Interpolation Search for Point Cloud Intersection

Jan Klein
University of Paderborn, Germany
janklein@uni-paderborn.de

Gabriel Zachmann
University of Bonn, Germany
zach@cs.uni-bonn.de

ABSTRACT

We present a novel algorithm to compute intersections of two point clouds. It can be used to detect collisions between implicit surfaces defined by two point sets, or to construct their intersection curves. Our approach utilizes a proximity graph that allows for quick interpolation search of a common zero of the two implicit functions.
First, pairs of points from one point set are constructed, bracketing the intersection with the other surface. Second, an interpolation search along shortest paths in the graph is performed. Third, the solutions are refined. For the first and third step, randomized sampling is utilized.
We show that the number of evaluations of the implicit function and the overall runtime is in $O(\log\log N)$, where N is the point cloud size. The storage is bounded by $O(N)$.
Our measurements show that we achieve a speedup by an order of magnitude compared to a recently proposed randomized sampling technique for point cloud collision detection.

Keywords: Collision detection, weighted least squares, proximity graphs, implicit surfaces.

1 INTRODUCTION

In the past few years, point clouds have had a renaissance caused by the wide-spread availability of 3D scanning technology. Interaction with objects thus represented often requires intersection tests between pairs of objects. Other applications, such as Boolean operations [1] or physically-based simulation [10], require fast construction of points on the intersection curves.

In order to do that, one must define an appropriate surface (even if it is not explicitly reconstructed). The simple weighted least-squares (WLS) definition of point cloud surfaces is quite attractive and can be evaluated very fast [3]. In order to overcome a problem caused by Euclidean distances in the weighting functions, [12] proposed a method that utilizes (conceptually) a Voronoi diagram and a geometric proximity graph to approximate geodesic distances between the query point and the cloud points.

In this paper, we present a method that can quickly find intersection points on objects represented by point clouds. It converges even if the sampling is sparse, compared to the surface areas, and even if the distance between the surfaces contains local minima.

The idea is to utilize a proximity graph over the point clouds and perform interpolation search along geodesic paths through these graphs. The search is initialized by randomized sampling that tries to find two points on one object and on different sides of the other object. Then, our interpolation search converges quickly to an approximate intersection point. Finally, the space surrounding that is sampled to get very accurate (discrete) intersection points.

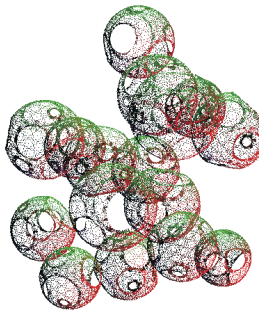


Figure 1: One of our point clouds for benchmarking our novel intersection method ($> 137\,000$ points).

Our new algorithm can be combined very easily with any acceleration data structure for collision detection or intersection construction. For instance, with bounding volume hierarchies [11], the algorithm presented here would be invoked at the leaves.

In the following, we will first give a review of related work. Section 3 gives a quick recap of the WLS surface definition and the proximity graph we are using. Section 4 describes the details of our new algorithm while Section 5 shows its performance.

2 RELATED WORK

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" [2, 3, 4].

WSCG 2005 conference proceedings, ISBN 80-903100-7-9
WSCG'2005, January 31–February 4, 2005
Plzen, Czech Republic.

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 [5].

Geometric queries on point clouds have been studied extensively. An interesting result related to our problem can be found in [7, 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 on the implicit surfaces defined by such object representations. The work most related to ours is [21]. They 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 proximity graph available here. In addition, our new method is much simpler.

In [11] a bounding volume (BV) hierarchy for point cloud collision detection was proposed. The BV traversal first visits leaves where intersections are more likely. Then, a sampling technique similar to [21] determines the intersection points.

An algorithm to perform Boolean operations on solids was presented in [1]. However, their algorithm does not work for surfaces implicitely defined, and it requires closed surfaces.

As mentioned above, our method is based on proximity graphs, which have been studied extensively in the past decade. There is a broad spectrum of them, including the Delaunay graph, nearest-neighbor graph, γ-graph, α-shape, and the spheres-of-influence graph, to name but a few; see [9] for a good survey.

