contains one more point.[6] With this increased neighborhood point set, we compute a new ellipsoid, as before. We repeat this procedure, until the ellipsoid contains $r$ points, or until $\sigma(\mathbf{p}) = \frac{\lambda_1}{\lambda_1+\lambda_2+\lambda_3}$ exceeds a predefined threshold. Here, $\lambda_1$ is the smallest eigenvector, so that $\sigma(\mathbf{p})$ captures the surface variation [PGK02].
The advantage of this is, that in areas with small variance, we can automatically choose large spheres-of-influence, while in areas with high curvature or close sheets these spheres are kept small. Thus, the edges of the anisotropic SIG are less prone to "short-circuit" sharp features (as depicted in Figure 4.13).
Finally, for determining the edges of the anisotropic SIG, we need to check intersections of ellipsoid. This could be computed exactly by [Sch94, Ch. 3.3], for instance, but we have opted for a simpler, approximate check: we just sample each ellipsoid by a number of points and check whether any of them is contained in the other one.

### 4.2.5 Automatic and local bandwidth computation

A critical parameter is the bandwidth $h$ in (4.3). On the one hand, if this is chosen too small, then variance may be too large, i.e., noise, holes, or other artifacts may appear in the surface. On the other hand, if it is chosen too large, then bias may be too large, i.e., small features in the surface will be smoothed out. To overcome this problem, [PGK02] proposed to scale the parameter $h$ adaptively.
Here, we can use the proximity graph (the $r$-SIG) to estimate the local sampling density, $r(\mathbf{x})$, and then determine $h$ accordingly. Thus, $h$ itself is a function $h = h(\mathbf{x})$.
Assuming the terminology from Section 4.2.1 (see Figure 4.7), let $r_1(\mathbf{x})$ and $r_2(\mathbf{x})$ be the lengths of the longest edges incident to $\mathbf{p}_1^*(\mathbf{x})$ and $\mathbf{p}_2^*(\mathbf{x})$, respectively. Then, we determine $r(\mathbf{x})$ as the weighted average of $r_1(\mathbf{x})$ and $r_2(\mathbf{x})$ subdivided by $r$, the number of nearest neighbors that determine the radius of each sphere-of-influence

$$r(\mathbf{x}) = \frac{1}{r} \cdot \left(\frac{\|\hat{\mathbf{p}}(\mathbf{x}) - \mathbf{p}_2^*(\mathbf{x})\|}{\|\mathbf{p}_2^*(\mathbf{x}) - \mathbf{p}_1^*(\mathbf{x})\|} \cdot r_1(\mathbf{x}) + \frac{\|\hat{\mathbf{p}}(\mathbf{x}) - \mathbf{p}_1^*(\mathbf{x})\|}{\|\mathbf{p}_2^*(\mathbf{x}) - \mathbf{p}_1^*(\mathbf{x})\|} \cdot r_2(\mathbf{x})\right) \tag{4.8}$$

[5] Alternatively, we could have computed a smallest enclosing ellipsoid by [Wel91], but the benefit seemed questionable.
[6] This can be done efficiently by transforming the points in the coordinate system of the ellipsoid and then choosing the closest one.



plain WLS, $h = 22$ | 2-SIG with pruning, $\eta = 1.7$ | plus boundary detection

**Figure 4.12:** Our method offers automatic sampling density estimation individually for each point, which allows to determine the bandwidth automatically and independently of scale and sampling-density (middle), and to detect boundaries automatically (right).

$$h(\mathbf{x}) = \frac{\eta\, r(\mathbf{x})}{\sqrt{\ln \frac{1}{\theta_\varepsilon}}}, \tag{4.9}$$

where $\theta_\varepsilon$ is a suitably small value (see Section 4.1.1). Thus, $\mathbf{p}_i$ with distance $\eta\, r(\mathbf{x})$ from $\hat{\mathbf{p}}(\mathbf{x})$ will be assigned a weight of $\theta_\varepsilon$ (see Equation 4.3).
We have now replaced the scale- and sampling-dependent parameter $h$ by another one, $\eta$, that is independent of scale and sampling density. In our experience, this can just be set to 1, or it can be used to adjust the amount of "smoothing". Note that this automatic bandwidth detection works similarly for many other kernels as well (see Section 4.1.1).
Depending on the application, it might be desirable to involve more and more points in (4.1), so that $\mathbf{n}(\mathbf{x})$ becomes a least squares plane over the complete point set $\mathcal{P}$ as $\mathbf{x}$ approaches infinity. In that case, we can just add $\|\mathbf{x} - \hat{\mathbf{p}}(\mathbf{x})\|$ to (4.8).
Figure 4.12 shows, that the automatic bandwidth determination allows the WLS approach to handle point clouds with varying sampling densities without any manual tuning. Notice that the smoothing of the different sampling densities is very similar, compared to the "scale" (i.e., density).

### 4.2.6 Automatic boundary detection

Another benefit of the automatic sampling density estimation is a very simple boundary detection method. Our method builds on the one proposed by

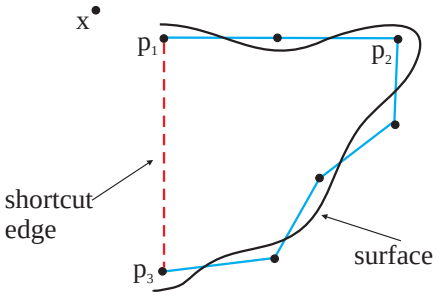



**Figure 4.13:** Shortcuts across cavities lead to imprecise approximate geodesic distances, especially if the path length is multiplied by the number of hops along the path.

[AA04a].
The idea is simply to discard points $\mathbf{x}$ with $f(\mathbf{x}) = 0$, if they are "too far away" from $\mathbf{a}(\mathbf{x})$ *relative* to the sampling density in the vicinity of $\mathbf{a}(\mathbf{x})$. More precisely, we define a new implicit function

$$\hat{f}(\mathbf{x}) = \begin{cases} f(\mathbf{x}), & \text{if } |f(\mathbf{x})| > \varepsilon \vee \|\mathbf{x} - \mathbf{a}(\mathbf{x})\| < 2r(\mathbf{x}) \\ \|\mathbf{x} - \mathbf{a}(\mathbf{x})\|, & \text{else .} \end{cases} \tag{4.10}$$

Figure 4.12 (right) shows, that this simple method is able to handle different sampling densities fairly well.

### 4.2.7 Smooth Surfaces

It has been shown that well-sampled surfaces lead to smooth distance fields with non-zero gradients. Therefore, the MLS surface is a continuously differentiable $C^\infty$ manifold, iff the weighting function is continuously differentiable [Lev98, Lev03].
Moreover, the proximity graph should not contain edges which significantly "short-circuit" cavities. In that case, the shortcut would lead to much higher weights for farther points than for some points in-between. This problem is illustrated in Figure 4.13. Assume, point $\mathbf{p}_1$ is the closest point to some space point $\mathbf{x}$. Then, $\mathbf{p}_2$ can have a higher approximate geodesic distance from $\mathbf{p}_1$ than $\mathbf{p}_3$ from $\mathbf{p}_1$.
An outlier detection algorithm as proposed in Section 4.2.2 cannot guarantee that there are no shortcuts any longer, especially if the lengths of neighboring edges are nearly the same. To overcome the problem, we could do the following (note that this is only an idea for future work). As we will show in Section 4.3.1, it is sufficient to use only paths up to some length and store them in a CPSP

(close-pairs shortest-path) lookup table, because computing shortest paths on the fly would be too expensive. As a consequence, we could precompute simple B-splines through sets of points that are connected through the graph and lie within a certain radius. Thus, the proximity graph is only used for the partitioning/preselection of the points, and the B-splines are used for determining the lengths of shortest paths (note that in 3D the preselection becomes more difficult; probably B-spline surfaces could also be used).
However, there can be still discontinuities in our geodesic $d_{\mathsf{geo}}$ and, thus, in the function $f$. These can occur at the borders of the Voronoi regions of the cloud points, in particular at borders where the Voronoi sites are far apart from each other, such as the medial axis.
More precisely, they can occur at points $\mathbf{x}$ where the point $\hat{\mathbf{p}}(\mathbf{x})$ jumps to a different edge. In the case that the new edge is adjacent to the old one, only the length $l(\mathbf{p}_2^*, \mathbf{p})$ changes. Otherwise, both lengths $l(\mathbf{p}_1^*, \mathbf{p})$ and $l(\mathbf{p}_2^*, \mathbf{p})$ change. That means, given two points $\mathbf{x}_1$ and $\mathbf{x}_2$ that are very close to each other ($||\mathbf{x}_1 - \mathbf{x}_2|| < \epsilon$), the difference between their geodesic distances to a point $\mathbf{p} \in \mathcal{P}$ can be large

$$|d_{\mathsf{geo}}(\mathbf{x}_1, \mathbf{p}) - d_{\mathsf{geo}}(\mathbf{x}_2, \mathbf{p})| \gg \epsilon,$$

so that the geodesic distance function is not continuous

$$\lim_{x_1 \to x_2} d_{\mathsf{geo}}(\mathbf{x}_1, \mathbf{p}) \neq d_{\mathsf{geo}}(\mathbf{x}_2, \mathbf{p}).$$

This problem could be reduced by interpolating a small set of neighboring splines. Using a simple Gaussian distribution of points around the point $\mathbf{x}$, we could interpolate the geodesic distances for that points.

## 4.3 Running time and Complexity

This section gives an overview of the complexity of our auxiliary data structures and the time for evaluating $f(\mathbf{x})$ in the 3D case. Therefore, we introduce the following definition of sampling radius. Since our surfaces do not interpolate the point cloud, we do not use the notion of *ε-sampling* [ACDL00]. In addition, we believe our definition is more practical.

**Definition 8** (Sampling radius)**.** *Consider a set of spheres, centered at points $p_i \in \mathcal{P}$, that cover the surface defined by $\mathcal{P}$, where all spheres have equal radius. We define the sampling radius $r(\mathcal{P})$ as the minimal radius of such a sphere covering.*

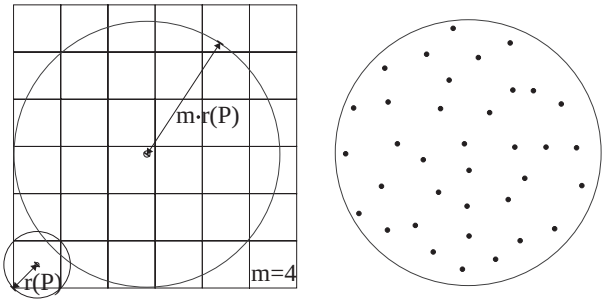



**Figure 4.14:** Under reasonable assumptions, the close-pairs shortest-paths map has size $O(n)$. Left: a sphere with radius $m \cdot r(\mathcal{P})$ can be covered by $O(m^3)$ spheres with radius $r(\mathcal{P})$. Right: $\lceil\sqrt{3} \cdot m\rceil^3$ uniformly distributed points inside.

### 4.3.1 Close-Pairs Shortest-Paths

Computing shortest paths on the fly during the computations of our geodesic distances would be, of course, prohibitively expensive, so we pre-compute them. However, computing and storing an *all pairs shortest paths* (APSP) matrix would also be infeasible for larger point clouds. As a consequence, we propose to compute only a relevant subset of shortest paths.
Since the Gaussian (4.3) decays fairly quickly (for reasonable choices of $h$), and other weight functions have bounded support, we need to store only paths up to some length $l = d_{\mathsf{geo}}(\mathbf{x}, \mathbf{p})$. The contribution of nodes in Equations 4.2 and 4.5 that are farther away can be neglected. The length $l$ should be chosen so that $\theta(\mathbf{x}, \mathbf{p})$ falls below the machine's precision, or some other, small threshold $\theta_\varepsilon$. In practice, $l \leq m \cdot r(\mathcal{P})$ with $m \approx 5$ has shown to be suitable (when the bandwidth of $\theta$ is automatically detected as described in Section 4.2.5).
However, now there are regions in $\mathbb{R}^3$ where only a small number of $\mathbf{p}_i$ are taken into account for computing $\mathbf{a}(\mathbf{x})$ and $\mathbf{n}(\mathbf{x})$. We amend this by dismissing points $\mathbf{x}$ for which the number $c$ of $\mathbf{p}_i$ taken into account would be too small. Note that $c$ and $\theta_\varepsilon$ are independent parameters. We remark here that [AA04a] proposed an amendment, too, although differently specified and differently motivated.
Overall, we define the surface $S$ as the constrained zero-set of $f$, i.e.,

$$S = \big\{\mathbf{x} \in \mathbb{R}^3 \mid f(\mathbf{x}) = 0\,,\, \#\{\mathbf{p} \in P : \|\mathbf{p} - \mathbf{x}\| < r_\varepsilon\} > c\big\}, \tag{4.11}$$

where Equation 4.3 implies $r_\varepsilon = h \cdot \sqrt{|\log \theta_\varepsilon|}$.
The following lemma shows how to compute the relevant subset of shortest paths in linear time.

**Lemma 1.** *Let a proximity graph or complex $\mathcal{G}$ over a point cloud $\mathcal{P}$ with $n$ uniformly distributed points as well as an arbitrary constant $m$ be given. If we restrict the all-pairs shortest-path problem to find only path up to some length $m \cdot r(\mathcal{P})$ in $\mathcal{G}$, we can solve it in time $O(n)$. The resulting paths can be stored in a CPSP (close-pairs shortest path) map of size $O(n)$.*

*Proof:* Let $\theta(d) = e^{-d^2/h^2}$. As assumed, only points $\mathbf{p}_j$ with $\theta(\|\mathbf{x} - \mathbf{p}_j\|) \geq \theta(m \cdot r(\mathcal{P}))$ have an influence in Equation 4.1. As a consequence, for each point $\mathbf{p}_i \in \mathcal{P}$ we can run a single-source shortest-path (SSSP) algorithm[7] for the source $\mathbf{p}_i$ to points $\mathbf{p}_j$, where $\theta(d_{\mathsf{geo}}(\mathbf{p}_i, \mathbf{p}_j)) < \theta(m \cdot r(\mathcal{P}))$. Such points $\mathbf{p}_j$ are obviously contained in a sphere $S_i$ with radius $m \cdot r(\mathcal{P})$ centered at $\mathbf{p}_i$, and they can easily be determined by a depth-first or breath-first search starting at $\mathbf{p}_i$. The following lemma shows that only a constant number of points is inside $S_i$, if $\mathcal{P}$ is a uniform (possibly noisy) sampling of a surface. As a consequence, we have to start $n$ times a SSSP algorithm for a point set of constant size. Overall, our CPSP map can be computed and stored in $O(n)$ time and space. ∎

**Lemma 2.** *Let a point cloud $\mathcal{P}$ with uniformly distributed points $\mathbf{p}_i \in \mathbb{R}^d$ ($d \in \{2, 3\}$) and sampling radius $r(\mathcal{P})$ be given. Then, at most $\lceil \sqrt{d} \cdot m \rceil^d$ points $\in \mathcal{P}$ lie in a sphere with radius $m \cdot r(\mathcal{P})$.*

*Proof:* In the following, we consider only the 3D case ($d = 3$), the 2D case can be shown analogously.
A sphere $S_1$ with radius $m \cdot r(\mathcal{P})$ can be covered with at most $c := \lceil \sqrt{3} \cdot m \rceil^3$ smaller spheres of radius $r(\mathcal{P})$. This has already been shown by Rogers [Rog63]: the sphere $S_1$ can be covered by a cube with side length $2mr(\mathcal{P})$ and the smaller spheres with radius $r(\mathcal{P})$ cover cubes with side length $\sqrt{\frac{4}{3}} r(\mathcal{P})$ (see Figure 4.14 left). As a consequence, the larger cube can be covered by

$$c := \lceil \frac{2mr(\mathcal{P})}{\sqrt{\frac{4}{3}} r(\mathcal{P})} \rceil^3 \tag{4.12}$$

smaller cubes and therefore by the same number of spheres with radius $r(\mathcal{P})$. That means, $c$ uniformly distributed spheres of radius $r(\mathcal{P})$ with centers in $S_1$ cover $S_1$. Only if the spheres are not uniformly distributed, more than $c$ spheres with sampling radius $r(\mathcal{P})$ are necessary to cover $S_1$. ∎
Note that in many practical cases where $S_1$ contains only a single 2D surface with low curvature, $O(\lceil \sqrt{2} \cdot m \rceil^2)$ is already an upper bound for the number of points inside $S_1$.

[7] In case of a Sphere-of-Influence Complex $\mathcal{G}$, we have to use our modified algorithm [Sch04] to determine efficiently (in time $O(n)$) approximate shortest path on the triangles.

### 4.3.2 Pre-computations of Proximity Graphs

Although the complexity of the 3D Delaunay graph can be quadratic in the worst-case, it can be shown that the complexity of the 3D Delaunay graph is linear, if we impose two (mild) sampling conditions on the point cloud. First, the points must be uniformly distributed on a fixed number of facets of $\mathbb{R}^3$. Second, the sample cannot become arbitrarily dense locally. These conditions can be expressed in a more formal way by the notion of $(\epsilon, \kappa)$-sampling [AB04].

**Lemma 3.** *If a point cloud $\mathcal{P}$ is a $(\epsilon, \kappa)$-sample of the true surface, the complexity of the corresponding 3D Delaunay graph is linear in $n = |\mathcal{P}|$. Moreover, it can be constructed in time $O(n \log^2 n)$.*

*Proof:* Attali and Boissonnat [AB04] showed recently, that the complexity of the Delaunay graph is linear in the number of points, if the point cloud is a $(\epsilon, \kappa)$-sample of the true surface. Chan et al. [CSY97] proposed an output sensitive algorithms to compute the Delaunay graph. Their result says, that one can construct the Delaunay graph in 3D in time $O((n + f) \log^2 f)$, where $f$ is the number of faces in the Delaunay triangulation. Applied with $f = O(n)$ (which is the result of Attali and Boissonnat) gives an upper bound of $O(n \log^2 n)$. ∎

**Lemma 4.** *The $r$-SIG can be determined in time $O(n)$ on average for uniformly and independently point-sampled models with size $n$ in any fixed dimension. Moreover, it consumes only linear space in the worst case.*

*Proof:* Dwyer [Dwy95] proposed an algorithm to determine a SIG in linear time in the average case for uniform point clouds. As $r$ is constant, this algorithm can easily be modified, so that it can also compute the $r$-SIG in linear time (see Algorithm 5). The algorithm consists of three steps.
First, the algorithm identifies the $r$-nearest neighbors of each point by utilizing the *spiral search* proposed by [BWY80]: the space is subdivided into $O(n)$ hypercubic cells, the points are assigned to cells, and the $r$-nearest neighbors of each point $\mathbf{p}$ are found by searching the cells in increasing distance from the cell containing $\mathbf{p}$. As $O(1)$ cells are searched for each point on average and a single query can be done in $O(1)$ [BWY80], this first step can be done in time $O(n)$.
Second, each point is inserted into every cell that intersects the $r$-nearest-neighbor sphere. On average, most spheres are small, so that each point is inserted into a constant number of cells, and a constant number of points is inserted into each cell.
Finally, within each cell, all pairs of points that have been assigned to this cell are tested for intersection of their spheres-of-influence. Because each cell contains only a constant number of points, this can also be done in time $O(n)$.

```
constructSIG(P)
initialize grid with n cells
for all p ∈ P do
    assign p to its grid cell
for all p ∈ P do
    find r-th nearest neighbor to p by searching the grid cells in
    spiral order around p with increasing distance
for all p ∈ P do
    for all cells around p that intersect the sphere-of-influence
    around p (in spiral order) do
        assign p to cell
for all cells in the grid do
    for all pairs p_i, p_j of points assigned to the current cell do
        if spheres-of-influence of p_i and p_j intersect then
            create edge p_i p_j
```

Algorithm 5: Simple algorithm to compute the $r$-SIG in $O(n)$ time on average.

[AH85] have shown that the 1-SIG has at most $c \cdot n$ edges, where $c$ is a constant. This $c$ is always bounded by 17.5 [AH85, ERW89]. [GPS92] extended this result to the $r$-SIG over a point cloud from $\mathbb{R}^d$ and showed that the number of edges is bounded by $c_d \cdot r \cdot n$ where the constant $c_d$ depends only on the dimension $d$. That means, the $r$-SIG consumes $O(n)$ space in the worst case. ∎
Moreover, as mentioned in [Tou88], ElGindy has observed that the line-segment intersection algorithm introduced by [BO79] can be used to construct a SIG in the plane in $O(n \log n)$ time in the worst case. The algorithm of [GPS92] constructs the $r$-SIG in time

$$O(n^{2-\frac{2}{1+\lfloor (d+2)/2 \rfloor}+\epsilon} + rn \log^2 n),$$

for any $\epsilon > 0$ in the worst case.