3 IMPLICIT SURFACE MODEL

In this section, we give a quick recap of the weighted least-squares (WLS) method [2, 3], which was originally introduced by McLain [13] in the context of contouring, plus its geodesic extension based on proximity graphs [12].

3.1 Weighted Least Squares

Let a point cloud $\mathscr{P}$ with N points $p_i \in \mathbb{R}^3$ be given. Then, an appealing definition of the surface from $\mathscr{P}$ is the zero set $S = \{x \mid f(x) = 0\}$ of an implicit function

$$f(x) = n(x) \cdot (a(x) - x) \tag{1}$$

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

$$a(x) = \frac{\sum_{i=1}^{N} \theta(x, p_i) p_i}{\sum_{i=1}^{N} \theta(x, p_i)}. \tag{2}$$

Usually, a Gaussian kernel (weight function)

$$\theta(x, p) = e^{-d(x,p)^2/h^2}, \quad d(x, p) = \|x - p\|, \tag{3}$$

is used, but other kernels work as well.

The bandwidth h of the kernel allows us to tune the decay of the influence of the points. It should be chosen such that no holes appear.

The normal $n(x)$ is defined as the direction of smallest weighted covariance, which is the smallest eigenvector of the centered covariance matrix $B(x) = \{b_{ij}(x)\}$ with

$$b_{ij}(x) = \sum_{k=1}^{N} \theta(x, p_k)(e_i(p_k - a(x)))(e_j(p_k - a(x))) \tag{4}$$

where $e_i, i \in \{0, 1, 2\}$ is a basis of $\mathbb{R}^3$.

The above definition can produce artifacts in the surface S, which are mainly caused by the Euclidean distance function $d(x, p)$ that does not take the topology of S into account. This problem can be solved by using a different distance function $d_{\text{geo}}(x, p)$ in (3) that is based on geodesic distances on the surface S. Therefore, a geometric proximity graph can be utilized where the nodes are points $\in \mathscr{P}$. Then, geodesic distances between the points can be approximated by shortest paths on the edges of the graph.

We use the following *geodesic kernel*:

$$\theta(x, p) = e^{-d_{\text{geo}}(x,p)^2/h^2} \tag{5}$$

when computing f by (1)–(4).

3.2 Geodesic Distance Approximation

There is a whole spectrum of different proximity graphs over a set $\mathscr{P}$ of points. We decided to use the the sphere-of-influence graph (SIG) as it has reduced artifacts in WLS point cloud surfaces dramatically [12]. In this section, we will give a short overview of this fairly little known proximity graph [6, 14]. Moreover, we will shortly summarize how to precompute and store the geodesic distances.

The Sphere-of-Influence Graph (SIG). The idea is to connect points if their "spheres of influence" intersect. More precisely, for each point p_i the distance d_i to its nearest neighbor (NN) is determined and two points p_i and p_j are connected by an edge if $\|p_i - p_j\| \leq d_i + d_j$.

As a consequence, the SIG tends to connect points that are "close" to each other relative to the local point density. In noisy or irregularly sampled point clouds, however, a lot of isolated "mini-clusters" can appear, even though there are no holes in the original surface. Because our root bracketing will utilize the graph, it would fail in such a situation.

Therefore, we use the r-SIG($\mathscr{P}$): instead of computing the distance to the NN for each node, we compute the distance to the r-nearest neighbor and then proceed as in the case of $r = 1$. That means, the larger r, the more nodes are directly connected by an edge. In our experience, it seems best to choose $r = 3$ or $r = 4$, and

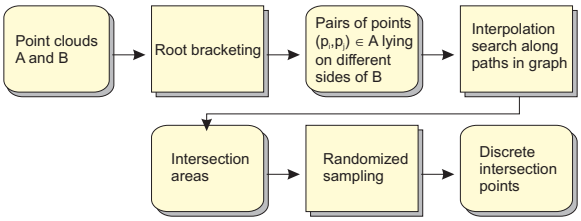


Figure 2: Outline of our point cloud collision detection.

then prune away all "long" edges by an outlier detection algorithm [22].

Precomputing Geodesic Distances. Computing shortest paths on-the-fly during the collision detection process would be, of course, prohibitively expensive, so we pre-compute and store them in a *close-pairs shortest-paths* (CPSP) map [12].

Since the Gaussian (3) decays fairly quickly, we need to store only paths up to some length for defining the surface. The contribution of nodes in Equations 2 and 4 that are farther away can be neglected. That means, for each point p_i we have to run a single-source-shortest path algorithm, but only for points whose influence in p_i is larger than some small threshold.

In [12] it is shown that all these geodesic distances for a whole point cloud of size N can be computed and stored in $O(N)$ time and space.

4 CONSTRUCTING POINTS ON THE INTERSECTION

Given two point clouds A and B, the goal is to determine whether or not there is an intersection, i.e., a common root $f_A(x) = f_B(x) = 0$, and, possibly, to compute a sampling of the intersection curve(s), i.e., of the set $\mathscr{X} = \{x \mid f_A(x) = f_B(x) = 0\}$. Both can be achieved very quickly by exploiting the proximity graph.

First, our algorithm tries to bracket intersections by two points on one surface and on either side of the other surface (see Figure 2). Second, for each such bracket, it finds an approximate point in one of the point clouds that is close to the intersection (see Figure 3). Finally, this approximate intersection point is refined by subsequent randomized sampling. This last step is optional, depending on the accuracy needed by the application.

In the following, we describe each step in detail.

4.1 Root Bracketing

Finding common roots of two (or more) nonlinear functions is extremely difficult [17]. Even more so here, because the functions are not described analytically, but algorithmically.

As mentioned before, our algorithm starts by constructing random pairs of points on different sides of one of the surfaces. The two points should not be too far apart, and, in addition, the pairs should evenly sample the surface.

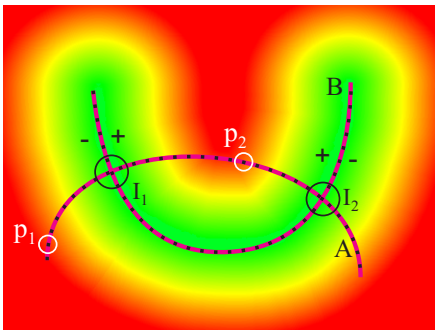


Figure 3: Two point clouds A and B and their intersection spheres I_1 and I_2. Our root finding procedure, when initialized with $p_1, p_2 \in A$, will find an approximate intersection point inside the *intersection sphere* I_1.

An exhaustive enumeration of all pairs is, of course, prohibitively expensive. Therefore, we propose the following randomized (sub-)sampling procedure.

Assume that the implicit surface is conceptually(!) approximated by surfels (2D discs) of equal size [16, 19]. Let $\mathrm{Box}(A,B) = \mathrm{Box}(A) \cap \mathrm{Box}(B)$ and $\overline{A} = A \cap \mathrm{Box}(A,B)$. Then, we want to randomly draw points $p_i \in \overline{A}$ such that each surfel s_i gets occupied by at least one p_i; here, "occupied by p_i" means that the projection of $a(p_i)$ along the normal $n(p_i)$ onto the supporting plane of s_i lies within the surfel's radius.

For each p_i we can easily determine another point p_j (if any) in the *neighborhood* of p_i so that p_i and p_j lie on different sides of f_B. We represent the neighborhood of a point p_i by a sphere c_i centered at p_i.

An advantage of this is that the application can specify the density of the intersection points that are to be returned by our algorithm. From these, it is fairly easy to construct a discretization of the complete intersection curves (for instance, by utilizing randomized sampling again).

Note that we never need to actually construct the surfels, or assign the points from A explicitly to the neighborhoods, which we describe in the following. Section 4.2 describes how to choose the radius of the spheres c_i.

In order to find a $p_j \in A \cap c_i$ on the "other side" of f_B, we use $f_B(p_i) \cdot f_B(p_j) \leq 0$ as an indicator. This, of course, is reliable only if the normals $n(x)$ are consistent throughout space. If the surface is manifold and connected, this can be achieved by a method similar to [8].