### 4.3.3 Function Evaluation

**Lemma 5.** *The implicit function $f(\mathbf{x})$ using our new geodesic kernel can be evaluated in time $O(\log n)$. This is the same time as in the case of the Euclidean kernel.*

*Proof:* For evaluating $f(\mathbf{x})$ we have to compute a couple of geodesic distances, but all paths are starting from the same point $\hat{\mathbf{p}}(\mathbf{x})$. That means, the nearest neighbor search has to be performed only once.

Under mild conditions, this can be done in $O(\log n)$ time by utilizing a Delaunay hierarchy [Dev02], but this may not always be practical. Using a modified k-d tree (BBD tree), an approximate nearest neighbor can be found in $O(\log n)$ [AMN+98]. Note that the BBD tree can be constructed in time $O(n \log n)$ in any dimension and consumes only $O(n)$ space. Alternatively, a simple k-d tree, whose nodes are always cut on the longest side, allows for an approximate nearest neighbor search in time $O(\log^3 n)$ [DDG00].

As shown in Section 4.3.1, all points $\mathbf{p}_i$ influencing $\mathbf{x}$ can be determined in constant time by a depth-first or breadth-first search.

Overall, $f(\mathbf{x})$ can be determined in $O(\log n)$, which is the same time as in the case of the Euclidean kernel, if we would also restrict the influence of points there. ∎

In order to achieve also a fast practical function evaluation, we implemented the following algorithm for computing the smallest eigenvector.

First, we compute the three eigenvalues by determining the roots of the cubic characteristic polynomial of $\mathbf{B}$ [PFTV93]. Let $\lambda$ be the smallest of them. Then, we compute the associated eigenvector using the Cholesky decomposition of $\mathbf{B} - \lambda\mathbf{I}$.

The second step is possible, because $\mathbf{B} - \lambda\mathbf{I}$ is positive semi-definite (because $\lambda$ is the smallest of all eigenvalues). Thus, the Cholesky decomposition can be performed, if full pivoting is done [Hig90].

Let $\lambda(B) = \{\lambda_1, \lambda_2, \lambda_3\}$ with $0 < \lambda_1 \leq \lambda_2 \leq \lambda_3$. Then, $\lambda(\mathbf{B} - \lambda\mathbf{I}) = \{0, \lambda_2 - \lambda_1, \lambda_3 - \lambda_1\}$. Let $\mathbf{B} - \lambda\mathbf{I} = \mathbf{USU}^\mathsf{T}$ be the singular value decomposition. Then, $x^\mathsf{T}(\mathbf{B} - \lambda\mathbf{I})x = x^\mathsf{T}\mathbf{USU}^\mathsf{T}x = y^\mathsf{T}\mathbf{S}y \geq 0$.

In our experience, this method is faster than the Jacobi method by a factor of 4, and it is faster than singular value decomposition by a factor 8.

### 4.3.4 Dynamic Point Clouds

Because of their lack of inherent connectivity, point-based models seem to be very suitable in dynamic settings.[8] In this section, we discuss the computational overhead for updating our additional data structures in dynamic settings.

In the worst case, inserting or deleting a point $\mathbf{p}_i$ in the general Delaunay graph or SIG can change $\Theta(n)$ edges. However, we can bound that number by $O(1)$, because we are only interested in paths up to some length $mr(P)$ (see Section 4.3.1). The following lemma summarizes our results.

[8] Note that this advantage vanishes if, for instance, hierarchical data structures are used to accelerate the rendering.

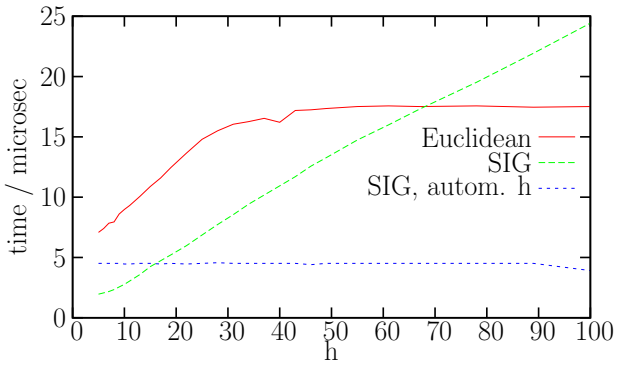



**Figure 4.15:** Average evaluation time of $f(\mathbf{x})$ depending on the kernel bandwidth $h$. The timings for $SIG(\mathcal{P})$ and $DG(\mathcal{P})$ are nearly identical (therefore, we omit one of them). Please note that our implementation is not yet fully optimized. Timing was done on a 1 GHz Pentium 3.

**Lemma 6.** *Given an arbitrary proximity graph where each node has only a constant number of neighbors. Then, our CPSP map and the corresponding graph can be updated in $O(\log n)$ time, if a point is inserted. Deletion can be performed in constant time.*

*Proof:* As we are only interested in paths up to some length $mr(P)$, the neighbor relations have to be updated for a constant number of points, namely for the points inside the sphere $S_i$ centered at $\mathbf{p}_i$ and with radius $mr(P)$. Thereby, not only edges inside the sphere have to be added or removed, also edges to points outside the sphere can change. Of course, those edges have also only a maximum length of $mr(P)$, so that the number of edges, which have to be modified, remains constant. That means, we have to update the graph inside a sphere $S_i'$ centered at $\mathbf{p}_i$ with radius $2mr(P)$.
In the case of inserting a point, the computation of the point set inside $S_i'$ causes an additional factor of $O(\log n)$ using a Delaunay hierarchy or BBD tree, respectively, as we have to perform a nearest neighbor search in the graph to find the starting point
After updating the proximity graph, all affected entries in our CPSP map have to be recomputed. The paths in $S_i$ can easily be updated by an APSP algorithm on the set $\mathcal{P} \cap S_i$. As already shown in Section 4.3.1, $\mathcal{P} \cap S_i$ (which has constant size) can be determined by a simple depth-first or breath-first search in time $O(1)$, and therefore, our CPSP map and the corresponding graph can be updated in time $O(\log n)$.

■

Note that the modified k-d tree for finding the nearest neighbors also has to be

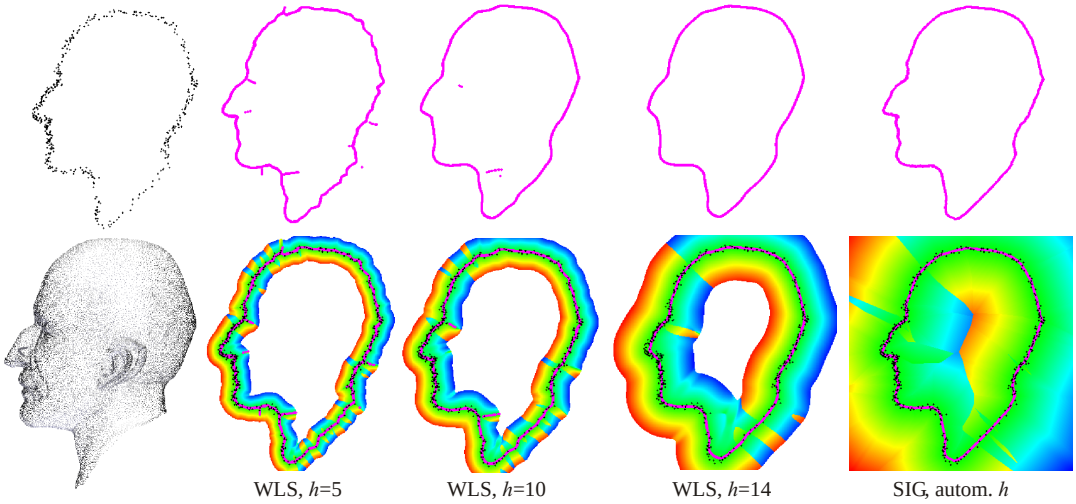



**Figure 4.16:** Reconstructed surface based on WLS and our new surface definition (rightmost) for a noisy point cloud obtained from the 3D Max Planck model (leftmost). Notice how our new definition, including automatic bandwidth detection, is able to handle fine detail, as well as sparse sampling, without manual tuning.

updated. This can be done in time $O(\log n)$ [AMN+98].

## 4.4 Results

We implemented our new point cloud surface definition in C++. It is easy to implement and can be evaluated very quickly: once the graphs are built, we can evaluate $f(\mathbf{x})$ simply by finding the nearest neighbor, traversing the graph, computing a number of weights from the CPSP table, and finally one eigenvector by Cholesky decomposition. Note that we have not implemented the idea of smoothing the surface using B-splines (Section 4.2.7), the sphere-of-influence complex and the anisotropic SIG.

First of all, Figure 4.15 shows the performance that can be achieved using our new surface definition for a reasonable choice of $h$. Although our implementation is not fully optimized, the performance is of the same order as that of the Euclidean kernel.

Figure 4.16, 4.18 and 4.19 illustrate the quality depending on the Euclidean kernel and our new geodesic one, respectively. Moreover, in order to give a numerical hint of the quality, we determined the root mean square error (RMSE)

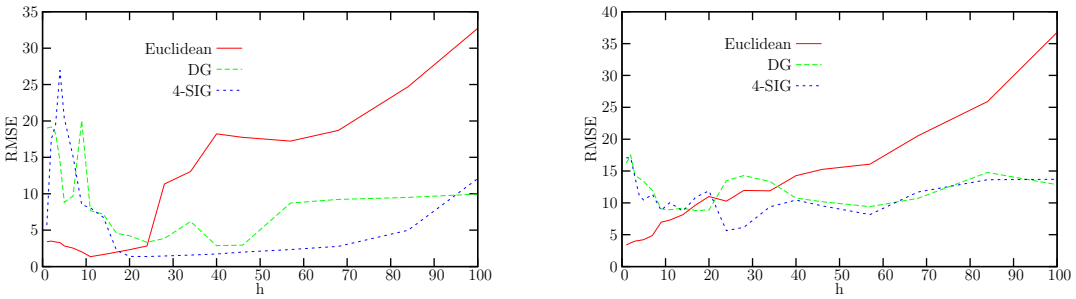
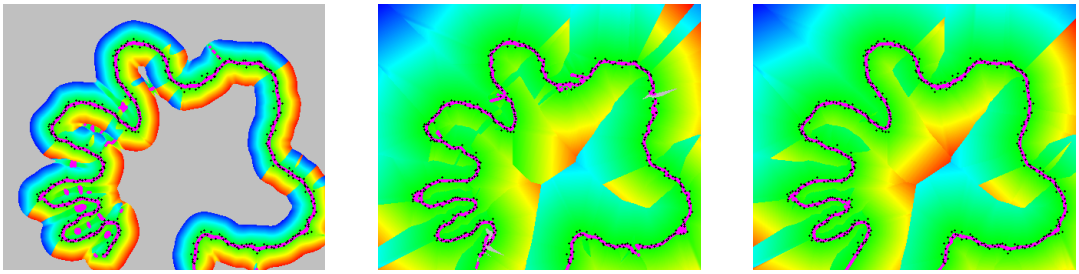



**Figure 4.17:** RMSE depending on the bandwidth, $h$, of the kernel. Our new kernel is less sensitive to the choice of $h$ than the old one. Refer to Figure 4.16 and Figure 4.18 for the corresponding models.



plain WLS | pruned DG | pruned 4-SIG

**Figure 4.18:** Comparison of the Euclidean and our new geodesic kernel for a noisy point cloud. In each example, the bandwidth was chosen so that the RMSE is minimal.

for the deviation (i.e., distance) of the reconstructed surface from the original surface

$$\sqrt{\frac{1}{|\hat{S}|}\sum_{\mathbf{x}\in\hat{S}} f(\mathbf{x})^2},$$

where $\hat{S}$ denotes a sampling of the original surface.
Obviously, our geodesic kernel approximates the surface very well, while the Euclidean kernel produces several artifacts. Even when the bandwidth $h$ (see Equation 4.3) is chosen optimally with respect to the RMSE, the Euclidean kernel produces severe artifacts (see Figure 4.18).
We also performed experiments to assess the sensitivity of our surface definition with respect to the kernel bandwidth $h$. The plots in Figure 4.17 (left and center) show that our new kernel is less sensitive to the choice of $h$ than the old one for two different example surfaces: for a large range of the bandwidth, the RMSE using our new surface definition is quite low. In contrast, the Euclidean

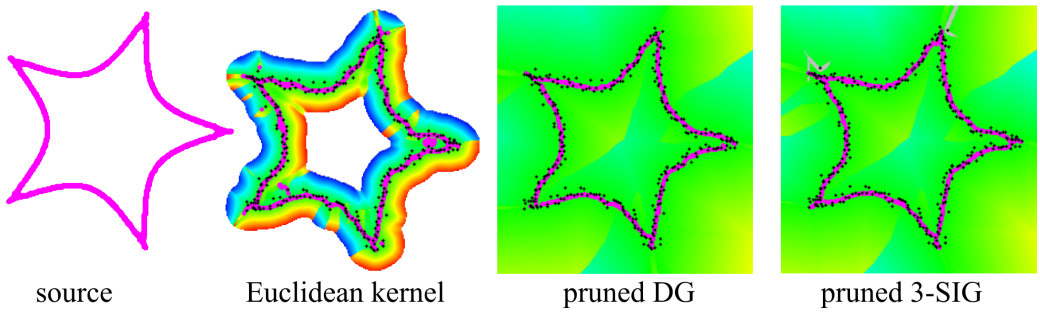



**Figure 4.19:** Artifacts at cusps (where the surface is not $C^1$ continuous) can also be avoided by our new kernel.

kernel yields a relatively low RMSE only for a small bandwidth. Note that in almost every case, the RMSE of the Euclidean kernel is larger than the RMSE of our new kernel. Note further, that in most cases the minimal RMSE of our new definition is clearly smaller than the minimal RMSE of the old one. However, sometimes the minimal RMSE of the Euclidean kernel is smaller than or new one, although the surface quality of the Euclidean kernel is lower. Thus, it should be thought about a better error measurement, e.g., the Hausdorff distance.

It might seem that there is still one parameter in our new approach, which requires fine-tuning, namely $r$, the radius of our modified sphere-of-influence graph $r$-SIG. However, numerous measurements for different point clouds suggest that $r \in [3 \dots 6]$ seems to be a good choice for all models.

## 4.5 Summary and Discussion

We proposed a geodesic distance function $d_{\mathsf{geo}}(\mathbf{p}, \mathbf{q})$ for implicit surfaces defined over point clouds. Our main idea is to approximate geodesic distances by shortest path in a geometric proximity graph or complex.

This distance function can be evaluated in time $O(\log n)$ for arbitrary points (if the points are not on the surface, we determine their nearest neighbor on an edge or triangle of the graph or complex, respectively). Furthermore, we also showed that the implicit function $f(\mathbf{x})$ using our geodesic kernel can be determined in $O(\log n)$ time.

The results in Section 4.4 showed that artifacts in the surface can be reduced dramatically in all cases. The automatic determination of the local bandwidth allows for defining surfaces over noisy and irregularly sampled point clouds. The results are much better than using a fine-tuned, but fixed bandwidth. For well-

sampled point clouds, the automatic determination is as good as fine-tuning, and in practice it saves a lot of time. For objects with boundaries, we can efficiently use our automatic sampling density estimation in order to detect the boundaries.
One important factor why point clouds have become such a popular shape representation is their property of simple and fast updates with respect to insertion, deletion or moving single points. Thus, point clouds constitute a good basis for deformable objects. As a consequence, it is very important that our auxiliary data structure (CPSP map) can also be updated very efficiently. We have shown that our CPSP table can be updated in time $O(\log n)$ for inserting or moving points. If points are deleted, the CPSP map can be updated in constant time.

## 4.6 Future Work

Although we have proposed some ideas to reduce discontinuities in the surface, an unsolved problem is that we cannot guarantee that the surface is continuously differentiable under all circumstances. For the kind of collision detection we are dealing with, the $C^\infty$ is not really essential. Rather it is important for us that $f(\mathbf{x})$ gives a good approximation of the distance from $\mathbf{x}$ to the surface, which can only be achieved if no artifacts occur in the surface. However, for collision handling and collision response, smooth surfaces with consistent normals are desirable. Furthermore, in the area of CAD systems, high order continuous surfaces are sometimes required [Cha01]. Therefore, it would be very appealing if a slight modification of our new definition could solve this problem.
Moreover, we believe that the approximation of geodesic distances can be improved. This could probably be achieved by alternative paths in the proximity graphs, e.g., by defining a new "long path" with the restriction of a certain curvature. In the 3D case, the shortest or "long" path can also be determined on the 2-simplices of the sphere-of-influence complex instead of computing them only on the edges.

# 5 Point Cloud Collision Detection

Given two point clouds $\mathcal{P}_A$ and $\mathcal{P}_B$, the goal is to determine whether or not there is an intersection, i.e., a common root $f_A(\mathbf{x}) = f_B(\mathbf{x}) = 0$, and, possibly, to compute a sampling of the intersection curve(s), i.e., of the set

$$\mathcal{Z} = \{\mathbf{x} \in \mathbb{R}^3 \mid f_A(\mathbf{x}) = f_B(\mathbf{x}) = 0\}.$$

Finding common roots of two (or more) nonlinear functions is extremely difficult [PFTV93]. Even more so here, because the functions are not described analytically, but algorithmically.
One could just utilize one of the many general root finding algorithms [PFTV93, PKKG03]. However, by exploiting the special structure and additional knowledge of the problem, we can make this decision much faster.
In the following, we present novel algorithms and data structures to check whether or not there is a collision between two point clouds. The algorithms treat the point clouds as a representation of an implicit function that approximates the point cloud, as described in Chapter 4.
Note that we never explicitly reconstruct the surface. Thus, we avoid the additional storage overhead and additional error that would be introduced by a polygonal reconstruction.
In the first part of this section, we present a novel algorithm for constructing point hierarchies by repeatedly choosing a suitable subset. This incorporates

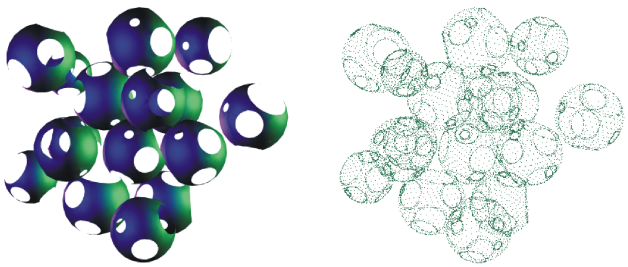


**Figure 5.1:** Given the implicit surfaces over point clouds, we would like to determine if there is a collision between them and/or would like to compute a sampling of the intersection curve(s). Right: point cloud models.

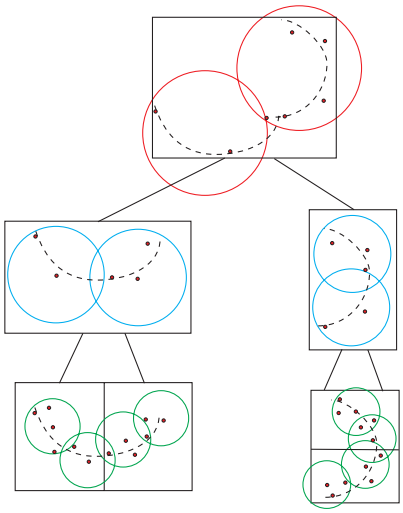


**Figure 5.2:** Our approach constructs a point hierarchy where each node stores a sample of the points underneath, which yields different levels of detail of the surface. In addition, we store a sphere covering of the surface of each node. Note that we compose a sphere covering of many more spheres.

a hierarchical sphere covering, the construction of which is motivated by a geometrical argument.

This hierarchy allows us to formulate two criteria that guide the traversal to those parts of the tree where a collision is more likely. That way we obtain a *time-critical* algorithm that returns a "best effort" result should the time budget be exhausted. In addition, the point hierarchy makes it possible for the application to specify a maximum "collision detection resolution" instead of a time budget. At the leaves, a pair of nodes is interrogated by a number of test points.

In the second part, we propose an approach that exploits spatial coherence of the surface in a different way, namely by its topology, captured with the help of geometric proximity graphs. Our approach can be used without any additional BVH, so that it fits very well with deformable point clouds. The running time can be bounded by $O(\log n)$ for deformable point clouds and by $O(\log \log n)$ for rigid ones. It is well understood, that it can also be combined with the method proposed in the first part. For instance, the first approach could determine close areas of the point clouds, while the second one actually determines intersections.

```
traverse(A, B)
if simple BVs of A and B do not overlap then
    return
if sphere coverings do not overlap then
    return
if A and B are leaves then
    return approx. distance between surfaces inside
for all children A_i and B_j do
    compute priority of pair (A_i, B_j)
traverse(A_i, B_j) with largest priority first
```

Algorithm 6: Outline of our hierarchical algorithm for point cloud collision detection.

## 5.1 Terms and Definitions

For the sake of accuracy and conciseness, we introduce the following definitions.