Utilizing our proximity graph (which is a supergraph of the nearest-neighbor graph), we can propagate a normal to each point $p_i \in A$. Then, when defining $f(x)$, we choose the direction of $n(x)$ according to the normal stored with the NN of x in A.[1]

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

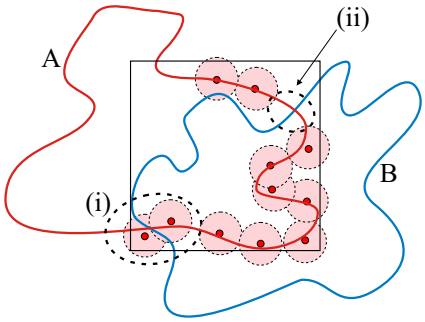


Figure 4: If the spherical neighborhoods c_i (red) are too small, not all collisions can be found. (i) adjoining neighborhoods do not overlap sufficiently, their intersection contains no cloud point. (ii) surface is not covered by neighborhoods c_i.

In order to sample A such that each (conceptual) surfel is represented by at least one point in the sample, we use the following

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

Proof: see Appendix A.

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 generated.

Now, given a point $p_i \in A$, we have to determine another point $p_j \in A \cap c_i$ on the other side of f_B.

This is done by testing $f_B(p_i) \cdot f_B(p_j) \leq 0$ for all points $p_j \in A \cap c_i$. In the next section, we show that $c_i \cap A$ is only a small, constant number of points. Therefore, a point p_j on the other side of f_B can be determined in time $O(1)$ (if it exists). We utilize our proximity graph and a breadth-first search to access the points in the spherical neighborhood c_i.

4.2 Size of Neighborhoods

The radius of the spherical neighborhoods c_i has to be chosen so that, on the one hand, all c_i cover the whole surface defined by A. On the other hand, the intersection with each adjoining neighborhood of c_i has to contain at least one point in A to miss no collisions lying in the intersection of two neighborhoods. The situation is illustrated in Figure 4.

To determine the minimal radius of a spherical neighborhood c_i, we introduce the notion of *sampling radius*.

Definition 1 (Sampling radius)
Let a point cloud A as well as a subset $A' \subseteq A$ be given. Consider a set of spheres, centered at A', that cover the surface defined by A (not A'), where all spheres have equal radius. We define the sampling radius $r(A')$ as the minimal radius of such a sphere covering.

Remember that we draw $n = O(a \cdot \ln a + c \cdot a)$ random and independent points from $\overline{A}$. Let A' denote the point cloud consisting of these random points. Then, spheres with radius $2 \cdot r(A')$ centered at points in A' contain always points of the neighboring spheres and, of course, cover the surface.

The sampling radius $r(A')$ can obviously be estimated as the radius r of a surfel $s_i \in S_A$.

Let F_A denote the surface area of the implicit surface over $\overline{A}$. Then, the surfel radius r can be determined by

$$\frac{F_A}{a} = \pi \cdot r^2 \Rightarrow r = \sqrt{\frac{F_A}{a \cdot \pi}}.$$

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

$$F_A = |\overline{A}| \cdot \pi r(A)^2.$$

Overall, $r(A')$ can be estimated by

$$r(A') = r(A) \cdot \sqrt{\frac{|\overline{A}|}{a}} \approx r(A) \cdot \sqrt{\frac{\mathrm{Vol}(A,B)}{\mathrm{Vol}(A) \cdot a} \cdot |A|}.$$

The size of $\overline{A}$ can easily be estimated depending on the ratio of $\mathrm{Vol}(A)$ and $\mathrm{Vol}(A,B)$, the sampling radius $r(A)$ can easily be determined in the preprocessing.

In [12] it has already been shown that for uniformly distributed points $p_i \in \mathbb{R}^3$ and a sampling radius of $r(A)$ only $O(\lceil\sqrt{2} \cdot m\rceil^2)$ points $\in A$ lie in a sphere with radius $m \cdot r(A)$. If we choose $m = 2r(A')/r(A)$, then at most $O(\lceil\sqrt{2} \cdot 2r(A')/r(A)\rceil^2) = O(1)$ points $\in A$ lie in a spherical neighborhood with radius $2 \cdot r(A')$ because $m = 2r(A')/r(A)$ is constant.