**Definition 9** (Bounding volumes $A, B$)**.** *We treat the terms bounding volume (BV) and node of a hierarchy synonymous. $A$ and $B$ will always denote BVs of two different hierarchies. The hierarchies are built over two point clouds $\mathcal{P}_1$ and $\mathcal{P}_2$.*

**Definition 10** (Cloud points $\mathcal{P}_A, \mathcal{P}_B$)**.** *Given a point cloud $\mathcal{P}_1$, the subset $\mathcal{P}_1 \cap A$ is denoted as $\mathcal{P}_A$. Moreover, for a second point cloud $\mathcal{P}_2$, we define $\mathcal{P}_B = \mathcal{P}_2 \cap B$. Of course, if $A$ and $B$ are the root BVs of $\mathcal{P}_1$ and $\mathcal{P}_2$, then $\mathcal{P}_A = \mathcal{P}_1$ and $\mathcal{P}_B = \mathcal{P}_2$.*

**Definition 11** (Sample points $\mathcal{P}'_A, \mathcal{P}'_B$)**.** *A sample of the point set $\mathcal{P}_A$ is denoted as $\mathcal{P}'_A$ ($\mathcal{P}'_A \subset \mathcal{P}_A$). Analogous, $\mathcal{P}'_B$ is a sample of $\mathcal{P}_B$.*

**Definition 12** (Test points)**.** *A test point is an arbitrary point, that is not necessarily contained in a given point cloud.*

## 5.2 An Efficient Point Cloud Hierarchy Traversal

Given two point clouds $\mathcal{P}_A$ and $\mathcal{P}_B$, we pursue a hierarchical approach to quickly determine points $\mathbf{x}$ such that $f_A(\mathbf{x}) = f_B(\mathbf{x}) = 0$ by exploiting the spatial knowledge of the surface.

The idea of our algorithm is to create a hierarchy where the points are stored in its leaves. At each inner node, we store a sample of the point cloud underneath, a simple BV (such as a box), and a sphere covering for the part of the surface corresponding to the node (see Figure 5.2). The point cloud samples effectively represent a simplified surface, while the sphere coverings define a neighborhood around it, that contains the original surface.
The sphere coverings, on the one hand, can be used to quickly eliminate the possibility of an intersection of parts of the surface. The simplified point clouds, on the other hand, together with the sphere coverings, can be used to determine kind of a likelihood of an intersection between parts of the surface.
Given two such point cloud hierarchies, the corresponding objects can be tested for collision by simultaneous traversal (see Algorithm 6), controlled by a priority queue. For each pair of nodes that still needs to be visited, our algorithm tries to estimate the likelihood of a collision, assigns a priority, and descends first into those pairs with largest priority.
If the traversal reaches two leaves, we have to test whether there is a collision between the implicit surfaces inside. For that purpose, we propose two approaches: a simple and easy to implement randomized sampling technique (RST, Section 5.2.3) and an interpolation search (iSearch, Section 5.3) on the proximity graph over the point cloud.

### 5.2.1 Point Cloud Hierarchy

In this section, we will describe a method to construct a hierarchy of point sets, organized as a tree, and a hierarchical sphere covering of the surface. This hierarchy is used by our collision detection algorithm.
In order to make our point hierarchy memory efficient, we do not compute an optimal sphere covering, nor do we compute an optimal sample for each inner node. Instead, we combine both of them, so that the sphere centers are also the sample.
In the first step, we construct a binary tree where each leaf node is associated with a subset of the point cloud. In order to do this efficiently, we recursively split the set of points by a top-down process. We create a leaf when the number of cloud points is below a threshold. We store a suitable BV with each node to be used during the collision detection process. Since we are striving for maximum collision detection performance, we should split the set so as to minimize the volume of the child BVs [Zac02].
Note that so far, we have only partitioned the point set and assigned the subsets to leaves.

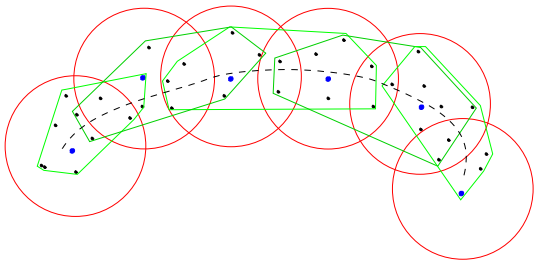


**Figure 5.3:** The set of convex hulls induced by the leaves underneath an inner node of our hierarchy can be covered by spheres thus obtaining a neighborhood around $P$ containing the surface.

Remember that we limit the region of influence of points by our CPSP map so that the surface is defined as

$$S = \big\{\mathbf{x} \in \mathbb{R}^3 \mid f(\mathbf{x}) = 0\,,\, \#\{\mathbf{p} \in P : \|\mathbf{p} - \mathbf{x}\| < r_\varepsilon\} > c\big\}, \tag{5.1}$$

where Equation 4.3 implies $r_\varepsilon = h \cdot \sqrt{|\log \theta_\varepsilon|}$. As a consequence, we have to assign some neighboring points $\mathbf{p}_i$ lying in the $r_\varepsilon$-border around $A$ to each node $A$. So, if $\mathbf{x} \in A$, we need to consider only the points inside the BV $A$ plus the points within its $r_\varepsilon$-border in order to evaluate the implicit function $f(\mathbf{x})$. Note that in the following, $\mathcal{P}_A$ denotes the points from $\mathcal{P}_1$ lying inside $A$ or its $r_\varepsilon$-border.
In the second step, we construct a simplified point cloud and a sphere covering for each level of our hierarchy. Actually, we will do this such that the set of sphere centers are exactly the simplified point cloud. One of the advantages is that we need virtually no extra memory to store the simplified point cloud.
In the following, we will derive the construction of a sphere covering for one node of the hierarchy, such that the centers of the spheres are chosen from the points assigned to the leaves underneath. In order to minimize memory usage, all spheres of that node will have the same radius. (This problem bears some relationship to the general mathematical problem of thinnest sphere coverings, see [CS93] for instance, but here we have different constraints and goals.)
More specifically, let $A$ be the node for which the sphere covering is to be determined. Let $A'_1, \ldots, A'_n$ be the leaves underneath $A$. Denote by $\mathcal{P}_{A'_i}$ all cloud points lying in $A'_i$ or its $r_\varepsilon$-border, and let $\mathrm{CH}(\mathcal{P}_{A'_i})$ be its convex hull. Let $\mathcal{P}_A = \bigcup \mathcal{P}_{A'_i}$.
For the moment, assume that the surface in $A$ does not have borders (such as intentional holes). Then,

$$\forall \mathbf{x} \in A'_i : \mathbf{a}(\mathbf{x}) \in \mathrm{CH}(\mathcal{P}_{A'_i}).$$

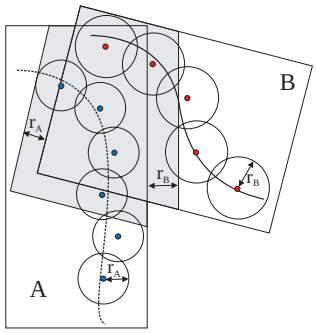



**Figure 5.4:** Using the BVs and sphere coverings stored for each node, we can quickly exclude intersections of parts of the surfaces.

Therefore, if $\mathbf{x} \in A$ and $f(\mathbf{x}) = 0$, then $\mathbf{x}$ must be in $H = \bigcup_i \mathrm{CH}(\mathcal{P}_{A'_i})$.
So instead of trying to find a sphere covering for the surface contained in $A$ directly, our goal is to find a set $K = \{K_i\}$ of spheres, centered at $\mathbf{k}_i \in \mathcal{P}_A$, and a common radius $r_A$, such that $\mathrm{Vol}(K) = \mathrm{Vol}(\bigcup K_i)$ is minimal, with the constraints that $K$ covers $H$, and bounded size $|K| \leq c_A$ (see also Figure 5.3). This problem can be solved by a fast randomized algorithm, which does not even need an explicit representation of the convex hulls (see below). (An exact solution would be an expensive combinatorial optimization problem over an exponential number of possible sets $\{\mathbf{k}_i\}$. In addition, constructing the convex hulls and minimal enclosing spheres are fairly expensive, too.) In Section 5.2.5, we will derive suitable bounds $c_A$ on the size of $K$.
Our randomized algorithm first tries to determine a "good" sample $\mathcal{P}'_A \subset \mathcal{P}_A$ as sphere centers $\mathbf{k}_i$, and then computes an appropriate $r_A$. In both stages, the basic operation is the construction of a random point within the convex hull of a set of points, which is trivial.

**Construction of $\mathcal{P}'_A$:**
The idea is to choose sample points $\mathbf{k}_i \in \mathcal{P}_A$ in the interior of $H$ so that the distances between them are of the same order. Then, a sphere covering using the $\mathbf{k}_i$ should be fairly tight and thin.
We choose a random point $\mathbf{q}$ lying in BV $A$; then, we find the closest point $\mathbf{p} \in \mathcal{P}_A$ (this is equivalent to randomly choosing a Voronoi cell of $\mathcal{P}_A$ with probability depending on its size); finally, we add $\mathbf{p}$ to the set $\mathcal{P}'_A$. We repeat this random process until $\mathcal{P}'_A$ contains the desired number of sample points (see Section 5.2.5). In order to obtain more evenly distributed $\mathbf{k}_i$'s, and thus a better $\mathcal{P}'_A$, we can use quasi-random number sequences.
Since we want to prefer random points in the interior over points close to the border of $H$, we compute $\mathbf{q}$ as the weighted average of *all* points $P_{A'_i}$ of a

randomly chosen $A'_i$.

**Determining $r_A$:**
Conceptually, we could construct the Voronoi diagram of the $\mathbf{k}_i$, intersect that with $H = \bigcup_i \mathrm{CH}(P_{A'_i})$, determine the radius for the remainder of each Voronoi cell, and then take the maximum. Since the construction of the Voronoi diagram in 3D takes $O(n^2)$ in the worst case ($n$ = number of sites) [dBvKOS00], we propose a method similar to Monte-Carlo integration as follows.
Initialize $r_A$ with 0. Generate randomly and independently test points $\mathbf{q} \in H$. If $\mathbf{q} \notin K$, then determine the minimal distance $d$ of $\mathbf{q}$ to $\mathcal{P}'_A$, and set $r_A = d$. Repeat this process until a sufficient number of test points has been found to be in $K$.
In other words, we continuously estimate

$$\frac{\mathrm{Vol}(K \cap H)}{\mathrm{Vol}(H)} \approx \frac{\#\text{ points} \in K \cap H}{\#\text{ points} \in H} \tag{5.2}$$

and increase $r_A$ whenever we find that this fraction is less than 1. In order to improve this estimate, we can apply kind of a stratified sampling: when $\mathbf{q} \notin K$ was found, we choose the next $r$ test points in the neighborhood of $\mathbf{q}$ (for instance, by a uniform distribution confined to a box around $\mathbf{q}$).

### 5.2.2 Exclusion and Priority Criterion

Utilizing the sphere coverings of each node, we can quickly eliminate the possibility of an intersection of parts of the surface (see Figure 5.4). Note that we do not need to test all pairs of spheres. Instead, we use the BV of each node to eliminate spheres that are outside the BV of the other node.
As mentioned above, we strive for a time-critical algorithm. Therefore, we need a way to estimate the likelihood of a collision between two inner nodes $A$ and $B$, which can guide our Algorithm 6.
Assume, for the moment, that the sample points in $A$ and $B$ describe closed manifold surfaces $f_A = 0$ and $f_B = 0$, respectively. Then, we could be certain that there is an intersection between $A$ and $B$, if we would find two points on $f_A$, that are on different sides of $f_B$.
Here, we can achieve only a heuristic. Assuming that the points $\mathcal{P}'_A$ are close to the surface, and that $f'_B$ is close to $f_B$, we look for two test points $\mathbf{p}_1, \mathbf{p}_2 \in \mathcal{P}'_A$ such that $f'_B(\mathbf{p}_1) < 0 < f'_B(\mathbf{p}_2)$ (Figure 5.5).
In order to improve this heuristic, we consider only test points $\mathbf{p} \in \mathcal{P}'_A$ that are outside the $r_B$-neighborhood around $f_B$, because this decreases the probability that the sign of $f_B(\mathbf{p}_1)$ and $f_B(\mathbf{p}_2)$ is equal.

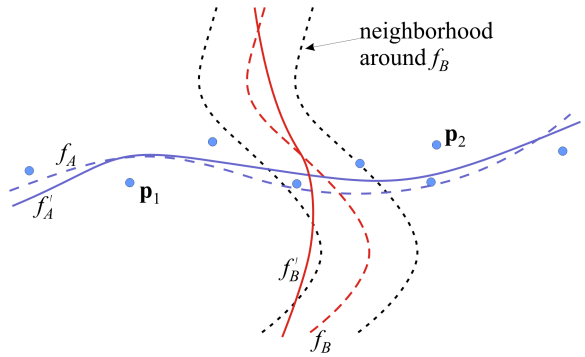



**Figure 5.5:** Using the sample of two nodes and their $r$-neighborhoods, we can efficiently determine whether or not an intersection among the two nodes is likely.

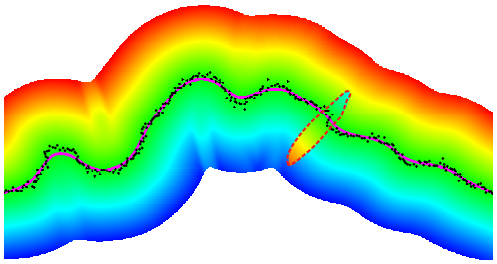


**Figure 5.6:** Visualization of the implicit function $f(\mathbf{x})$ over a 2D noisy point cloud (black dots). Points $\mathbf{x} \in \mathbb{R}^2$ with $f(\mathbf{x}) \approx 0$, are shown magenta. Red denotes $f(\mathbf{x}) \gg 0$ and blue denotes $f(\mathbf{x}) \ll 0$. The normal $\mathbf{n}(\mathbf{x})$ flips only across the red dashed line.

Overall, we estimate the likelihood of an intersection proportional to the number of points on both sides.

This argument holds only, of course, if the normal $\mathbf{n}_B(\mathbf{x})$ in Equation 4.1 does not "change sides" within a BV $B$. In our experience, fortunately, this appears to be rarely the case, in particular, if one uses the covariance matrix centered at $\mathbf{a}(\mathbf{x})$ as proposed in Equation 4.5 (see Figure 5.6).

From perturbation theory we know how changes in the matrix $\mathbf{B}$ (cf. Equation 4.5) affect the eigenvectors. Let $\lambda_i$ be the eigenvalues with right eigenvectors $\mathbf{x}_i$ of $\mathbf{B}$. Let matrix $\mathbf{F}, \|\mathbf{F}\|_2 = 1$, be a perturbation of $\mathbf{B}$. By a continuity argument, we have

$$(\mathbf{B} + \varepsilon\mathbf{F})\mathbf{x}_k(\varepsilon) = \lambda_k(\varepsilon)\mathbf{x}_k(\varepsilon)\ , \quad \|\mathbf{x}_k(\varepsilon)\|_2 = 1, \tag{5.3}$$

where $\mathbf{y}_i$ are the left eigenvectors. One can show [GvL89] that $\mathbf{x}_k(\varepsilon)$ behaves

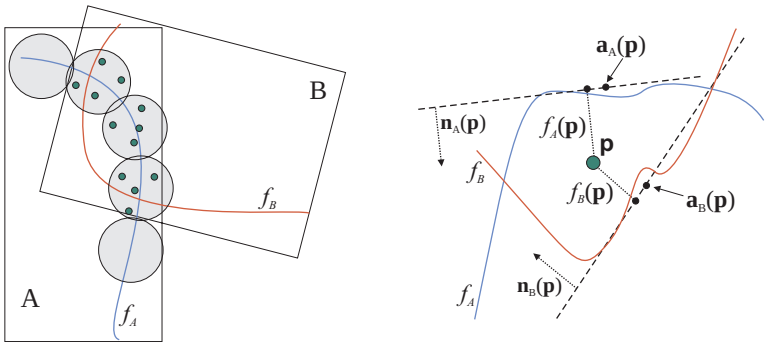



**Figure 5.7:** In order to efficiently estimate the distance between the surfaces contained in a pair of leaves, we generate a number of random test points (left) and estimate their distance from $A$ and $B$ (right).

like

$$\mathbf{x}_k(\varepsilon) \approx \mathbf{x}_k + \sum_{i=1, i\neq k}^{n} \frac{\mathbf{y}_i^\top \mathbf{F} \mathbf{x}_k}{(\lambda_k - \lambda_i)\mathbf{y}_i^\top \mathbf{x}_i} \mathbf{x}_i + O(\varepsilon^2) \tag{5.4}$$

Here, we are interested in the sensitivity of the smallest eigenvector, $\mathbf{x}_3$. We assume that $\lambda_3$ is singular. Then, a necessary condition for $\mathbf{x}_3$ to be "well behaving" is that its eigenvalue is well separated from the other two eigenvalues. In other words, if the smallest eigenvalue of $\mathbf{B}$ is well separated from the other two eigenvalues for all points $\mathbf{x} \in B$ (the BV), then we can hope that $\mathbf{n}(\mathbf{x})$ does not "flip", and $\mathbf{f}_B(\mathbf{x})$ will indeed assume consistently different signs on different sides.

### 5.2.3 RST: Randomized Sampling Technique

When the traversal has reached two leaf nodes, $A$ and $B$, we could just apply the traversal criterion again, and return an intersection, if it is met.
Ideally, however, we would like to find a test point $\mathbf{p}$ such that $f_A(\mathbf{p}) = f_B(\mathbf{p}) = 0$ (where $f_A$ and $f_B$ are defined over $\mathcal{P}_A$ and $\mathcal{P}_B$, respectively).
Note that it would not be sufficient to compute the distances between the points stored in $A$ and $B$, because an intersection point is not necessarily close to a cloud point.
In practice, such an intersection point cannot be found in a reasonable amount of time, so we generate randomly and independently a constant number of test points $\mathbf{p}$ lying in the sphere covering of object $A$ (see left of Figure 5.7). Then, we take

$$d_{AB} \approx \min_{\mathbf{p}} \{|f_A(\mathbf{p})| + |f_B(\mathbf{p})|\} \tag{5.5}$$

as an estimate of the distance of the two surfaces (see right of Figure 5.7), and report an intersection if $d_{AB} < d_\varepsilon$.
More precisely, if a precise collision point's distance from the surfaces is to be smaller than $2\epsilon$, we cover each sphere of the sphere covering of $A$ lying inside $B$ by $s$ smaller spheres with radius $\epsilon$. Then, we sample each sphere by $O(s \ln s)$ points so that each of the $s$ spheres gets a point with high probability (see Lemma 7, Section 5.3.1). For each of these, we just determine the distance to both surfaces.
Rogers [Rog63] showed that a sphere with radius $a \cdot b$ can be covered by at most $s = \lceil \sqrt{3} \cdot a \rceil^3$ smaller spheres of radius $b$. Since we would like to cover a sphere of radius $r_A$ by spheres with radius $b = \epsilon$, we have to choose $a = \frac{r_A}{\epsilon}$, so that $a \cdot b = r_A$. As a consequence,

$$s = \lceil \frac{\sqrt{3} \cdot r_A}{\epsilon} \rceil^3.$$

In order to obtain a better and faster estimate, we propose in Section 5.3 an approach that utilizes a geometric proximity graph for both, for the surface definition and for the collision detection process.

### 5.2.4 Time-Critical Collision Detection

The traversal prioritization and the leaf intersection test described above, facilitate a time-critical approach: on the one hand, if the time budget is exhausted, the collision detection process returns a "best effort" answer to the collision query (depending on the minimal distance between the objects that has been found so far). This is needed in time-critical applications where a real-time response is needed under all circumstances. On the other hand, if there is still time left, our algorithm can spend more time on the collision detection in leaf nodes to increase the accuracy.
This is done by trying to spend the same time $t_{\max}$ for each collision query by adjusting the number of test points and the distance $d_\varepsilon$ that has to be found between the objects (see Section 5.2.3). If the time needed is larger than $t_{\max}$, the number of test points is gradually decreased and $d_\varepsilon$ is increased, and vice versa otherwise.

### 5.2.5 Automatic Bandwidth Detection

Our algorithm has to evaluate $f(\mathbf{x})$ for subsets $\mathcal{P}'_A \subset P$, which have different sampling densities. As a consequence, we could automatically adjust the bandwidth $h$ to the sampling density as proposed in Section 4.2.5, so that no holes

appear higher up in our point cloud hierarchy. It is, of course, inevitable that intentional holes in the surface are closed at higher levels, but this just produces a few "false positives" during the traversal.
If, for any reason, our surface definition proposed in Section 4 is not used and, thus, the bandwidth cannot be estimated as in Section 4.2.5, we propose an alternative to determine the bandwidth automatically without any proximity graph. For that, we first extend the notion of sampling radius, introduced in Section 4.3, for a sample $\mathcal{P}'_A$.