4.3 Interpolation Search

Having determined two points $p_1, p_2 \in A$ on different sides of object B, the next goal is to find a point $\hat{p} \in A$ "between" p_1 and p_2, such that the approximate distance from B is small enough, i.e., $|f_B(\hat{p})| < \varepsilon$. In the following, we will call such a point *approximate intersection point* (AIP). The true intersection curve $f_B(x) = f_A(x) = 0$ will pass close to $\hat{p}$ (usually, it does not pass through any points of the point clouds).

Depending on the application, $\hat{p}$ 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 4.5.

If B does not have boundaries (e.g., holes) and A is sufficiently densely sampled, then there must be a point

```
l, r = 1, n
d_{l,r} = f_B(P_1), f_B(P_n)
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
```

Algorithm 1: Pseudo-code of our root finding algorithm based on interpolation search. P is an array containing the points of the shortest path from $p_1 = P_1$ to $p_2 = P_n$, 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.

$\hat{p} \in A$ lying on the shortest path between p_1 and p_2 for which $|f_B(\hat{p})| < \varepsilon$. Let us assume that f_B is monotone along the path $\overline{p_1 p_2}$ (this can always be ensured by making the surfels small enough). Then, instead of doing an exhaustive search along the path, we could utilize binary search to find $\hat{p}$. Better yet, we can utilize interpolation search, which makes sense here, because the "access" to the key of an element, i.e., an evaluation of f_B, is fairly expensive [20]. The runtime of interpolation search is in $O(\log\log m)$, $m =$ number of elements.

Algorithm 1 for our interpolation search assumes that the shortest paths are precomputed and stored in the CPSP map (Section 3.2). Analogously to [12], it is easy to see that the storage is still linear.

However, in practice, the memory consumption could be too large for huge point clouds. In that case, we can compute the path P on-the-fly at runtime by Algorithm 2. 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).

4.4 Models with Boundaries

If the models have boundaries and the sampling rate of our root bracketing algorithm is too low, not all intersections will be found (see Figure 5). 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 3.2). 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.

```
q.insert(p_1);  clear P
repeat
    p = q.pop
    P.append( p )
    for all p_i adjacent to p do
        if d_geo(p_i, p_2) < d_geo(p_1, p_2) then
            insert p_i into q with priority d_geo(p_i, p_2)
until p = p_2
```

Algorithm 2: This algorithm can be used to initialize P for Algorithm 1 if storing all shortest paths in the CPSP map is too expensive. (q is a priority queue.)

4.5 Precise Intersection Points

If two point clouds are intersecting, our interpolation search returns a set of AIPs. Around each of them, an *intersection sphere* of radius $r = \|x - p_1\|$ where

$$x = \frac{1}{d_1 + d_2}(d_2 p_1 + d_1 p_2)$$

contains a true intersection point (p_1 and p_2 are the points $\in A$ with smallest distance to B lying on different sides of B, $d_i = f_B(p_i)$). The idea is illustrated in Figure 6. If AIPs are not precise enough, then we can sample each such sphere to get more accurate (discrete) intersection points.

More precisely, if a precise collision point's distance from the surfaces is to be smaller than ε_2, we cover a given intersection sphere by s smaller spheres with *diameter* ε_2 and sample that volume by $s \cdot \ln s + c \cdot s$ points so that each of the s spheres gets a point with high probability (see Appendix A). For each of these, we just determine the distance to both surfaces.

Rogers [18] 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 the intersection sphere by spheres with radius $b = \varepsilon_2/2$, we have to choose $a = 2r/\varepsilon_2$, so that $a \cdot b = r$. As a consequence,

$$s = \lceil \sqrt{3} \cdot \frac{2r}{\varepsilon_2} \rceil^3.$$

For example, if we would like to cover an intersection sphere with spheres of *radius* ε, then $\varepsilon_2 = 2 \cdot \varepsilon$ and $s = \lceil \sqrt{3} \cdot r/\varepsilon \rceil^3$.

4.6 Complexity Considerations

In this section we analyze the runtime of our novel approach and the number of evaluations of the implicit function that are necessary to detect all intersections for a given sampling density described by the number a of surfels.