**Definition 13** (Sampling radius of $\mathcal{P}'_A$)**.** *Let a point cloud $\mathcal{P}_A$ as well as a subset $\mathcal{P}'_A \subseteq \mathcal{P}_A$ be given. Consider a set of spheres, centered at $\mathcal{P}'_A$, that cover the surface defined by $\mathcal{P}_A$ (not $\mathcal{P}'_A$), where all spheres have equal radius. We define the sampling radius $r(\mathcal{P}'_A)$ as the minimal radius of such a sphere covering.*

Moreover, we can define the sampling radius for the set $\mathcal{P}_A$ as a special case with $\mathcal{P}'_A = \mathcal{P}_A$ in the definition above. And, as a special case of that, we define the sampling radius $r(\mathcal{P})$ of the whole point cloud $P$, where $A$ is the root BV in the definition above. The last definition is equal to that proposed in Section 4.3.
Since our surfaces do not interpolate the point cloud, we do not use the notion of *$\varepsilon$-sampling* [ACDL00]. In addition, we believe our definition is more practical. Given $r(\mathcal{P}'_A)$, we can determine the bandwidth $h$ such that points up to a distance of about $m$ times the sampling radius will have an influence in Equation 4.1, if $\mathbf{x}$ is close to the surface:

$$h = \sqrt{-\frac{(r(\mathcal{P}'_A) \cdot m)^2}{\log \theta_\varepsilon}} \tag{5.6}$$

with $\theta_\varepsilon < 1$. This follows from Equation 4.3 and Equation 5.1 where we restrict the horizon of influence of points by our CPSP map.
Obviously, we could plug in $r_A$ as sampling radius $r(\mathcal{P}'_A)$ at inner nodes. However, this can be an overestimate, because the spheres, as constructed in Section 5.2.1, could cover intentional holes, which results in an imprecise $h$.
Alternatively, we estimate $r(\mathcal{P}'_A)$ by

$$r(\mathcal{P}'_A) = \sqrt{\frac{|\mathcal{P}_A|}{|\mathcal{P}'_A|}} \cdot r(\mathcal{P}_A) \tag{5.7}$$

which is explained in detail in Section 5.3.2 in a very similar context.

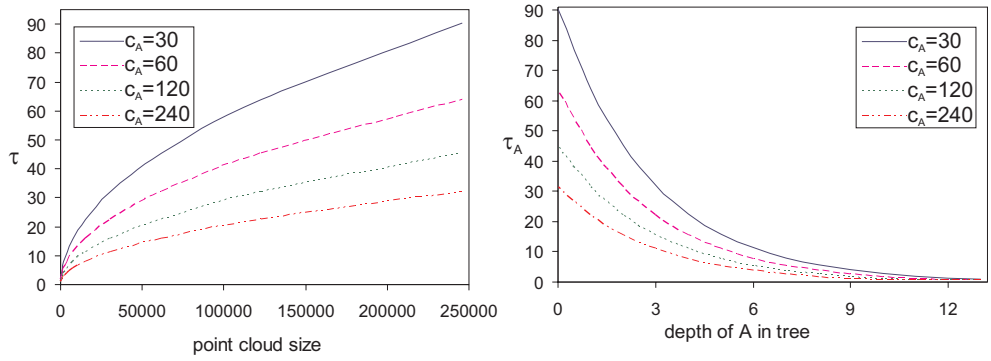



**Figure 5.8:** Left: $\tau = \sqrt{\frac{|\mathcal{P}|}{c_A}}$ depending on the point cloud size $|\mathcal{P}|$ for different *sampling densities* $c_A = |\mathcal{P}'_A|$. Right: $\tau_A$ depending on the depth of $A$ in the binary tree. In this plot, we restrict the depth of the tree by 13 and the number of points per leaf by $|\mathcal{P}_{A'_i}|$.

### 5.2.6 Sample Size

A criterion for the number of sample points to be stored in each node is the quality of the corresponding surface. Obviously, the quality depends on the sampling radius: the smaller the sampling radius, the fewer errors occur in the surface (a comparison of the traversal time depending on the quality of the surface can be found in Section 5.2.8).
Using Equation 5.7, we can derive an estimate for the number $c_A$ of sample points needed in a node $A$ in order to achieve a radius $r(\mathcal{P}'_A)$ that is only $\tau_A$ times larger than $r(\mathcal{P}_A)$

$$\begin{aligned} & r(\mathcal{P}'_A) \leq \tau_A \cdot r(\mathcal{P}_A) \\ \Rightarrow & \sqrt{\frac{|\mathcal{P}_A|}{|\mathcal{P}'_A|}} \cdot r(\mathcal{P}_A) \leq \tau_A \cdot r(\mathcal{P}_A) \\ & \overset{c_A = |\mathcal{P}'_A|}{\Rightarrow} c_A \geq |\mathcal{P}_A| / {\tau_A}^2. \end{aligned} \tag{5.8}$$

As a consequence, given a point cloud $\mathcal{P}$, the sampling radius is at most $\tau \cdot r(\mathcal{P})$ throughout the point hierarchy, if $c_A = |P|/\tau^2$ points per node are stored. Then, the largest sampling radius $\tau \cdot r(\mathcal{P})$ can be found in the root node, while the sampling radius in the leaves is $r(\mathcal{P})$. For instance, if the sampling radius in every node is to be at most $50 r(\mathcal{P})$ and the point cloud consists of 75 000 points, then at most $c_A = 30$ points per node need to be stored.

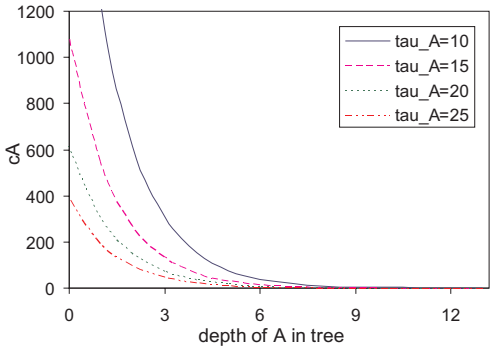



**Figure 5.9:** If $\tau_A$ is constant throughout the hierarchy, the quality of the reconstructed surface is roughly the same for different nodes (of same curvature). Then, the number $c_A = |\mathcal{P}'_A|$ of sample points depends on the depth of $A$ in the tree. The deeper $A$ in the hierarchy, the fewer points have to be stored.

Figure 5.8 (left) shows the dependence of $\tau_A$ on the point cloud size for different choices of $c_A$. Moreover, the right plot in Figure 5.8 shows $\tau_A$ depending on the depth of a node $A$ for different numbers $c_A = |\mathcal{P}'_A|$ of sample points per node

$$\tau_A = \sqrt{\frac{2^{t_{max}-t_A} \cdot |\mathcal{P}_{A'_i}|}{c_A}},$$

where $t_{max}$ denotes the maximum depth of the point cloud hierarchy, $t_A$ denotes the depth of $A$ and $|\mathcal{P}_{A'_i}|$ denotes the number of points per leaf (we assume that each leaf node contains the same number of points).
Of course, if $\tau_A$ changes, the quality of the reconstructed (parts of) surfaces may also change. To avoid this, we can choose $\tau_A = \tau$ constant throughout the hierarchy and adjust the number of sample points for each node depending on its depth

$$c_A = \frac{2^{t_{max}-t_A} \cdot |\mathcal{P}_{A'_i}|}{\tau^2}.$$

Then, the quality of the reconstructed surfaces is roughly the same for different nodes of same curvature. The corresponding plot is shown in Figure 5.9. Of course, we always use $\tau_A = 1$ for a leaf node $A$, in order to miss no intersections because of imprecise reconstructed surfaces.

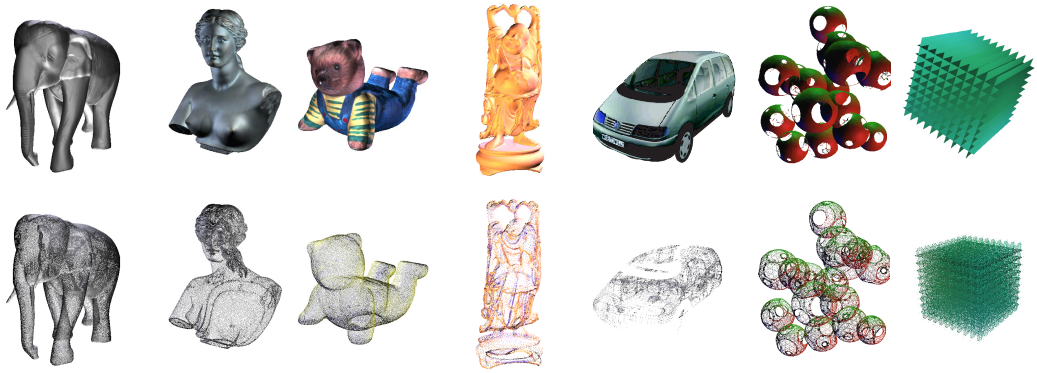



**Figure 5.10:** Some of the models of our test suite, by courtesy of (left to right): Polygon Technology Ltd, Stanford, Volkswagen. The two artificial models (spheres and grid) show that our approach works well with non-closed geometry, too. The numbers are the sizes of the respective point clouds.

| | Car | Buddha | Aphrodite | Elephant | Grid |
|---|---|---|---|---|---|
| # cloud points | 35,056 | 62,299 | 89,036 | 148,689 | 197,315 |
| # sample points | 45,012 | 90,068 | 134,376 | 180,180 | 360,404 |
| avg. depth of a node | 11 | 12 | 13 | 13 | 14 |

**Table 5.1:** Comparison of the number of cloud points and sample points as well as the average depth of a node in the hierarchy (only objects with appreciably different values are listed).

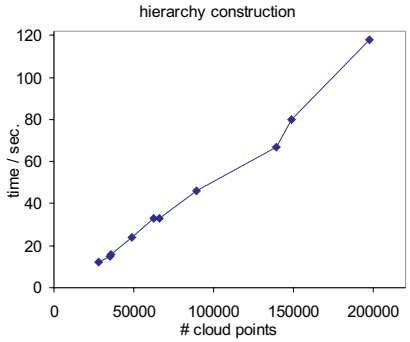



**Figure 5.11:** This plot shows the build time of our point cloud hierarchies for various objects.

### 5.2.7 Running time and Complexity

The complexity of constructing our point cloud hierarchy is in $O(n \log n)$, where $n$ is the number of points. Our algorithm first constructs a hierarchy where the points are only stored at its leaves. This hierarchy is built so as to minimizes the volumes of the child BVs [Zac02]. The running time for constructing this (preliminary) hierarchy in a top down manner is in $O(n \log n)$ [Zac02]. Note that during the construction we can easily assign the points in the $r_\varepsilon$-border to each node using two splitting planes with distance $r_\varepsilon$ to the original one. As, for a given node, the number of points in the $r_\varepsilon$-border is constant, our modified construction causes no extra costs.
Then, for each of the $O(n)$ nodes of the hierarchy, we construct the samples $\mathcal{P}'_A$ and the radius $r_A$. Thereby, for each node we generate a constant number of points and determine their nearest neighbors which can be done in $O(\log n)$, e.g., by using a modified k-d tree [AMN+98]. Overall, the samples and the radius can also be determined in time $O(n \log n)$.
As we perform a BVH traversal where the costs at each leaf node are constant, the running time of our approach can be analyzed by the technique proposed in Section 3.7.

### 5.2.8 Results

We implemented our new algorithms in C++. To date, the implementation is not fully optimized. In the following, all results were obtained on a 2.8 GHz Pentium-IV.
For timing the performance and measuring the quality, we have used a set of objects (see Figure 5.10), most of them with varying complexities (with

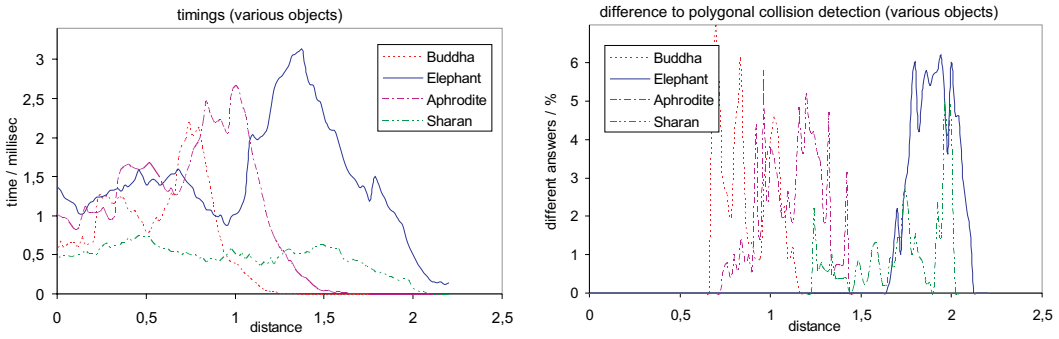



**Figure 5.12:** Left: timings for different objects. Right: differences to polygonal collision detection of the objects; note that the polygonal models are *not* a tessellation of the true implicit surface, but just a mesh of the point cloud. The results for the teddy are very similar to that of the car, and are therefore omitted.

respect to the number of points). Benchmarking is performed by the procedure proposed in Section 3.8.1 which computes average collision detection times for a range of distances between two identical objects.

### Hierarchy Construction

Our point cloud hierarchies can be built in a fairly short time, so that the construction can be performed at startup time (see Figure 5.11).

The memory usage of a hierarchy is low: with each node, we store a BV, a pointer to the child nodes, one float for the radius $r_A$, a constant number $c_1$ of pointers to the sample or to the cloud points lying in $A$, and also a number $c_2$ of pointers to points in the $r_\varepsilon$-border of the BV. That means, we need $32+4(c_1+c_2)$ bytes for each node. In practice, $c_1+c_2$ is between 15 and 30 so that at most 150 bytes per node is needed. For example, the hierarchy of our largest model (the grid) consisting of about 65,000 nodes consumes about 9 MB main memory. Of course, we also have to store the cloud points in main memory. Table 5.1 gives an overview of the number of sample points stored at inner nodes as well as the average depth of a node in the hierarchy.

### Time and Quality

Each plot in Figure 5.12 (left) and Figure 5.13 (left) shows the average running time for a model of our test suite, which is in the range 0.5–2.5 millisec (the two artificial models are considered later in this section). This makes our new algorithm suitable for real-time applications, and, in particular, physically-based simulation in interactive applications. Using our time-critical approach, the

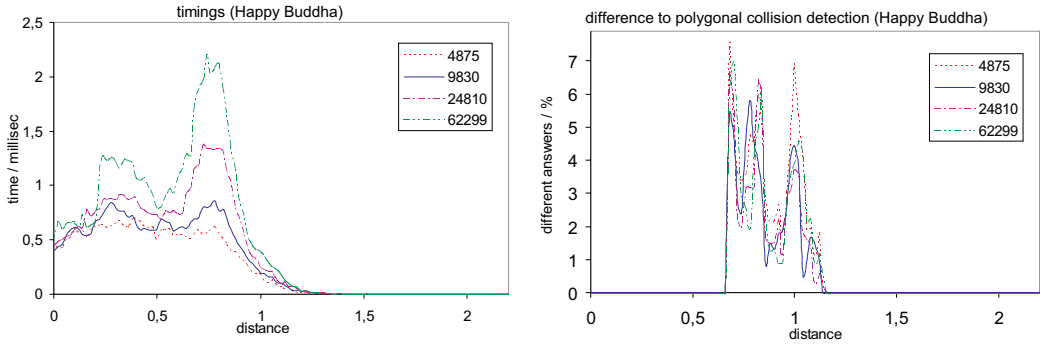



**Figure 5.13:** Left: timings for different object complexities (# points) of the happy buddha model. Right: differences to polygonal collision detection of the objects.

detection time can be decreased even further (e.g., when there are too many collisions going on). Details are given at the end of this section.

For each object of our test suite, we have also compared the outcome of our new algorithm with a traditional polygonal collision detection using a very high-resolution polygonal model. That way, we can give some experimental hints about the error probability of our new algorithm. Note that the polygonal models are *not* a tessellation of the true implicit surface, but just a tessellation of the given point cloud. The results in Figure 5.12 (right) and Figure 5.13 (right) show that the difference is always relatively low on average. For distances between 0.6 and 2, about 1.2% (happy buddha), 1.06%(elephant), 1.20% (aphrodite) and 0.64% (car) different answers are reported. Here, only collision tests were considered where at least the root BVs intersect. The differences can be explained by two facts: first, the implicit surface defined by the vertices of a polygonal object is obviously different from the polygonal model. Second, our intersection finding algorithm in the leaf nodes is very simplistic at the moment (an improvement is given in Section 5.3).

Equivalent measurements for our two artificial models can be found in Figure 5.14. Note that the models have boundaries, and that the spheres model consists of several unconnected components. Obviously, our approach achieves results as good as for the other models.

Note that for large distances between the two grid models the running time decreases, but the difference to the polygonal collision detection increases. It is obvious, to spend more time in such cases (by generating more test points) which is explained in the following.

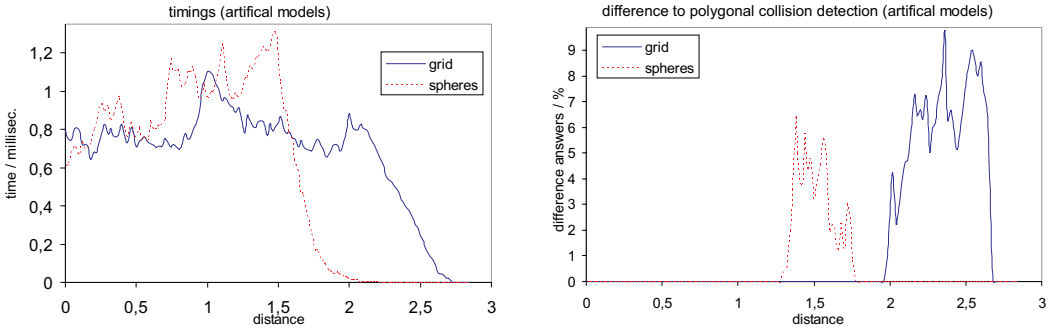



**Figure 5.14:** Timings and difference to polygonal collision detection of artificial models. The differences can be reduced by increasing the number of test points if the time budget is not exhausted (see Figure 5.15).

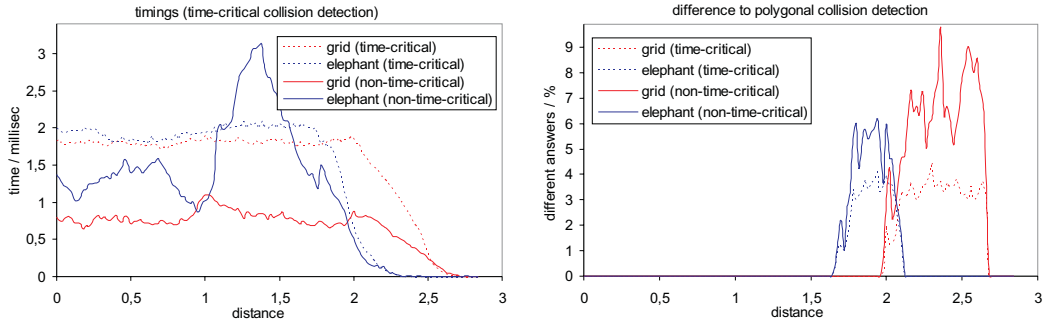



**Figure 5.15:** Timings and differences using both the time-critical and non-time-critical algorithms. The differences measure not errors but the number of different reports from the point-based versus the polygonal collision detection algorithms.

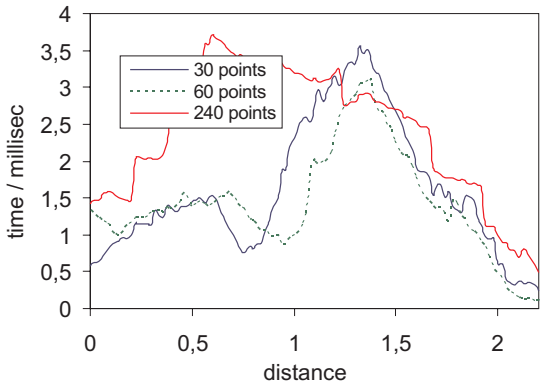



**Figure 5.16:** Running time for different sample sizes using our Elephant model (about 150,000 points): 30 points ($\tau = 70$), 60 points ($\tau = 50$), 240 points ($\tau = 25$). The best performance is achieved for $\tau = 50$. Note that the leaves constitute always the whole point cloud.

### Time-Critical Collision Detection

The results using our time-critical extension can be found in Figure 5.15. Obviously, it always spends almost the same amount $t_{\max}$ of time on the collision detection. Therefore, it can adapt the number of test points in the leaves of the point hierarchy much better.

Note further that the measured average collision query time is sometimes lower than $t_{\max}$, because sometimes a result can be achieved earlier, and sometimes the traversal does not reach any leaf nodes.

### Running Time depending on Sample Size

We found out, that the running time behaves similar for different objects, if the surfaces are of the same quality (with respect to the sampling radius). That means, if choosing the sample size at inner nodes using a fixed $\tau$ (so that the sampling radius is smaller than $\tau r(\mathcal{P})$ throughout the hierarchy), similar results are achieved for different hierarchies.

For example, if the surface quality at inner nodes is very high, then the traversal time is also very high. Note that a high quality at inner nodes will not improve the quality of the collision detection. Moreover, if the sample size is chosen too small, then during traversal a lot of "false positives" are produced. As a consequence, priority is given to regions of the objects where no collision takes place. Figure 5.16 illustrates the situation and shows that for $\tau = 50$ the best results are achieved. We have obtained similar results for all our other objects.

## 5.3 Interpolation Search for Point Cloud Intersection

The approach proposed in the previous section can be applied very well to rigid point clouds. However, if the point clouds are deformable, updating our point cloud hierarchies at runtime would probably be too expensive because changes in the leaves have to be propagated to upper nodes (including the calculation of bounding boxes, bounding spheres and the computation of the points sets, that approximate the surface).

In general, non-hierarchical data structures seem to be more promising for collision detection for deformable objects [ABG+01, HMB01, FL01], although some geometric data structures suggest a natural BV hierarchy [LCLL02].

Furthermore, it would be desirable to have an approach that exploits the topology of the surfaces in order to determine the intersection points more quickly. In contrast, the RST approach proposed in the previous section generates test points independently of the two surfaces inside the intersection volume of two leaf nodes.

In the following, we propose an approach that needs no BV hierarchy at all. It builds only on top of a geometric proximity graph, e.g., on the sphere-of-influence graph that can be updated in time $O(\log n)$. Therefore, it constitutes the basis for collision detection for arbitrary deformable point clouds. Moreover, it can also be applied to any kind of hierarchical data structure to increase the performance for the intersection tests at the leaves. If a constant number of intersection points is sufficient, our approach has a running time of $O(\log n)$. For non-deformable point clouds, the running time can be bounded by only $O(\log \log n)$, because the proximity graph does not need to be updated.

An outline of our approach is given in Figure 5.17. First, our algorithm tries to bracket intersections by two points on one surface and on either side of the other surface. Second, for each such bracket, it finds an approximate point in one of the point clouds that is close to the intersection (see Figure 5.18). We utilize our proximity graph and perform an interpolation search along the shortest path between each pair of bracket points for this. Finally, this approximate intersection point is refined by subsequent sampling. This last step is optional, depending on the accuracy needed by the application.

In the following, we describe each step in detail.

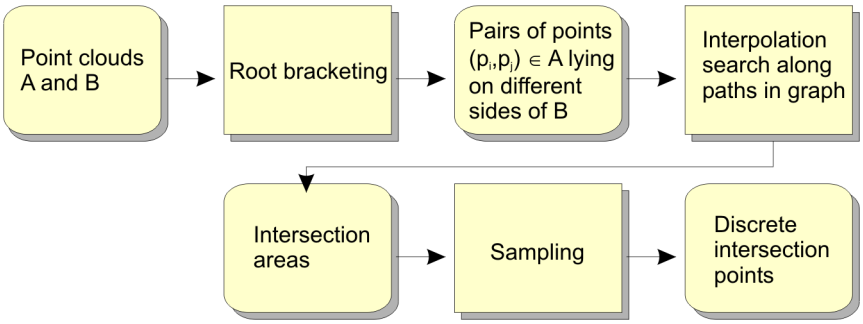



**Figure 5.17:** Outline of our point cloud collision detection using an interpolation search. The sampling of the intersection areas (second row) can optionally be used if the collision points are to be determined very precisely.

### 5.3.1 Root Bracketing

Our algorithm starts by constructing pairs of points on different sides of one of the surfaces. Here, we have the following goals in order to implement a collision detection that is as fast and accurate as possible:

- The pairs should evenly sample the surface.
- The two points should not be too far apart from each other.
- An explicit spatial data structure should be avoided.
- The number of brackets should always be bounded by a constant number.

Only if the pairs are evenly distributed over the surface, *all* intersection points with respect to a certain sampling density can be found. If two points of such a pair are too far apart from each other, we have no local control over the resulting intersection points. Moreover, the consistency of the normals defined by the weighted least squares approach cannot be guaranteed over large distances. Of course, we do not want to use an explicit spatial (hierarchical) data structure so that our approach can be used for deformable point clouds. The last point, a constant number of brackets, is necessary in order to guarantee an overall running time of $O(\log n)$ for deformable point clouds and a running time of $O(\log \log n)$ for rigid objects.
An exhaustive enumeration of all pairs is, of course, prohibitively expensive. Therefore, we propose the following randomized (sub-)sampling procedure.
Let $Vol(A \cap B)$ denote the intersection volume of two point clouds $\mathcal{P}_A$ and $\mathcal{P}_B$. Further, let $\overline{\mathcal{P}_A} = \mathcal{P}_A \cap Vol(A \cap B)$ denote the points from $\mathcal{P}_A$ lying in the intersection.

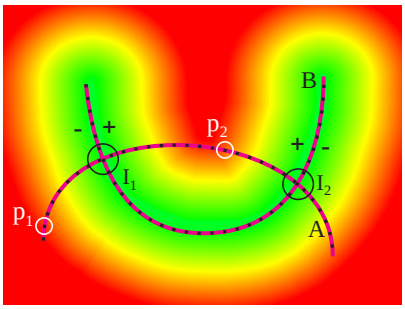



**Figure 5.18:** Two point clouds $\mathcal{P}_A$ and $\mathcal{P}_B$ and their intersection spheres $I_1$ and $I_2$. Our root finding procedure, when initialized with $\mathbf{p}_1, \mathbf{p}_2 \in \mathcal{P}_A$, will find an approximate intersection point inside the *intersection sphere* $I_1$.

Now, assume that the implicit surface is conceptually approximated by surfels (2D discs) of equal size [PvBZG00, RL00] (see also Figure 5.19b). Then, we want to randomly draw points $\mathbf{p}_i \in \overline{\mathcal{P}_A}$ such that each surfel $s_i$ gets occupied by at least one $\mathbf{p}_i$. Here, "occupied by $\mathbf{p}_i$" means that the projection of $\mathbf{a}(\mathbf{p}_i)$ along the normal $\mathbf{n}(\mathbf{p}_i)$ onto the supporting plane of $s_i$ lies within the surfel's radius. These randomly chosen points $\mathcal{P}'_A$ constitute the candidate points for the root brackets (Figure 5.19c).
Then, for each candidate $\mathbf{p}_i \in \mathcal{P}'_A$ we determine another point $\mathbf{p}_j \in \mathcal{P}'_A$ (if any) in the *neighborhood* of $\mathbf{p}_i$ so that $\mathbf{p}_i$ and $\mathbf{p}_j$ lie on different sides of $f_B$ (described in the next sections). We represent the neighborhood of a point $\mathbf{p}_i$ by a sphere $S_i$ centered at $\mathbf{p}_i$. Note that $\mathbf{p}_j$ is chosen only from the candidate points.
An advantage of this is, that the application can specify the density of the intersection points, which are reported by our algorithm. From these, it is fairly easy to construct a discretization of the complete intersection curves (e.g., by utilizing randomized sampling again).
Note that we never need to actually construct the surfels, or assign the points from $\mathcal{P}_A$ explicitly to the neighborhoods, which we describe in the following Section 5.3.2.
In order to sample $\mathcal{P}_A$ such that each (conceptual) surfel is represented by at least one point in the sample, we use the following

**Lemma 7.** *Let $\mathcal{P}_A$ be a uniformly sampled point cloud. Further, let $S_A$ denote the set of conceptual surfels approximating the surface of $\mathcal{P}_A$ inside the intersection volume of $\mathcal{P}_A$ and $\mathcal{P}_B$, and let $a = |S_A|$. Then, in order to occupy each surfel with at least one point with probability $p = \frac{1}{e^{(e^{-c})}}$, where $c$ is an arbitrary constant, we have to draw $N = O(a \ln a + c \cdot a)$ random and independent points from $\overline{\mathcal{P}_A}$. These points are denoted as $\mathcal{P}'_A$.*

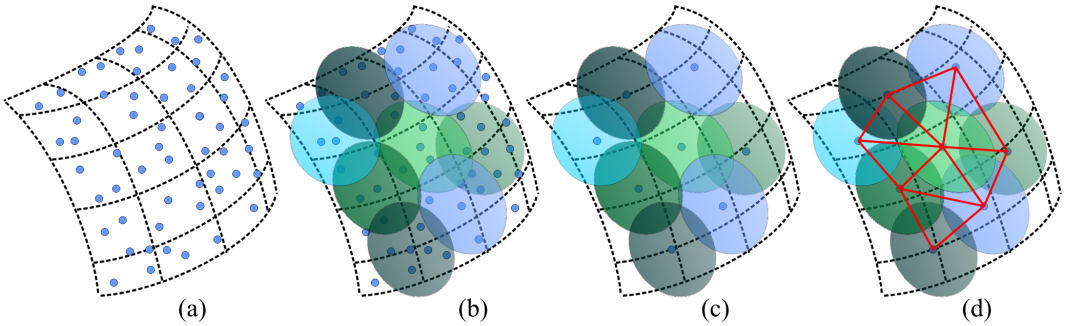



**Figure 5.19:** (a) An implicit surface defined over a point cloud. (b) The implicit surface can be approximated by surfels of the same size. (c) Candidate points for the root bracketing can be found by occupying each surfel by at least one point from point cloud $\mathcal{P}_A$. (d) Pairs of candidate points are tested whether they lie on different sides of the surface defined by $\mathcal{P}_B$. These pairs can be obtained by a sphere-of-influence graph over the candidate points.

Proof: We can reduce the problem to a simple urn model. Given $a$ bins (corresponding to the number of surfels), how many balls (corresponding to the number of points to be drawn) have to be thrown i.i.d. into the bins so that every bin gets at least one ball with high probability?

Let $X$ denote the number of drawings required to put at least one ball into each bin. It is well known that the expectation value of $X$ is $a \cdot H_a$, where $H_a$ is the $a$-th harmonic number [MR95, p. 57f].

Let $c$ be an arbitrary constant. The $a$-th harmonic number is about $\ln a \ \pm 1$ which is asymptotically sharp, and so $c \cdot a$ additional balls are enough to fill each bin with probability $p$ which depends on $c$. Overall, $N = a \ln a + c \cdot a$ balls have to be thrown.

To compute the dependence of $p$ on $c$, we refer to the proof given by Motwani and Raghavan [MR95, p. 61ff]. They showed that the probability $p = Pr[X \leq N]$ can be determined by $p = \frac{1}{e^{(e^{-c})}}$ for a sufficiently large number of bins. ∎

For instance, if we want $p \geq 97\%$, we have to choose $c = 3.5$, and if $a = 30$, then $N \approx 200$ random points have to be drawn.

The next section will show how to choose an appropriate size for the neighborhoods $S_i$, or, if a constant size of the neighborhoods is desired, how to choose $a$, the number of surfels, properly. After that, Section 5.3.3 will propose an efficient way to determine the other part $\mathbf{p}_j$ of the root brackets, given a point $\mathbf{p}_i \in \mathcal{P}'_A$.

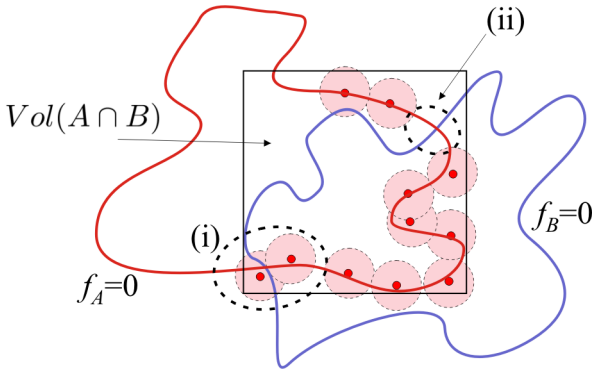



**Figure 5.20:** If the spherical neighborhoods $S_i$ (red) are too small, not all collisions can be found. (i) adjoining neighborhoods do not overlap sufficiently, their intersection contains no randomly chosen cloud point. (ii) surface is not covered by neighborhoods $S_i$.

### 5.3.2 Size of Neighborhoods and Surfel Density

One of our goals for the root bracketing is that the number of brackets can always be bounded by a constant number. There are two possibilities to fulfill this requirement:

- We set the number of surfels to be constant under all circumstances. As a consequence, the size (radius) of the neighborhoods has to be adapted according to the intersection volume of the two point clouds. The larger the intersection volume, the larger the neighborhoods have to be chosen.
- We set the size of the neighborhoods to be constant. Therefore, the surfel density (the number of surfels inside the intersection volume) has to be adapted to the intersection volume. The larger the intersection volume, the more surfels have to be chosen.

#### Constant number of surfels

If the number $a$ of surfels has to be constant for an arbitrary constellation of the objects and, thus, for an arbitrary intersection volume, the radius of the spherical neighborhoods $S_i$ has to be chosen so that, on the one hand, all $S_i$ cover the whole surface defined by $\mathcal{P}_A$. On the other hand, the intersection with each adjoining neighborhood of $S_i$ has to contain at least one candidate point in $\mathcal{P}'_A$ so as to not miss any collisions lying in the intersection of two neighborhoods. The situation is illustrated in Figure 5.20.
The minimal radius of a spherical neighborhood $S_i$ can be determined by the *sampling radius* of the sample points $\mathcal{P}'_A$ (see Definition 13). It is easy to see

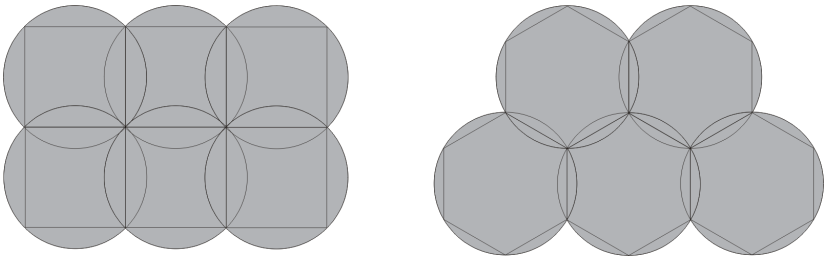


**Figure 5.21:** Left: simple arrangement of surfels so that no holes are between them. Right: a comb-shaped arrangement reduces the number of surfels needed to cover the implicit surface.

that spheres with radius $2r(\mathcal{P}'_A)$ centered at points in $\mathcal{P}'_A$ contain always points of the neighboring spheres and, of course, cover the surface. Therefore, the radius of $S_i$ is set to $2r(\mathcal{P}'_A)$.
The sampling radius $r(\mathcal{P}'_A)$ can obviously be estimated as the radius $r_s$ of a surfel $s_i \in S_A$, the set of surfels approximating the surface of $\mathcal{P}_A$.
Let $F_A$ denote the surface area of the implicit surface over $\overline{\mathcal{P}_A}$ and let us assume $\mathcal{P}_A$ is a uniformly sampled point cloud. Then, the surfel radius $r_s$ can be determined by

$$\frac{F_A}{a} \approx \pi {r_s}^2 \;\Rightarrow\; r_s \approx \sqrt{\frac{F_A}{\pi a}}. \tag{5.9}$$

However, the surfels cannot cover the surface without overlapping each other (which was assumed in Equation 5.9). Of course, we could say that each surfel covers only the area of the largest square that fits into a given surfel (Figure 5.21, left). Then, a surfel with radius $r_s$ covers only $2{r_s}^2$ area instead of $\pi {r_s}^2$. A better covering would require a comb-shaped arrangement of the surfels as shown in Figure 5.21 (right). In that case, each surfel covers the area of the largest hexagon lying inside, namely $3/2\sqrt{3}{r_s}^2 \approx 2.6{r_s}^2$ surface area. That means, a better approximation can be given by

$$r_s \approx \sqrt{\frac{F_A}{2.6a}}.$$

Assume that the implicit surface over $\overline{\mathcal{P}_A}$ can also be approximated by surfels of size $r(\mathcal{P}_A)$. Then, $F_A$ can be estimated by

$$F_A \approx |\overline{\mathcal{P}_A}| \cdot 2.6r(\mathcal{P}_A)^2$$

or by

$$F_A \approx \frac{Vol(A \cap B) \cdot \text{surface area of } \mathcal{P}_A}{Vol(A)},$$

where $Vol(A \cap B)$ denotes the intersection volume of $A$ and $B$. Overall, $r(\mathcal{P}'_A)$ can be estimated by $r_s$

$$r_s \approx \sqrt{\frac{|\overline{\mathcal{P}_A}| \cdot r(\mathcal{P}_A)^2}{a}} \approx \sqrt{\frac{\frac{\text{Vol}(A \cap B)}{\text{Vol}(A)}|\mathcal{P}_A|}{a}} \cdot r(\mathcal{P}_A) \tag{5.10}$$

if both $\mathcal{P}_A$ and $\mathcal{P}'_A$ do not contain significant discrepancies (i.e., the local sampling radius does not vary too much). The size of $\overline{\mathcal{P}_A}$ can be estimated depending on the ratio of $\text{Vol}(A)$ and $\text{Vol}(A \cap B)$, the sampling radius $r(\mathcal{P}_A)$ can easily be determined in the preprocessing.

**Constant size of neighborhoods**
As shown, the size of a neighborhood can be seen as the diameter of a single surfel from $S_A$. That means, if the size of the neighborhood is chosen constant under all circumstances, the number $a$ of surfels has to be adjusted, so that they cover the implicit surface defined by $\mathcal{P}_A$ inside the intersection volume.
Using Equation 5.10, the number $a$ can easily be determined as

$$a \approx \frac{\frac{\text{Vol}(A \cap B)}{\text{Vol}(A)}|\mathcal{P}_A| \cdot r(\mathcal{P}_A)^2}{{r_s}^2}. \tag{5.11}$$

### 5.3.3 Completing the Brackets

Given a candidate point $\mathbf{p}_i \in \mathcal{P}'_A$, we have to determine other points $\mathbf{p}_j \in \mathcal{P}'_A \cap S_i$ on the other side of $f_B$ in order to bracket the intersections. For that, we use $f_B(\mathbf{p}_i) \cdot f_B(\mathbf{p}_j) \leq 0$ as an indicator. This, of course, is reliable only if the normals $\mathbf{n}(\mathbf{x})$ are consistent throughout space. If the surface is manifold, this can be achieved by a method similar to [HDD+92].
Utilizing our proximity graph (which is a supergraph of the nearest-neighbor graph), we can propagate a normal to each point $\mathbf{p}_i \in \mathcal{P}_A$. Then, when defining $f(\mathbf{x})$, we choose the direction of $\mathbf{n}(\mathbf{x})$ according to the normal stored with the nearest neighbor of $\mathbf{x}$ in $\mathcal{P}_A$.[1]

[1] Surprisingly, the direction of $\mathbf{n}(\mathbf{x})$ is consistent over fairly large volumes without any preconditioning.

From a theoretical point of view, finding points on the other side could be done by testing $f_B(\mathbf{p}_i) \cdot f_B(\mathbf{p}_j) \leq 0$ for all points $\mathbf{p}_j \in \mathcal{P}'_A \cap S_i$ in time $O(1)$, because $|\mathcal{P}'_A|$ can be bounded by a constant if $a$ is chosen constant as proposed in Section 5.3.2. In practice however, the sets $\mathcal{P}'_A \cap S_i$ cannot be determined quickly. Therefore, in the following, we propose two adequate alternatives that work in time $O(\log \log n)$ and time $O(1)$, respectively.

**Sampling the anulus around $\mathbf{p}_i$**

We observe that $\mathcal{P}'_A \cap S_i \approx \mathcal{P}'_A \cap A_i$, where $A_i := \{\mathbf{x} \mid 2r_s - \delta \leq \|\mathbf{x} - \mathbf{p}_i\| \leq 2r_s\}$ is an *anulus* around $\mathbf{p}_i$ (or, at least, these are the $\mathbf{p}_j$ that we need to consider to ensure a certain bracket density). By construction of $\mathcal{P}'_A$, $\mathcal{P}'_A \cap A_i$ has a similar distribution as $\mathcal{P}_A \cap A_i$. Observe further, that we do not necessarily need $\mathbf{p}_j \in \mathcal{P}'_A$.

Overall, the idea is to construct a random sample $P''_A \subset \mathcal{P}_A \cap S_i$ such that $P''_A \subset A_i$, $|P''_A| \approx |\mathcal{P}'_A \cap A_i|$, and such that $P''_A$ has a similar distribution as $\mathcal{P}'_A \cap A_i$. Then, we test all points from that set $P''_A$, whether they lie on different sides.

The sample $P''_A$ can be constructed quickly by the help of Lemma 7: we just choose randomly $O(b \ln b)$ many points from $\mathcal{P}_A \cap A_i$, where $b := |\mathcal{P}'_A \cap S_i|$. As one can see, the anulus is used to restrict the point set $\mathcal{P}_A \cap S_i$ from which we generate our approximate set $\mathcal{P}'_A \cap S_i$. As a consequence, fewer points have to be considered, and, the more important thing, $P''_A$ contains no points from $\mathcal{P}_A - A_i$, which are usually not contained in $\mathcal{P}'_A \cap S_i$.