In general, evaluating $f(x)$ takes $O(logN)$ time, even if the support of the kernel is bounded, because the NN of x has to be determined (using, for instance, a kD-tree). Here, fortunately, one evaluation can be done in

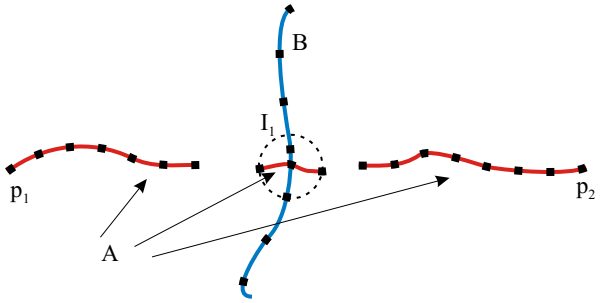


Figure 5: Models with boundaries can cause errors (I_1 could remain undetected), which can be avoided by "virtual" edges in the proximity graph.

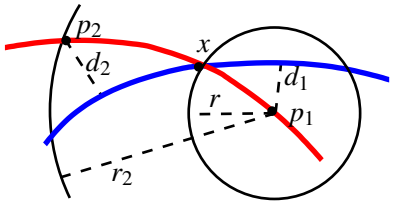


Figure 6: An intersection sphere centered at an AIP p_i. Its radius r can be determined approximately with the help of a second point on the other side of B.

only $O(1)$ time: the root bracketing and interpolation search evaluate $f(x)$ only at points $x \in A \cup B$, and computing the precise intersection points can use a brute force NN search in constant time, starting from the AIP.

Our root bracketing algorithm looks $O(a \ln a)$ times for a pair (p_i, p_j) of points lying on different sides. Each time, $f(x)$ has to be evaluated only $O(1)$ times, as the spherical neighborhood around p_i contains only a constant number of points. As a consequence, $O(a \ln a)$ evaluations of f_B have to be performed which is also the overall runtime of our root bracketing algorithm.

Then, for at most $O(a \ln a)$ many pairs, our interpolation search has to be started. 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 p_i and p_j.

Overall, the f_B has to be evaluated at most $O(a \ln a \cdot \log \log m))$ times. As $N \gg m$ and a is constant, this number can also be bounded by $O(\log \log N)$.

5 RESULTS

We implemented our new algorithm in C++. As of yet, the implementation is not fully optimized. All results were obtained on a 2.8 GHz Pentium-IV.

For timing the performance, we used a set of objects (see Fig. 11), most of them with several resolutions. Benchmarking was performed by the procedure proposed in [23], which computes average collision detection times for a range of distances between two identical objects, which are scaled uniformly so that they fit into a cube of size 2^3.

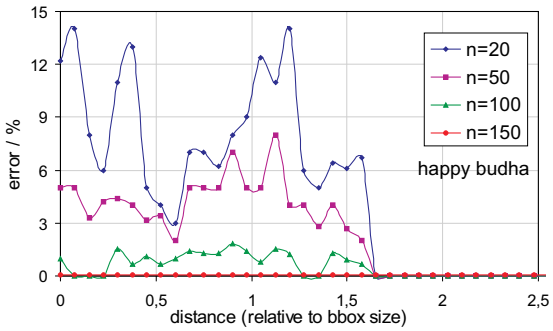


Figure 7: If the sampling density is too small, our approach can miss some intersections ($n = O(a \ln a)$, see Section 4.1).

5.1 Minimal Bracket Density

As mentioned before, 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 can become too large that the normal $n(x)$ flips its sign without x actually changing sides. In that case, our method could fail to find pairs of points on different sides of the surface.

Therefore, we propose to estimate the minimal number of surfels (which directly influences the radius of the spherical neighborhoods) by the following preprocessing procedure. For each distance, a large number of collisions tests is performed, each with a different constellation between the objects. A collisions test stops after the first intersection has been found. Each of these tests is performed with a different sampling density, expressed by the number $n = O(a \ln a)$ (see Section 4.1). Then, we use the minimal sampling density for which all collisions have been found.

The results for one object can be found in Figure 7, 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 minimal number, so that the error rate of all intersection tests for all our models is only 0.1%. This number was used for all further tests.

5.2 Interpolation Search vs Randomized Sampling

In order to evaluate the performance of our new algorithm, we compared it to the simpler randomized sampling technique (RST) proposed in [11]. No BV hierarchies were used.