We can describe the set $\mathcal{P}_A \cap A_i$ very quickly, if the points in the CPSP map stored with $\mathbf{p}_i$ are sorted by their geodesic[2] distance from $\mathbf{p}_i$. Then, we just need to use interpolation search to find the first point with distance $2r_s - \delta$ and the last point with distance $2r_s$ from $\mathbf{p}_i$. This can be done in time $O(\log \log |\mathcal{P}_A \cap S_i|)$ per point $\mathbf{p}_i \in \mathcal{P}'_A$. Thus, the overall time to construct all brackets is in $O(\log \log n)$.

**Completing the brackets utilizing the sphere-of-influence graph**

Remember, that for a candidate point $\mathbf{p}_i \in \mathcal{P}'_A$, we first want to determine all points $\mathbf{p}_j \in \mathcal{P}'_A \cap S_i$ (after that, we can easily test all that points, whether they lie on the other side compared to $\mathbf{p}_i$). Note further that the radius of the spherical neighborhoods $S_i$ is always the same for a fixed constellation of the objects. That means, the points $\mathbf{p}_j \in \mathcal{P}'_A \cap S_i$ can exactly be determined by the help of a sphere-of-influence graph. More precisely, after drawing the $|\mathcal{P}'_A|$ candidate points from the intersection volume, we construct a SIG over the set $\mathcal{P}'_A$ (Figure 5.19d) where all the spheres have the same radius $r_s$. Then, for each

[2] By using the geodesic distance (or, rather, the approximation thereof) we basically impose a different topology on the space, where $\mathcal{P}_A$ is embedded, but this is actually desirable.

$\mathbf{p}_i \in \mathcal{P}'_A$ we test all points adjacent to $\mathbf{p}_i$, whether they lie on the other side. As the size of set $\mathcal{P}'_A$ can be bounded by a constant (see Section 5.3.2), the SIG over $\mathcal{P}'_A$ can be constructed in constant time and, as a consequence, the overall time to construct all brackets is in $O(1)$.

### 5.3.4 Interpolation Search

Having determined two points $\mathbf{p}_i, \mathbf{p}_j \in \mathcal{P}_A$ on different sides of surface $\mathcal{P}_B$, the next goal is to find a point $\hat{\mathbf{p}} \in \mathcal{P}_A$ "between" $\mathbf{p}_i$ and $\mathbf{p}_j$ that is "as close as possible" to $\mathcal{P}_B$. In the following, we will call such a point *approximate intersection point* (AIP). The true intersection curve $f_B(\mathbf{x}) = f_A(\mathbf{x}) = 0$ will pass close to $\hat{\mathbf{p}}$ (usually, it does not pass through any points of the point clouds). Depending on the application, $\hat{\mathbf{p}}$ might already suffice. If the true intersection points are needed, then we refine the output of the interpolation search by the procedure described in Section 5.3.6.
Here, we can exploit the proximity graph: we just consider the points $P_{1m}$ that are on the shortest path between $\mathbf{p}_i$ and $\mathbf{p}_j$, and we look for $\hat{\mathbf{p}}$ that assumes $\min_{\mathbf{p}\in P_{ij}}\{|f(\mathbf{p})|\}$.
Instead of doing an exhaustive search along the path, we can utilize interpolation search to look for $\hat{\mathbf{p}}$ with $f(\hat{\mathbf{p}}) = 0$.[3] This makes sense here, because the "access" to the key of an element, i.e., an evaluation of $f_B(\mathbf{x})$, is fairly expensive [Sed89]. The average running time of interpolation search is in $O(\log\log m)$, where $m$ is the number of points along the shortest path. For that, $f_B$ has to be monotonic along the path $\overline{\mathbf{p}_i\mathbf{p}_j}$. The pivot element $\mathbf{p}_x$ between two points $\mathbf{p}_l$ and $\mathbf{p}_r$ lying on different sides can be interpolated by

$$x = l + \lceil \frac{-d_l}{d_r - d_l}(r - l) \rceil.$$

Algorithm 7 for our interpolation search assumes that the shortest paths are precomputed and stored in the CPSP map (Section 4.3.1). Analogously to Lemma 1, it is easy to see that the storage is still linear.
However, in practice, memory usage for storing the shortest paths could be too large for huge point clouds. In that case, we can compute the path $P$ on the fly at runtime by Algorithm 8. Theoretically speaking, the overall algorithm is now in linear time. However, in practice, it still behaves sublinear, because the reconstruction of the path is negligible compared to evaluating $f_B$ (see Section 5.3.8). Note that in the case our root finding algorithm returns two points

[3] In practice, the interpolation search will never find exactly such a $\hat{\mathbf{p}}$, but instead a pair of adjacent points on the path that straddle $B$

```
iSearch(P)
l, r = 1, m
d_{l,r} = f_B(P_1), f_B(P_m)
while |d_l| > ε and |d_r| > ε and l < r do
    x = l + ⌈ (−d_l)/(d_r − d_l) (r − l) ⌉ {*}
    d_x = f_B(P_x)
    if d_x < 0 then
        l, r = x, r
    else
        l, r = l, x
return p_l, p_r
```

Algorithm 7: Pseudo-code of our root finding algorithm based on interpolation search. $P$ is an array containing the points of the shortest path from $\mathbf{p}_i = P_1$ to $\mathbf{p}_j = P_m$, which can be precomputed. $d_i = f_B(P_i)$ approximates the distance of $P_i$ to object $B$. (*) Note that either $d_l$ or $d_r$ is negative.

*far* away from each other, so that their geodesic distance is not included in the CPSP map, we just have to use the Euclidean distance measure in Algorithm 8. Of course, we can also integrate the path finding algorithm into the interpolation search (see Algorithm 9) to improve the result for irregularly sampled point clouds. Using this algorithm, we can find a pivot element $\mathbf{p}_x$ by interpolating the geodesic distance between $\mathbf{p}_i$ and $\mathbf{p}_j$ depending on their approximate distances to the surface defined by $\mathcal{P}_B$. In contrast, in Algorithm 7 we only interpolate the indices of the array $P$ depending on the distances of $\mathbf{p}_i$ and $\mathbf{p}_j$ to the surface over $\mathcal{P}_B$. That means, Algorithm 7 will achieve its best results (with respect to the running time) for regularly sampled point clouds while Algorithm 9 fits also very well to noisy or irregularly sampled point sets.

If $f_B$ is not monotonic along the paths between the brackets, but the sign of $f_B(\mathbf{x})$ is consistent, then we can utilize binary search to find $\hat{\mathbf{p}}$. The complexity in that case is $O(\log m)$.

If the sign of $f_B$ is not consistent, but $f_B$ is bitonic (that means, $|f_B|$ is monotonic), we could utilize the golden section search [PFTV93, p. 397f]. In that case, we can find the minimum, if there is only a single minimum between $p_1$ and $p_2$. Otherwise, we cannot ensure that the golden section search has not found only a local minimum. The golden section search guarantees that each new function evaluation will bracket the minimum to an interval just 0.61803 times the size of the preceding interval. This is comparable to, but not quite as good as, the 0.5 that holds when finding the roots by bisection.

**shortestPath($\mathbf{p}_i, \mathbf{p}_j$)**
$q$.insert($\mathbf{p}_i$); clear $P$
$\mathbf{p} = \mathbf{p}_i$
**while** $\mathbf{p} \neq \mathbf{p}_j$ and $q$ is not empty **do**
  $\mathbf{p} = q$.pop
  $P$.append( $\mathbf{p}$ )
  **for all** $\mathbf{p}_k$ adjacent to $\mathbf{p}$ **do**
    **if** $d_{\text{geo}}(\mathbf{p}_k, \mathbf{p}_j) < d_{\text{geo}}(\mathbf{p}_i, \mathbf{p}_j)$ **then**
      insert $\mathbf{p}_k$ into $q$ with priority $d_{\text{geo}}(\mathbf{p}_k, \mathbf{p}_j)$

Algorithm 8: This algorithm can be used to initialize $P$ for Algorithm 7 if storing all shortest paths in the CPSP map is too expensive. ($q$ is a priority queue where priority is given to points with lowest geodesic distance.)

**iSearch2($\mathbf{p}_i, \mathbf{p}_j$)**
$\mathbf{p}_l = \mathbf{p}_i, \mathbf{p}_r = \mathbf{p}_j$
insert $p_l$ into $q$
$d_l = f_B(\mathbf{p}_l), d_r = f_B(\mathbf{p}_r)$
**while** $|d_l| > \epsilon$ and $|d_r| > \epsilon$ and $q$ is not empty **do**
  $\mathbf{p}_x = q$.pop
  **if** $d_{geo}(\mathbf{p}_x, \mathbf{p}_r) \leq |d_r|/(|d_l| + |d_r|) \cdot d_{geo}(\mathbf{p}_l, \mathbf{p}_r)$ **then**
    **if** $d_l \cdot d_x \leq 0$ **then**
      $\mathbf{p}_r = \mathbf{p}_x, d_r = f_B(\mathbf{p}_r)$
    **if** $d_r \cdot d_x \leq 0$ **then**
      $\mathbf{p}_l = \mathbf{p}_x, d_l = f_B(\mathbf{p}_l)$
  **else**
    **for all** $\mathbf{p}_k$ adjacent to $\mathbf{p}_x$ **do**
      **if** $d_{geo}(\mathbf{p}_k, \mathbf{p}_r) < d_{geo}(\mathbf{p}_l, \mathbf{p}_r)$ **then**
        insert $\mathbf{p}_k$ into $q$ with priority $d_{geo}(\mathbf{p}_k, \mathbf{p}_r)$
**return** $\mathbf{p}_l, \mathbf{p}_r$

Algorithm 9: This root finding algorithm determines a (subset of the) shortest path from $\mathbf{p}_i$ to $\mathbf{p}_j$ on the fly. As a consequence, we can find a pivot element $\mathbf{p}_x$ by interpolating the geodesic distance between $p_i$ and $p_j$ depending on their approximate distances to the surface of $\mathcal{P}_B$.

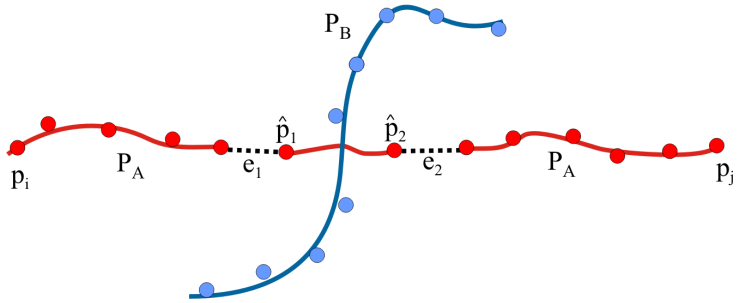



**Figure 5.22:** Models with boundaries can cause errors. The AIPs $\hat{\mathbf{p}}_1$ and $\hat{\mathbf{p}}_2$ could remain undetected, which can be avoided by "virtual" edges $e_1$ and $e_2$ in the proximity graph.

### 5.3.5 Models with Boundaries

If the models have boundaries and the sampling rate of our root bracketing algorithm is too low, not all intersections can always be found (see Figure 5.22). In that case, some AIPs might not be reached, because they are not connected through the proximity graph.
Therefore, we propose to modify the $r$-SIG. After constructing the graph, we usually prune away all "long" edges by an outlier detection algorithm (see Section 4.2.2 and Section 4.2.4). Now, we only mark these edges as "virtual". Thus, we can still use the $r$-SIG for defining the surface as before. For our interpolation search, however, we can also use the "virtual" edges so that small holes in the model are bridged (Figure 5.22).

### 5.3.6 Precise Intersection Points

If two point clouds are intersecting, our interpolation search computes a set of AIPs. More precisely, it computes a set of pairs of AIPs, where each pair consists of two points $\hat{\mathbf{p}}_1, \hat{\mathbf{p}}_2 \in \mathcal{P}'_A$ with $f_B(\hat{\mathbf{p}}_1) \cdot f_B(\hat{\mathbf{p}}_2) \leq 0$. A sphere containing those two points of radius

$$r = \max(\|\mathbf{x} - \hat{\mathbf{p}}_1\|, \|\mathbf{x} - \hat{\mathbf{p}}_2\|)$$

centered at a point $\mathbf{x}$

$$\mathbf{x} = \hat{\mathbf{p}}_1 + \lambda(\hat{\mathbf{p}}_2 - \hat{\mathbf{p}}_1), \quad \lambda = \frac{d_1}{d_1 + d_2}$$

contains also a true intersection point, where $d_i = f_B(\mathbf{p}_i)$. We call this sphere an *intersection sphere*.

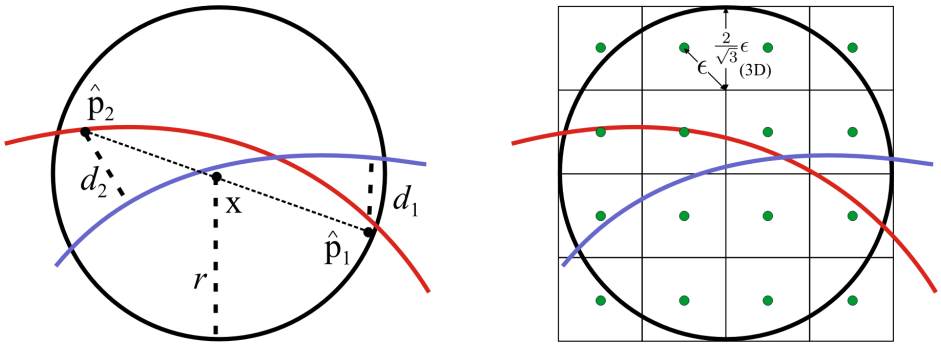



**Figure 5.23:** Left: an intersection sphere centered at $\mathbf{x}$ contains a true intersection point. Its radius $r$ can be computed approximately by the help of the two AIPs $\hat{\mathbf{p}}_1$ and $\hat{\mathbf{p}}_2$. The center $\mathbf{x}$ is determined by the intercept theorem. Right: the intersection sphere can be sampled by points placed at a regular grid.

The idea is illustrated in Figure 5.23. Assuming the surfaces were flat within the intersection sphere, the center $\mathbf{x}$ determined by the intercept theorem would be the real intersection point. However, this assumption is not always true so that the real intersection point(s) are within the sphere. So if the AIPs are not precise enough, then we can sample each such sphere to get more accurate (discrete) intersection points.
Of course, we could use our RST, proposed in Section 5.2.3, again. However, in contrast to the situation in Section 5.2.3, now the volume that has to be sampled is very simple, namely a single sphere. Therefore, a regular sampling would cause fewer sample points at the same output quality.
That means, if a precise collision point's distance from the surfaces is to be smaller than $\epsilon$, we cover a given intersection sphere by a regular grid with a cell size of $(\frac{2}{\sqrt{3}}\epsilon)^3$ and place a sample point in each cell. For each of these, we just determine the distance to both surfaces. That means, the intersection sphere of radius $r$ is sampled by

$$\lceil \frac{2r}{\frac{2}{\sqrt{3}}\epsilon} \rceil^3 = \lceil \frac{r\sqrt{3}}{\epsilon} \rceil^3$$

points. The situation is illustrated in Figure 5.23 (right).

### 5.3.7 Complexity Considerations

In this section, we analyze the running time and memory usage of our novel approach as well as the number of evaluations of the implicit function that are

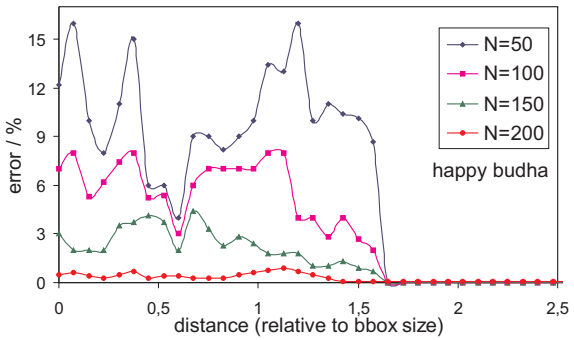



**Figure 5.24:** If the sampling density is too small, our approach can miss some intersections, $N = O(a \ln a)$.

necessary to detect all intersections for a given sampling density described by the number $a$ of surfels.

In general, evaluating $f(\mathbf{x})$ takes $O(\log n)$ time, even if the support of the kernel is bounded, because the nearest neighbor of $\mathbf{x}$ has to be determined (using, for instance, a $k$D tree). Here, fortunately, one evaluation can be done in only $O(1)$ time: the root bracketing and interpolation search evaluate $f(\mathbf{x})$ only at points $\mathbf{x} \in \mathcal{P}_A \cup \mathcal{P}_B$, and computing the precise intersection points can use a brute force nearest neighbor search in constant time, starting from the AIP.

As shown in Section 5.3.3, our root bracketing algorithm can be done in $O(1)$ time.

Then, for at most $O(a \ln a)$ many pairs, our interpolation search has to be started. In the average case, each single interpolation search needs $O(\log \log m)$ evaluations of $f_B$, where $m$ denotes the number of points along the shortest path between $\mathbf{p}_i$ and $\mathbf{p}_j$.

Overall, $f_B$ has to be evaluated $O(a \ln a \log \log m)$ times in the average case where we assume a uniform and independent distribution of the point clouds. As $n \gg m$ and $a$ can be bounded by a constant, this number can also be bounded by $O(\log \log n)$.

Our approach uses two additional data structures, the CPSP map and the sphere-of-influence graph. As already shown in Section 4.3.2, both data structures consume only linear space in the number of points.

If we would like to do collision detection for deformable point clouds, the running time increases up to $O(\log n)$, because the proximity graph as well as the close-pairs shortest-path map have to be updated.

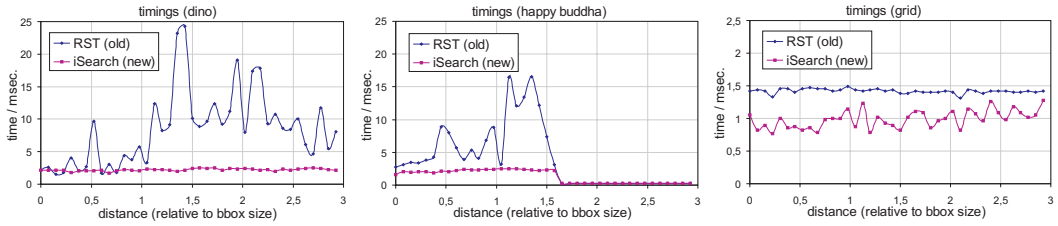



**Figure 5.25:** Timings for different models. Comparison of our interpolation search and the RST of Section 5.2.3.

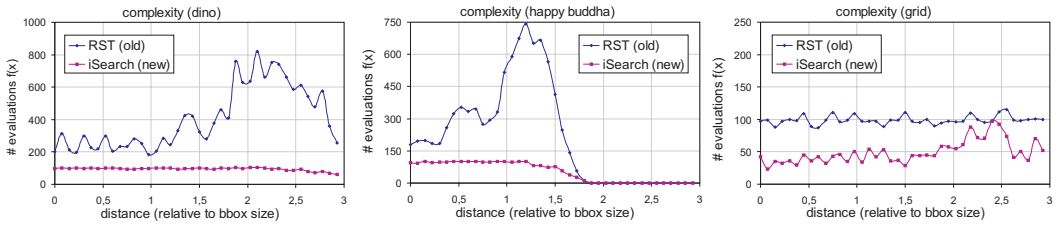



**Figure 5.26:** The number of evaluations of $f(\mathbf{x})$ can be decreased by an order of magnitude by our interpolation search.

## 5.3.8 Results

We implemented our new algorithm in C++. All results were obtained on a 2.8 GHz Pentium-IV using the benchmark procedure proposed in Section 5.2.8. Note that we have not implemented the golden section search that should be used if $f$ is bitonic.

### Minimal Bracket Density

As mentioned, if the number of (conceptual) surfels is too small, then the size of their neighborhoods can become too large, and, as a consequence, the likelihood increases that the normal $\mathbf{n}(\mathbf{x})$ flips its sign without $\mathbf{x}$ actually changing sides. In this case, our method could fail to find pairs of points on different sides of the surface.

Therefore, we propose to estimate the minimum number of surfels (which directly influences the radius of the spherical neighborhoods) using the following preprocessing procedure. We denote a sphere of radius $r_s$ (the surfel radius), that is centered at an arbitrary point from $\mathcal{P}'_A$ as a *collision sphere*, if there is a collision inside. For each distance between the two point cloud models, a large number of collisions tests is performed. Thereby, we count the number of

collision spheres and compare them with the number that is obtained using an exact method. This yields the following error measure

$$\text{error} = 1 - \frac{\#\text{ collision spheres (iSearch)}}{\#\text{ collision spheres (exact method)}}.$$

Note that the RST using a very fine sampling can be seen as an exact algorithm for the comparison.
Each of our tests is performed with a different sampling density, expressed by the number $N = O(a \ln a)$ (see Section 5.3.1). Then, we use the minimal sampling density for which all collisions have been found.
The results for one object can be found in Figure 5.24, which shows the error rate depending on different sampling densities. All our other models of our test suite show a similar behavior and it turned out that $N_{\min} = 200$ is the minimum number, so that the error rate of all intersection tests for all our models is at most 1.0%. This number was used for all further tests.