The number of sample points n_s that have to be generated for the RST can be determined as proposed in Section 4.5, depending on the same ε 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.

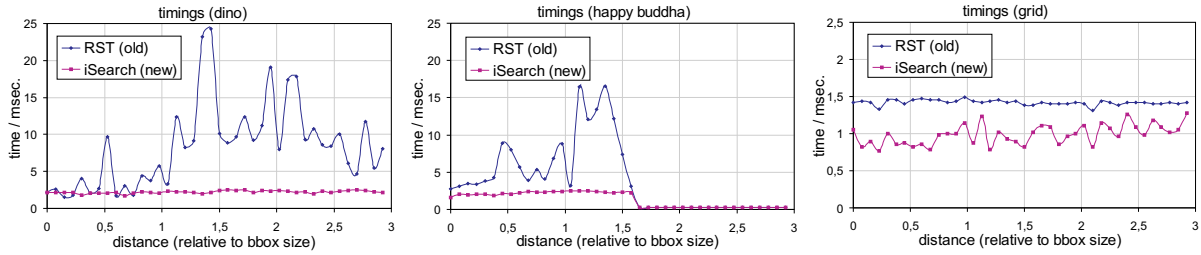


Figure 8: Timings for different models. Comparison of our novel technique and RST [11].

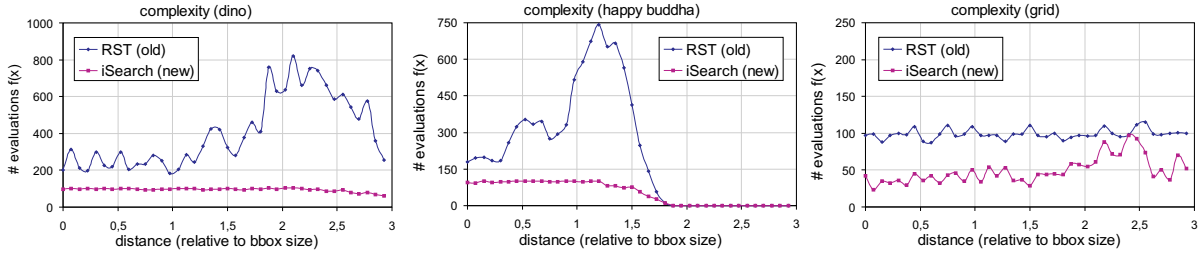


Figure 9: The number of evaluations of $f(x)$ can be decreased by an order of magnitude by our new approach.

However, in the case of non-collision, in particular in the case of small distances between the objects, the runtime 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 8 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 9. Note that the number of evaluations can exceed n_s in the case of the RST, since for each random point two evaluations are necessary.

5.3 Timings depending on Point Density

Figure 10 shows the runtime for detecting *all* intersections between two objects, depending on different densities of the point clouds. We define the 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 4.6.

Note that the CPSP maps (see Section 3.2) 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 2). For all our models, this was at most 10% of the overall runtime. Therefore, one can save a significant amount of memory in the CPSP map by computing array P in Algorithm 1 during run-time.

6 CONCLUSION AND FUTURE WORK

We have presented a novel algorithm for sampling the intersection curves between surfaces defined implicitly by point clouds with the weighted least-squares method plus proximity graph. It can be used, for instance, to accelerate hierarchical collision detection or Boolean operations on this kind of object representation.

Our approach exploits the proximity graph by interpolation search along shortest paths in the graph. The technique of randomized sampling has proven to be efficient for initializing that search.

Our measurements show that the number of function evaluations is reduced by an order of magnitude and a speedup of factor 5–10 is achieved in many cases, compared to a previous randomized sampling technique.

Moreover, theoretical and experimental evidence is given that the runtime grows only as $\log\log N$, ($N =$ the size of the point clouds).

We believe that this work opens up a number of further avenues for future work. Our new approach could be a way to handle *deformable* point clouds, since it does not utilize any spatial acceleration structure and the SIG can be updated in time $O(\log^3 N)$. From a theoretical point of view, a mathematically more rigorous estimation of the minimal sampling density would be appealing.