#### Interpolation Search vs Randomized Sampling

In order to evaluate the performance of our new algorithm, we compared it to the randomized sampling technique (RST) proposed in Section 5.2.3. No BV hierarchies were used.
The number $N_s$ of sample points, that have to be generated for the RST, can be determined as proposed in Section 5.3.6, depending on the same $\epsilon$ that is used for our new approach. As this number would always be large, we once again terminate both collision detection algorithms after the first intersection is found.
However, in the case of non-collision, in particular in the case of larger overlaps between the objects, the running time of the RST would be very long, because of the large $N_s$, which is a big drawback of the old method. Therefore, if $N_s$ is too large, we bound this number by 500. Note that in such cases the old method fails to report all intersection tests correctly, in contrast to our new method, which is another drawback of the old method.
Figure 5.25 shows that the collision queries can be answered much more quickly by our new approach.
The corresponding number of evaluations of the implicit function can be found in Figure 5.26. Note that the number of evaluations can exceed $N_s$ in the case of the RST, since two evaluations are necessary for each random point.

#### Timings depending on Point Density

Figure 5.27 shows the running time for detecting *all* intersections between two objects, depending on different densities of the point clouds. We define the

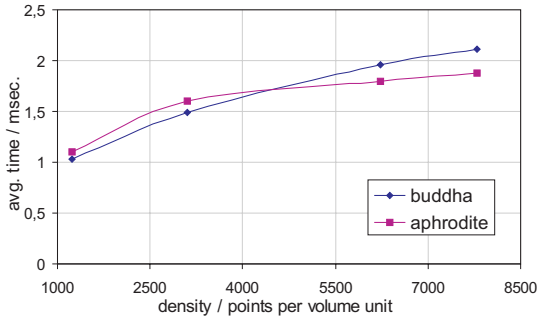



**Figure 5.27:** The plot shows the running time depending on the size of the point clouds. The running time is the average of all timings for distances between 0 and 1.5.

density of an object $A$ with $n$ points as the ratio of $n$ over the number of volume units of the AABB of $A$ (which is at most 8 as each object is scaled uniformly so that it fits into a cube of size $2^3$). This experiment supports our theoretical considerations of Section 5.3.7.
Note that the CPSP maps (see Section 4.3.1) were built, so that the time for evaluating the implicit function remains constant.
We also measured the time that would be needed to compute all nodes on the shortest path between $(p_i, p_j)$ used to initialize the interpolation search (see Algorithm 8). For all our models, this was at most 10% of the overall running time. Therefore, a significant amount of memory in the CPSP map can be saved by computing array $P$ in Algorithm 7 at runtime.

## 5.4 Summary and Discussion

In the first part of this section, we presented a point cloud hierarchy that allows for an efficient BVH traversal into parts of the surface where a collision is very likely. The incorporating sphere coverings are used to quickly discard pairs of BVs that do not collide. Our hierarchies are built, so that the sample points are also the centers of the sphere coverings. As a consequence, this data structure can be stored very memory efficiently. The traversal criterion depends on the consistency of the normals, but no errors occur at all, if the normals are inconsistent. Then, they just produce a few "false positives", so that only some more pairs of BVs are examined. The randomized sampling technique (RST) at the leaves allows for the computation of approximate collision points where the precision can be adjusted by the user or by our time-critical extension. Given

a certain time budget, this extension returns a "best effort" result, so that we can guarantee a real-time response under all circumstances.
We have shown how to adjust the kernel bandwidth, if not using our surface definition based on proximity graphs (Section 4), in order to define the surface even at inner nodes with as few artifacts as possible. Moreover, we examined the relation between the sampling radius (and, thus, between the surface quality) and the sample size of points stored in inner nodes of our hierarchy.
Finally, we have shown that the point cloud hierarchy can be built in $O(n \log n)$, and that the running time of our approach can be analyzed by the technique proposed in Section 3.7. We have also performed several measurements which show that for all our models, the collision queries can be done within a very short time and with very little error.
In the second part, we proposed an approach to accelerate the root finding process. The idea is to use the proximity graph not only for defining the surface, but also for finding common roots using an interpolation search along the shortest path in the proximity graph between two points lying on one surface and on either side of the other surface. This approach does not necessarily use any kind of hierarchical space partitioning data structure, so that it should be well-suited to deformable objects. Furthermore, we have shown how to handle models with boundaries.
If a constant number of intersection points to be reported is sufficient, the running time can be bounded by $O(\log \log n)$ or by $O(\log n)$ for deformable point clouds, respectively. Moreover, our approach can be used, for instance, to accelerate hierarchical collision detection or Boolean operations for this kind of object representation.
Our measurements show that the number of function evaluations is reduced by an order of magnitude and a speedup by factor 5–10 is achieved in many cases, compared to our randomized sampling technique (RST).
Note that our approach can make some errors, although our experiments have shown that the error rate is always very low. For example, if the surface is not uniformly sampled, the covering of all surfels could fail. As a result, parts of the surface are not tested for collision. Moreover, if the sampling rate is too low, or the sampling radius of the randomly chosen point set $\mathcal{P}'_A$ is estimated too low, our approach could miss some collisions. As mentioned, if gaps are not bridged by virtual edges and our sampling rate is too low, the interpolation search cannot be started for pairs of points lying on an interrupted shortest path. But it is not very complicated to avoid or to minimize these errors by increasing the number of sample points or the sampling radius. Moreover, we should try to bridge gaps using virtual edges.
If the normals based on our WLS approach are not consistent throughout space,

finding points on the other side could fail. Fortunately, our experiments show that the normals are consistent over fairly large volumes without any preconditioning. So, if the size of the surfels is not chosen too large, $\mathbf{n}(\mathbf{x})$ will be consistent most of the time.
The interpolation search converges if $f$ is monotone along the shortest path. In case the normals are not consistent so that the sign of $f(\mathbf{x})$ is not consistent, too, we could use the golden section search if $f$ is bitonic. If $f$ is not monotone or bitonic, we could use a simple binary search if the sign of $f$ is consistent.

## 5.5 Future Work

There are many avenues for further work. On the one hand, performance and accuracy can be increased during the traversal of the point cloud hierarchies. For instance, at inner nodes, the traversal criteria can probably be improved, e.g., by applying knowledge of the separation of the eigenvalues.
On the other hand, the point hierarchy and the hierarchical sphere covering could be improved so as to allow for faster collision detection. For instance, the sphere covering could be made tighter. Additionally, our method could be combined with the one presented by [CBC+01].
Another very important point regarding our hierarchy is how to draw the samples at inner nodes so as to minimize surface errors. Obviously, an adaptive sampling strategy that draws more points at *critical* parts of the objects seems to be more promising than a uniform sampling. Therefore, we first have to examine the question: which parts of the objects are critical with respect to the collision detection process. These parts could be areas with high curvature, exposed parts, or very fine or thin objects with a high aspect ratio. Possible criteria are the ratio of the size of the eigenvalues of the local covariance matrix or local eccentricities. A sampling strategy, that also allows for describing the quality of the reconstructed surface using some kind of error measure (e.g., the Hausdorff distance), could improve our approach. Until now, the surface quality can only be described by our sampling radius.
Starting from the finest resolution, an algorithm could remove points step by step that cause the minimum change of the Hausdorff distance to the original surface. As a consequence, an adaptive sampling strategy can automatically be achieved. Of course, we should store the Hausdorff distances at the corresponding inner nodes to allow for a time-critical collision detection: during the traversal of two BVHs, we can determine the likelihood that two parts of the surface will collide depending on the corresponding sample and Hausdorff distance.

The approach using our interpolation search to determine common roots more quickly could be a way to handle deformable point clouds, since it does not utilize any spatial acceleration structure. Moreover, the SIG (and the CPSP map) can be updated in time $O(\log^3 n)$ or $O(\log n)$, respectively, if using the BBD trees for finding the approximate nearest neighbors. From a theoretical point of view, a mathematically more rigorous estimation of the minimal bracket density would be appealing.

# 6 Conclusions and Future Work

This chapter summarizes the preceding chapters at a more abstract level and shows the most important directions for future work. Note that each chapter already concludes with a more detailed summary and a discussion of future work.

In this thesis, we were concerned with efficient algorithms and data structures for collision detection for polygon-based and point-based models. Our average-case approach for approximate collision detection does not need any intersection tests between the object's surfaces. The only condition of our approach is, that the surface has to be given or can be reconstructed during the construction of the BVH, so that the number of possible collision cells can be determined. As a consequence, our average-case approach is applicable to virtually all types of BVHs. Our measurements have shown that, if sufficient time is available, the error rate is always very low compared with a traditional, conservative approach, but is clearly faster (factor 3 to 6) than a sophisticated DOP tree algorithm. Moreover, we have shown that with decreasing available processing time, the error increases in a controlled way. With existing methods, this was not possible.

One of the most important points for future work is to ensure that the assumption of evenly distributed possible collision cells becomes even more realistic. As a possible solution, we propose to use two boundaries for each BV: one classical outer boundary that encloses the whole object and one inner boundary where no object primitives are allowed to lie within the corresponding inner volume. Moreover, it would be very interesting to examine this kind of BV for other collision detection approaches to accelerate them further.

In contrast to our average-case approach, our method for collision detection based on point cloud hierarchies does explicitly use the object's surface for guiding the traversal into those parts of the hierarchy where the collision is more likely, and also uses them for determining the intersection curves (or discrete intersection points) in the leaves. Our interpolation search does not need any hierarchies at all, and, because the corresponding data structure, the proximity graph, can be updated in logarithmic time, this method should be suitable for deformable point clouds as well. To the best of our knowledge, this is the first efficient algorithm (yielding a $O(\log n)$ behavior in the average case) that can

determine the intersections between implicit surfaces over point clouds.
If one would like to compare the average-case approach with our point cloud collision detection, it can be observed that with roughly same object complexities, the average-case approach is about 3 times faster, but has a higher error rate and does not compute the (discrete) intersection points. The most important point for future work is to implement and evaluate our approach for deformable objects.
Our method for analyzing the running time of collision detection algorithms can be used for any hierarchical approach that uses simultaneous BVH traversal (thus, it can be used for our approaches). It explains the sublinear running time behavior that has been observed for a long time.
In order to minimize the errors during the collision detection for point clouds, we have improved the surface definition based on weighted least squares by utilizing a new, approximate geodesic kernel. Previous to our work, the bandwidth of the kernel had to be determined manually, which now can be done automatically by using the proximity graph. Moreover, we have extended this definition so that it can handle objects with boundaries just as well. Compared to previous work, the artifacts can clearly be reduced by our approach.
In this thesis, we introduced novel approaches and data structures into the area of computer graphics, namely,

- an average-case approach for collision detection,
- a proximity graph for defining surfaces over point clouds,
- a point cloud BVH that approximates the surface at different levels, and
- an interpolation search for determining the zero set of implicit functions,

that allow us to estimate the running time and the quality of the collision detection process. Our evaluations, as well as our theoretical analysis, have shown that our methods are efficient and can be used in real-time environments and time-critical scenarios.
More importantly, we have shown new directions for future research and, all the more, new directions in the area of collision detection, namely approximate collision detection with controllable errors, and collision detection between point clouds.

# Bibliography


[AA03] Anders Adamson and Marc Alexa. Approximating and intersecting surfaces from points. In *Proc. Eurographics Symp. on Geometry Processing*, pages 230–239, 2003.

[AA04a] Anders Adamson and Marc Alexa. Approximating bounded, non-orientable surfaces from points. In *Shape Modeling International*, pages 243–252, 2004.

[AA04b] Anders Adamson and Marc Alexa. On normals and projection operators for surfaces defined by point sets. In *Eurographics Symposium on Point-Based Graphics (SPBG'04)*, pages 149–155, 2004.

[AB04] Dominique Attali and Jean-Daniel Boissonnat. A linear bound on the complexity of the delaunay triangulation of points on polyhedral surfaces. *Discrete and Computational Geometry*, 31(3):369–384, 2004.

[ABCO+03] M. Alexa, J. Behr, Daniel Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Computing and rendering point set surfaces. *IEEE Trans. on Visualization and Computer Graphics*, 9(1):3–15, 2003.

[ABG+01] Pankaj K. Agarwal, J. Basch, L. J. Guibas, J. Hershberger, and L. Zhang. Deformable free space tiling for kinetic collision detection. In *Algorithmic and Computational Robotics: New Directions (Proc. 5th Workshop Algorithmic Found. Robotics)*, pages 83–96, 2001.

[ACDL00] N. Amenta, S. Choi, T. K. Dey, and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. In *Proceedings of the sixteenth annual symposium on Computational geometry*, pages 213–222. ACM Press, 2000.

[ACT00] Sigal Ar, Bernard Chazelle, and Ayellet Tal. Self-customized BSP trees for collision detection. *Computational Geometry: Theory and Applications*, 15(1–3):91–102, 2000.

[AD03] Bart Adams and Philip Dutré. Interactive boolean operations on surfel-bounded solids. *ACM Transactions on Graphics (SIGGRAPH 2003)*, 22(3):651–656, 2003.

[AdBG+01] Pankaj K. Agarwal, Mark de Berg, Joachim Gudmundsson, Mikael Hammar, and Herman J. Haverkort. Box-trees and r-trees with near-optimal query time. In *Proc. Seventeenth Annual Symposium on Computational Geometry (SCG 2001)*, pages 124–133, 2001.

[AGN+04] Pankaj Agarwal, Leonidas Guibas, An Nguyen, Daniel Russel, and Li Zhang. Collision detection for deforming necklaces. *Computational Geometry: Theory and Applications*, 28:137–163, 2004.

[AH85] David Avis and Joe Horton. Remarks on the sphere of influence graph. *Discrete geometry and convexity, Annals of the New York Academy of Sciences*, 440:323–327, 1985.

[AHPK00] Pankaj K. Agarwal, Sariel Har-Peled, and Meetesh Karia. Computing approximate shortest paths on convex polytopes. In *Proc. 16th ACM Sympos. Comput. Geom.*, pages 270 – 279, 2000.

[AK04] Nina Amenta and Yong Kil. Defining point-set surfaces. *ACM Transactions on Graphics (SIGGRAPH 2004)*, 23(3):264–270, 2004.

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45:891–923, 1998.

[AMT02] Sigal Ar, Gil Montag, and Ayellet Tal. Deferred, self-organizing BSP trees. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2002)*, 21(3):269–278, September 2002.

[ASCL02] Nina Amenta, Tamal K. Dey S. Choi, and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. *Intl. Journal on Computational Geometry & Applications*, 12:125–141, 2002.

[Bar90] David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *ACM Transactions on Graphics (SIGGRAPH 1990)*, 9(3):19–28, 1990.

[BFA02] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics (SIGGRAPH 2002)*, 21(3):594–603, 2002.

[BH99] Wilhelm Barth and Ernst Huber. Computations with tight bounding volumes for general parametric surfaces. In *Proc. 15th European Workshop on Computational Geometry (CG 1999)*, pages 123–126, 1999.

[BHW96] Ronen Barzel, John Hughes, and Daniel N. Wood. Plausible motion simulation for computer graphics animation. In *Proc. Eurographics Workshop Computer Animation and Simulation*, pages 183–197, 1996.

[Blo94] Jules Bloomenthal. An implicit surface polygonizer. In Paul Heckbert, editor, *Graphics Gems IV*, pages 324–349. Academic Press, Boston, 1994.

[BLS00] Elizabeth D. Boyer, L. Lister, and B. Shader. Sphere-of-influence graphs using the sup-norm. *Mathematical and Computer Modelling*, 32(10):1071–1082, 2000.

[BMF03] R. Bridson, S. Marino, and R. Fedkiw. Simulation of clothing with folds and wrinkles. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer animation (SCA '03)*, pages 28–36. Eurographics Association, 2003.

[BO79] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *Transactions on Computing*, 28(9):643– 647, 1979.

[Boy79] John W. Boyse. Interference detection among solids and surfaces. *Commun. ACM*, 22(1):3–9, 1979.

[BSK04] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In *Eurographics Symposium on Point-Based Graphics (SPBG'04)*, pages 25–32, 2004.

[BT95] S. Bandi and D. Thalmann. An adaptive spatial subdivision of the object space for fast collision detection of animating rigid bodies. *Computer Graphics Forum (Proc. of EUROGRAPHICS 1995)*, 14(3):259–270, 1995.

[BW02] George Baciu and Wingo Sai-Keung Wong. Hardware-assisted self-collision for deformable surfaces. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST 2002)*, pages 129–136, Hong Kong, China, November 2002.

[BW03] George Baciu and Wingo Sai-Keung Wong. Image-based techniques in a hybrid collision detector. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):254–271, 2003.

[BWG03] Kavita Bala, Bruce Walter, and Donald P. Greenberg. Combining edges and points for interactive high-quality rendering. *ACM Transactions on Graphics (SIGGRAPH 2003)*, 22(3):631–640, 2003.

[BWY80] Jon Louis Bentley, Bruce W. Weide, and Andrew C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions on Mathematical Software*, 6(4):563–580, 1980.

[CBC+01] Jonathan C. Carr, Richard K. Beatson, Jon B. Cherrie, Tim J. Mitchell, W. Richard Fright, Bruce C. McCallum, and Tim R. Evans. Reconstruction and representation of 3D objects with radial basis functions. *ACM Transactions on Graphics (SIGGRAPH 2001)*, 20(3):67–76, 2001.

[CH90] J. Chen and Y. Han. Shortest paths on a polyhedron. In *Proc. 6th ACM Symp. on Computatinal Geometry*, pages 360 – 369, 1990.

[Cha01] Pavel Chalmoviansky. Subdivision surfaces in geometric modelling. In *European Research Consortium for Informatics and Mathematics (ERCIM) News, No. 44*, page 9, January 2001.

[Chu98] Adrian J. Chung. Re: Obb trees, comp.graphics.algorithms. usenet newsgroup article, jun 1998. http://groups.google.com/groups?selm=6le9nu$ef$91@magpie.doc.ic.ac.uk.

[CL95] W. S. Cleveland and C. L. Loader. Smoothing by local regression: Principles and methods. In W. Haerdle and M. G. Schimek, editors, *Statistical Theory and Computational Aspects of Smoothing*, pages 10–49. Springer, New York, 1995.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[CS93] John Horton Conway and Neil James Alexander Sloane. *Sphere Packings, Lattices, and Groups*. Springer-Verlag, New York, 2 edition, 1993.

[CSY97] T. M. Chan, J. Snoeyink, and C. K. Yap. Primal dividing and dual pruning: Output-sensitive construction of 4-d polytopes and 3-d Voronoi diagrams. *Discrete Comput. Geom.*, 18:433–454, 1997.

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[DCOT03] A. Karol D. Cohen-Or, S. Lev-Yehudi and A. Tal. Inner-cover of non-convex shapes. *International Journal of Shape Modeling*, 9(2):223–238, 2003.

[DDCB01] Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, and Alan H. Barr. Dynamic real-time deformations using space and time adaptive sampling. *ACM Transactions on Graphics (SIGGRAPH 2001)*, 20(3):31–36, 2001.

[DDG00] Matthew Dickerson, Christian A. Duncan, and Michael T. Goodrich. K-d trees are better when cut on the longest side. In *Proc. of the 8th Annual European Symposium on Algorithms (ESA 2000)*, pages 179–190, 2000.

[Dev02] Olivier Devillers. The Delaunay hierarchy. *Internat. J. Found. Comput. Sci.*, 13:163–180, 2002.

[DG04] Tamal K. Dey and Samrat Goswami. Provable surface reconstruction from noisy samples. In *Proc. Symp. on Computational Geometry*, pages 330–339, 2004.

[DHKS93] David P. Dobkin, John Hershberger, David G. Kirkpatrick, and Subhash Suri. Computing the intersection-depth of polyhedra. *Algorithmica*, 9(6):518–533, 1993.

[DK83] David P. Dobkin and David G. Kirkpatrick. Fast detection of polyhedral intersection. *Theor. Comput. Sci.*, 27:241–253, 1983.

[DK85] David P. Dobkin and David G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6(3):381–392, 1985.

[DK90] David P. Dobkin and David G. Kirkpatrick. Determining the separation of preprocessed polyhedra - a unified approach. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming (ICALP)*, pages 400–413, 1990.

[DO00] John Dingliana and Carol O'Sullivan. Graceful degradation of collision handling in physically based animation. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2000)*, 19(3):239–247, August 2000.

[Dwy95] Rex A. Dwyer. The expected size of the sphere-of-influence graph. *Computational Geometry: Theory and Applications*, 5(3):155–164, 1995.