ACKNOWLEDGEMENTS

This work was partially supported by DFG grant DA155/29-1 “Benutzerunterstützte Analyse von Materialflußsimulationen in virtuellen Umgebungen” (BAMSI), and the DFG program “Aktionsplan Informatik” by grant ZA292/1-1.

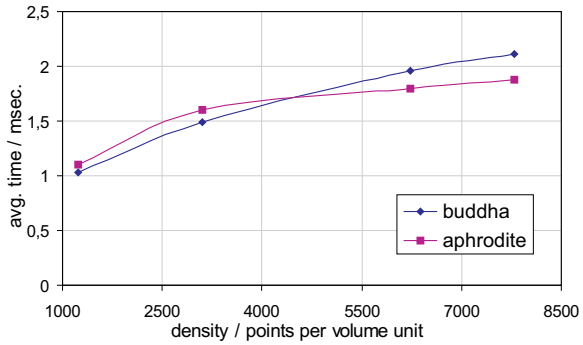
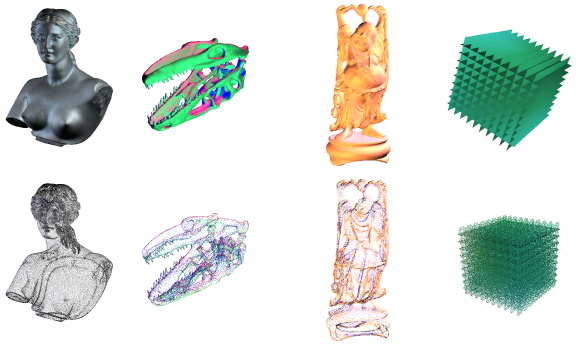


Figure 10: The plot shows the runtime depending on the size of the point clouds. The runtime is the average of all timings for distances between 0 and 1.5.



89 036 27 727 62 299 197 315

Figure 11: Some of the models of our test suite (courtesy of Polygon Tech. Ltd and Stanford). The numbers are the sizes of the respective point clouds.

A PROOF OF LEMMA 1

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 [15, 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. Therefore, $n = a \cdot \ln a + c \cdot a$ points $\in \mathrm{Vol}(A \cap B)$ have to be generated.

To compute the dependence of p on c, we refer to the proof given by Motwani and Raghavan [15, p. 61ff]. They showed that the probability $p = Pr[X \leq n] = e^{-e^{-c}}$ for a sufficiently large number of bins.

REFERENCES

[1] Bart Adams and Philip Dutré. Interactive boolean operations on surfel-bounded solids. In *Proc. of SIGGRAPH*, volume 22, pages 651–656, July 2003.

[2] Anders Adamson and Marc Alexa. Approximating and intersecting surfaces from points. In *Proc. Eurographics Symp. on Geometry Processing*, pages 230–239, Aachen, Germany, June23–25 2003.

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

[4] Anders Adamson and Marc Alexa. On normals and projection operators for surfaces defined by point sets. In *Eurographics Symp. on Point-Based Graphics*, pages 149–155, 2004.

[5] Nina Amenta and Yong Kil. Defining point-set surfaces. In *Proc. of SIGGRAPH*, pages 264–270, 2004.

[6] 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.

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

[8] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *Proc. of SIGGRAPH*, pages 71–78, 1992.

[9] J. W. Jaromczyk and Godfried T. Toussaint. Relative neighborhood graphs and their relatives. In *Proc. of the IEEE*, volume 80, pages 1502–1571, 1992.

[10] Richard Keiser, Matthias Mueller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Contact handling for deformable point-based objects. In *Vision, Modeling, Visualization (VMV)*, pages 315–322, Stanford, USA, November16–18 2004.

[11] Jan Klein and Gabriel Zachmann. Point cloud collision detection. In *Computer Graphics Forum (Proc. of EUROGRAPHICS 2004)*, pages 567–576, 30 August - 3 September 2004.

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

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

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

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

[16] Hanspeter Pfister, Jeroen van Baar, Matthias Zwicker, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proc. of SIGGRAPH*, pages 335–342, 2000.

[17] 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.

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

[19] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proc. of SIGGRAPH*, pages 343–352, 2000.

[20] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1989.

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

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

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