[EL01] Stephan A. Ehmann and Ming C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2001)*, 20(3):500–510, 2001.

[Eri04] Christer Ericson. *Real-Time Collision Detection*. Series In Interactive 3D Technology. Elsevier, San Francisco, United States of America, 2004.

[ERW89] Herbert Edelsbrunner, Günter Rote, and Emo Welzl. Testing the necklace condition for shortest tours and optimal factors in the plane. *Theoretical Computer Science*, 66(2):157–180, 1989.

[Far02] Gerald Farin. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[FGL03] Arnulph Fuhrmann, Clemens Groß, and Volker Luckas. Interactive animation of cloth including self collision detection. In *WSCG*, pages 141–148, 2003.

[FL01] Susan Fisher and Ming Lin. Fast penetration depth estimation for elastic bodies using deformed distance fields. In *Proc. International Conf. on Intelligent Robots and Systems (IROS)*, pages 330–336, 2001.

[FPRJ00] Sarah P. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. *ACM Transactions on Graphics (SIGGRAPH 2000)*, 19(3):249–254, 2000.

[FSG03] A. Fuhrmann, G. Sobottka, and C. Groß. Distance fields for rapid collision detection in physically based modeling. In *Proceedings of GraphiCon 2003*, pages 58–65, Moscow, September 2003.

[GD02] Stéphane Guy and Gilles Debunne. Layered shells for fast collision detection. Technical report, INRIA, 2002.

[GD04] Stéphane Guy and Gilles Debunne. Monte-carlo collision detection. Technical Report RR-5136, INRIA, March 2004.

[GDO00] F. Ganovelli, J. Dingliana, and C. O'Sullivan. Buckettree: Improving collision detection between deformable objects. In *Proc. of Spring Conference in Computer Graphics (SCCG2000)*, pages 156–163, Bratislava, 2000.

[GKJ+05] Naga Govindaraju, David Knott, Nitin Jain, Ilknurk Kabal, Rasmus Tamstorf, Russel Gayle, Ming C. Lin, and Dinesh Manocha. Interactive collision detection between deformable models using chromatic decomposition. *ACM Transactions on Graphics (SIGGRAPH 2005)*, 24(3), August 2005.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. *ACM Transactions on Graphics (SIGGRAPH 1996)*, 15(3):171–180, 1996.

[GLM03] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Fast self-collision detection in general environments using graphics

processors. Technical Report TR03-044, University of North Carolina at Chapel Hill, 2003.

[GLM04] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Fast and reliable collision culling using graphics hardware. In *Symposium on Virtual Reality Software and Technology (VRST 2004)*, 2004.

[Got00] Stefan Aric Gottschalk. *Collision queries using oriented bounding boxes*. PhD thesis, University of North Carolina, 2000. Director-Dinesh Manocha and Director-Ming C. Lin.

[GPS92] L. Guibas, J. Pach, and M. Sharir. Generalized sphere-of-influence graphs in higher dimensions. In *Manuscript*, 1992.

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

[GS01] Eitan Grinspun and Peter Schröder. Normal bounds for subdivision-surface interference detection. In *Proceedings of IEEE Visualization (VIS 2001)*, pages 333–340, 2001.

[GvL89] G. H. Golub and C. F. van Loan. *Matrix computations*. Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[GZ03] Alexander Gress and Gabriel Zachmann. Object-space interference detection on programmable graphics hardware. In *SIAM Conf. on Geometric Design and Computing*, Seattle, Washington, November13–17 2003.

[Här90] W. Härdle. *Applied nonparametric regression*, volume 19 of *Econometric Society Monograph*. Cambridge University Press, New York, 1990.

[Har96] John C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(9):527–545, 1996.

[HDD+92] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. *ACM Transactions on Graphics (SIGGRAPH 1992)*, 11(3):71–78, 1992.

[He99] Taosong He. Fast collision detection using quospo trees. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 55–62, 1999.

[Hig90] Nicholas J. Higham. Analysis of the Cholesky decomposition of a semi-definite matrix. In M. G. Cox and S. J. Hammarling, editors, *Reliable Numerical Computation*, pages 161–185. Oxford University Press, 1990.

[HMB01] Suejung Huh, Dimitris N. Metaxas, and Norman I. Badler. Collision resolutions in cloth simulation. In *IEEE Computer Animation Conf.*, pages 122–127, Seoul, Korea, November 2001.

[HTG04] B. Heidelberger, M. Teschner, and M. Gross. Detection of collisions and self-collisions using image-space techniques. In *Proceedings of the 12-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2004 (WSCG'2004)*, pages 145–152, University of West Bohemia, Czech Republic, February 2004.

[Hub96] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[JC98] David E. Johnson and Elaine Cohen. A framework for efficient minimum distance computations. In *Proc. IEEE Intl. Conf. Robotics and Automation*, pages 3678–3684, 1998.

[JP04] Doug L. James and Dinesh K. Pai. BD-Tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (SIGGRAPH 2004)*, 23(3):393–398, August 2004.

[JT92] J. W. Jaromczyk and Godfried T. Toussaint. Relative neighborhood graphs and their relatives. *Proc. of the IEEE*, 80(9):1502–1571, 1992.

[KB04] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[KEHKL+99] III Kenneth E. Hoff, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized voronoi diagrams using graphics hardware. *ACM Transactions on Graphics (SIGGRAPH 1999)*, 18(3):277–286, 1999.

[KGL+98a] S. Krishnan, M. Gopi, M. Lin, Dinesh Manocha, and A. Pattekar. Rapid and accurate contact determination between spline models using shelltrees. *Computer Graphics Forum*, 17(3):315–326, 1998.

[KGL+98b] Shankar Krishnan, M. Gopi, Ming C. Lin, Dinesh Manocha, and A. Pattekar. Rapid and accurate contact determination between spline models using ShellTrees. *Computer Graphics Forum (Proc. of EUROGRAPHICS 1998)*, 17(3):315–326, September 1998.

[KHM+98] James T. Klosowski, Martin Held, Jospeh S. B. Mitchell, Henry Sowrizal, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of $k$-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

[KKF+02] Jan Klein, Jens Krokowski, Matthias Fischer, Michael Wand, Rolf Wanka, and Friedhelm Meyer auf der Heide. The randomized sample tree: A data structure for interactive walkthroughs in externally stored virtual environments. In *Symposium on Virtual Reality Software and Technology (VRST 2002)*, pages 137 – 146, Hong Kong, China, November 2002.

[KKF+04] Jan Klein, Jens Krokowski, Matthias Fischer, Michael Wand, Rolf Wanka, and Friedhelm Meyer auf der Heide. The randomized sample tree: A data structure for interactive walkthroughs in externally stored virtual environments. *Journal of PRESENCE: Teleoperators & Virtual Environments*, 13(6):617 – 637, December 2004. The MIT Press.

[Klo98] James Thomas Klosowski. *Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments*. PhD thesis, State University of New York at Stony Brook, 1998. Adviser-Joseph S. Mitchell.

[KLRS04] Andreas Kolb, Lutz Latta, and Christof Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proc. of Graphics Hardware 2004*, pages 123–131, 2004.

[KM97] Shankar Krishnan and Dinesh Manocha. An efficient surface intersection algorithm based on lower-Dimensional formulation. *ACM Transactions on Graphics*, 16(1):74–106, January 1997.

[KNF04] Stefan Kimmerle, Matthieu Nesme, and Francois Faure. Hierarchy accelerated stochastic collision detection. In *Proc. 9th International Fall Workshop Vision, Modeling, and Visualization (VMV 2004)*, 2004.

[KS00] Takashi Kanai and Hiromasa Suzuki. Approximate shortest path on a polyhedral surface based on selective refinement of the discrete graph and its applications. In *Proc. Geometric and Processing*, pages 241 – 250, 2000.

[KSTK98] Yoshifumi Kitamura, Andrew Smith, Haruo Takemura, and Fumio Kishino. A real-time algorithm for accurate collision detection for deformable polyhedral objects. *Presence*, 7(1):36–52, 1998.

[KV03] Aravind Kalaiah and Amitabh Varshney. Statistical point geometry. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing 2003*, pages 107–115. Eurographics Association, 2003.

[KZ03a] Jan Klein and Gabriel Zachmann. ADB-trees: Controlling the error of time-critical collision detection. In *Proc. 8th International Fall Workshop Vision, Modeling, and Visualization (VMV 2003)*, pages 37–45, Mnchen, Germany, November 2003.

[KZ03b] Jan Klein and Gabriel Zachmann. Time-critical collision detection using an average-case approach. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST 2003)*, pages 22–31, Osaka, Japan, October 2003.

[KZ04a] Jan Klein and Gabriel Zachmann. Nice and fast implicit surfaces over noisy point clouds. In *ACM SIGGRAPH 2004, Sketches Session*, Los Angeles, USA, August 2004.

[KZ04b] Jan Klein and Gabriel Zachmann. Point cloud collision detection. In *Computer Graphics Forum (Proc. EUROGRAPHICS 2004)*, pages 567–576, Grenoble, France, September 2004.

[KZ04c] Jan Klein and Gabriel Zachmann. Point cloud surfaces using geometric proximity graphs. *Computers and Graphics*, 28(6):839–850, December 2004. Elsevier.

[KZ04d] Jan Klein and Gabriel Zachmann. Proximity graphs for defining surfaces over point clouds. In *Eurographics Symposium on Point-Based Graphics (SPBG'04)*, pages 131–138, Zurich, Switzerland, June 2004.

[KZ05a] Jan Klein and Gabriel Zachmann. The expected running time of hierarchical collision detection. In *ACM SIGGRAPH 2005, Poster Session*, Los Angeles, USA, August 2005. to appear.

[KZ05b] Jan Klein and Gabriel Zachmann. Interpolation search for point cloud intersection. In *Proceedings of the 13-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2005 (WSCG'2005)*, pages 163–170, Plzen, Czech Republic, February 2005.

[LAM01] Thomas Larsson and Tomas Akenine-Möller. Collision detection for continuously deforming bodies. In *Eurographics*, pages 325–333, 2001. short presentation.

[LC91] Ming C. Lin and John F. Canny. A fast algorithm for incremental distance calculation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1008–1014, 1991.

[LC92] Ming C. Lin and John F. Canny. Efficient collision detection for animation. In *Proc. of 3rd Eurographics Workshop on Animation and Simulation*, Cambridge, England, 1992.

[LCLL02] Rynson W. H. Lau, Oliver Chan, Mo Luk, and Frederick W. B. Li. A collision detection method for deformable objects. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST 2002)*, pages 113–120, November 2002.

[LCN99] Jean-Christophe Lombardo, Marie-Paule Cani, and Fabrice Neyret. Real-time collision detection for virtual surgery. In *Proc. of Computer Animation*, pages 82–90, Geneva, Switzerland, May 1999.

[Lee00] I.-K. Lee. Curve reconstruction from unorganized points. *Computer Aided Geometric Design*, 17(2):161–177, 2000.

[Lev98] David Levin. The approximation power of moving least-squares. *Mathematics of Computation*, 67(224):1517–1531, 1998.

[Lev03] David Levin. Mesh-independent surface interpolation. In Hamann Brunnett and Mueller, editors, *Geometric Modeling for Scientific Visualization*, pages 37–49. Springer, 2003.

[LS81] Peter Lancaster and Kes Salkauskas. Surfaces generated by moving least squares methods. *Mathematics of Computation*, 37(155):141–158, 1981.

[McL74] D. H. McLain. Drawing contours from arbitrary data points. *Computer Journal*, 17(4):318–324, 1974.

[Mel00] Stan Melax. Dynamic plane shifting bsp traversal. In *Graphics Interface 2000*, pages 213–220, 2000.

[MKE03] Johannes Mezger, Stefan Kimmerle, and Olaf Etzmuß. Hierarchical Techniques in Collision Detection for Cloth Animation. *Journal of WSCG*, 11(2):322–329, 2003.

[MOK95] Karol Myszkowski, Oleg G. Okunev, and Tosiyasu L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–512, 1995.

[MPT99] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Six degrees-of-freedom haptic rendering using voxel sampling. *ACM Transactions on Graphics (SIGGRAPH 1999)*, 18(3):401–408, 1999.

[MQ03] T. S. Michael and Thomas Quint. Sphere of influence graphs and the $l_\infty$-metric. *Discrete Applied Mathematics*, 127(3):447 – 460, 2003.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MW88] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. *ACM Transactions on Graphics (SIGGRAPH 1988)*, 7(3):289–298, 1988.

[Nea04] Andrew Nealen. An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods

for scattered data approximation and interpolation. Technical report, Discrete Geometric Modeling Group, TU Darmstadt, May 2004.

[OBA+03] Yutaka Ohtake, Alexander Belyaev, Marc Alexa, Greg Turk, and Hans-Peter Seidel. Multi-level partition of unity implicits. *ACM Transactions on Graphics (SIGGRAPH 2003)*, 22(3):463–470, 2003.

[PFTV93] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 2nd edition, 1993.

[PG95] Ian J. Palmer and Richard L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105–116, June 1995.

[PGK02] Mark Pauly, Markus H. Gross, and Leif Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings of IEEE Visualization 2002 (VIS 2002)*, pages 163–170, 2002.

[PKKG03] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. *ACM Transactions on Graphics (SIGGRAPH 2003)*, 22(3):641–650, 2003.

[Pro97] Xavier Provot. Collision and self-collision handling in cloth model dedicated to design garments. In *Proc. of Graphics Interface 1997*, pages 177 – 189, 1997.

[PvBZG00] Hanspeter Pfister, Jeroen van Baar, Matthias Zwicker, and Markus Gross. Surfels: Surface elements as rendering primitives. *ACM Transactions on Graphics (SIGGRAPH 2000)*, 19(3):335–342, 2000.

[RCFC03] Laks Raghupathi, Vincent Cantin, François Faure, and Marie-Paule Cani. Real-time simulation of self-collisions for virtual intestinal surgery. In Nicholas Ayache and Hervé Delingette, editors, *Proceedings of the International Symposium on Surgery Simulation and Soft Tissue Modeling*, number 2673 in Lecture Notes in Computer Science, pages 15–26. Springer-Verlag, 2003.

[RHHL02] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3D model acquisition. *ACM Transactions on Graphics*, 21(3):438–446, July 2002.

[RKC02] Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart. Fast continuous collision detection between rigid bodies. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2002)*, 21(3), 2002.

[RL85] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed R-trees. In *SIGMOD Conference*, pages 17–31, 1985.

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. *ACM Transactions on Graphics (SIGGRAPH 2000)*, 19(3):343–352, 2000.

[Rog63] C.A. Rogers. Covering a sphere with spheres. *Mathematika*, 10:157–164, 1963.

[Sch94] Sven Schönherr. Computation of smallest ellipsoids around point sets. Diploma thesis, Freie Universität Berlin, Berlin, Germany, 1994.

[Sch04] Dirk Schlenke. Approximative gedodätische Distanzen auf 3d Modellen. Bachelor thesis, University of Paderborn, Germany, 2004.

[Sed89] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, 2 edition, 1989.

[SF91] Mikio Shinya and Marie-Claire Forgue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):132–134, October–December 1991.

[SHH98] Subhash Suri, Philip M. Hubbard, and John F. Hughes. Collision detection in aspect and scale bounded polyhedra. In *SODA*, pages 127–136, 1998.

[SPG03] Christian Sigg, Ronald Peikert, and Markus Gross. Signed distance transform using graphics hardware. In *IEEE Vis2003*, pages 83–90, 2003.

[ST95] E. Schömer and C. Thiel. Efficient collision detection for moving polyhedra. In *11th Annual Symposium on Computational Geometry*, pages 51–60, June 1995.

[ST96] E. Schömer and C. Thiel. Subquadratic algorithms for the general collision detection problem. In *12th European Workshop on Computational Geometry*, pages 95–101, March 1996.

[TFY00] Sotoshi Tanaka, Yasushi Fukuda, and Hiroaki Yamamoto. Stochastic algorithm for detecting intersection of implicit surfaces. *Computers and Graphics*, 24(4):523 – 528, 2000.

[THM+03] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proc. 8th International Fall Workshop Vision, Modeling, and Visualization (VMV 2003)*, pages 47–54, 2003.

[TKH+05] Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, Marie-Paule Cani, François Faure, Nadia Magnenat-Thalmann, Wolfgang Strasser, and Pascal Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 24(1):61–81, 2005.

[TKZ+04] Matthias Teschner, Stefan Kimmerle, Gabriel Zachmann, Bruno Heidelberger, Laks Raghupathi, Arnulph Fuhrmann, Marie-Paule Cani, François Faure, Nadia Magnetat-Thalmann, and Wolfgang Strasser. Collision detection for deformable objects. In *Proc. Eurographics, State-of-the-Art Report*, pages 119–135, Grenoble, France, 2004.

[Tou88] Godfried T. Toussaint. A graph-theoretical primal sketch. In Godfried T. Toussaint, editor, *Computational Morphology*, pages 229–260, 1988.

[Tur90] Greg Turk. Interactive collision detection for molecular graphics. Technical Report TR90-014, University of North Carolina at Chapel Hill, 1990.

[US97] S. Uno and Mel Slater. The sensitivity of presence to collision response. In *Proc. of IEEE Virtual Reality Annual International Symposium (VRAIS)*, page 95, Albuquerque, New Mexico, March 1997.

[VB94] T. Lewis V. Barnett. *Outliers in Statistical Data*. John Wiley and Sons, New York, 1994.

[VCC98] Baba C. Vemuri, Y. Cao, and L. Chen. Fast collision detection algorithms with applications to particle flow. *Computer Graphics Forum*, 17(2):121–134, 1998.

[vdB97] Gino van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1–14, 1997.

[vdB03] Gino van den Bergen. *Collision Detection in interactive 3D environments*. Series In Interactive 3D Technology. Elsevier, San Francisco, United States of America, 2003.

[VM94] P. Volino and Nadia Magnenat Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum (Proc. of EUROGRAPHICS 1994)*, 13(3):155–166, 1994.

[VM95] Pascal Volino and Nadia Magnenat Thalmann. Collision and self-collision detection: Efficient and robust solutions for highly deformable surfaces. In *Computer Animation and Simulation '95*, pages 55–65, September 1995.

[VSC01] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2001)*, 20(3):260–267, September 2001.

[Wan04] Michael Wand. *Point-Based Multi-Resolution Rendering*. PhD thesis, Department of computer science and cognitive science, University of Tübingen, 2004.

[Wel91] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes Comput. Sci.*, pages 359–370. Springer-Verlag, 1991.

[Wen95] Holger Wendland. Piecewise polynomial, positive definite and compactly supported radial basis functions of minimal degree. *Advances in Computational Mathematics*, 4:389–396, 1995.

[WFP+01] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. *ACM Transactions on Graphics (SIGGRAPH 2001)*, 20(3):361 – 370, 2001.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984.

[Zac98] Gabriel Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium (VRAIS 1998)*, pages 90–97, Atlanta, Georgia, March 1998.

[Zac00] Gabriel Zachmann. *Virtual Reality in Assembly Simulation – Collision Detection, Simulation Algorithms, and Interaction Techniques*. PhD thesis, Darmstadt University of Technology, Germany, May 2000.

[Zac02] Gabriel Zachmann. Minimal hierarchical collision detection. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST 2002)*, pages 121–128, Hong Kong, China, November 2002.

[ZK03] Gabriel Zachmann and Günter Knittel. An architecture for hierarchical collision detection. In *Journal of WSCG '2003*, pages 149–156, University of West Bohemia, Plzen, Czech Republic, February3–7 2003.

[ZL03] Gabriel Zachmann and Elmar Langetepe. Geometric data structures for computer graphics. In *Proc. of ACM SIGGRAPH*. ACM Transactions of Graphics, 27–31July 2003.

[ZPvBG02] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. EWA splatting. *IEEE Trans. on Visualization and Computer Graphics*, 8(3):223–238, 2002.

[ZS99] Yunhong Zhou and Subhash Suri. Analysis of a bounding box heuristic for object intersection. *Journal of the ACM*, 46(6):833–857, 1999.

[ZS00a] Yunhong Zhou and Subhash Suri. Collision detection using bounding boxes: Convexity helps. In *Proceedings of the 8th Annual European Symposium on Algorithms (ESA 2000)*, pages 437–448, 2000.

[ZS00b] D. Zorin and P. Schröder. Subdivision for modeling and animation. In *ACM SIGGRAPH 2000, Course*, 2000.

[ZTK+05] Gabriel Zachmann, Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Laks Raghupathi, and Arnulph Fuhrmann. Real-time collision detection for dynamic virtual environments. In *Tutorial #4, IEEE VR*, pages 1–32, Bonn, Germany, 2005.

[ZY00] Dongliang Zhang and Matthew M. F. Yuen. Collision detection for clothed human animation. In *Proceedings of the 8th Pacific Conference on Computer Graphics and Applications*, pages 328–337. IEEE Computer Society, 2000.

# Index