Exact distance computation for deformable objects

Marc Gissler
University of Freiburg, Germany

Udo Frese
University of Bremen, Germany

Matthias Teschner
University of Freiburg, Germany

Abstract

We present a novel approach for the computation of the minimum distance between arbitrarily shaped, triangulated objects. The approach proceeds in two stages. In the first stage, the Gilbert-Johnson-Keerthi algorithm (GJK) is performed. We show how to employ characteristics of the algorithm to efficiently compute lower and upper bounds of the minimum distance between non-convex objects. We further show how to use these bounds to set up a spatial subdivision scheme that computes the exact minimum distance in the second stage of the algorithm. The knowledge of a lower and an upper bound allows for a twofold culling in the second stage. First, only a small part of the simulation domain is considered in the spatial subdivision. Thus, large object parts are culled. Second, the intrinsic properties of the spatial subdivision scheme are employed to further accelerate the computation of the minimum distance for the remaining object parts. The proposed algorithm does not rely on precomputed data structures. Therefore, it is particularly appropriate for deformable objects. Experiments indicate the efficiency of the approach compared to existing algorithms.

Keywords: physics based animation, proximity queries

1 Introduction

Efficient proximity queries - in particular the computation of exact distances - are a crucial task in interactive simulation systems with a variety of application areas such as computational surgery, games, virtual reality, path planning, robotics, and bioinformatics [1, 2, 3]. In general, distances have to be computed for arbitrarily shaped, dynamically moving, rigid or deformable objects.

The distance computation between rigid bodies is commonly accelerated by the precomputation of auxiliary data structures, e. g. surface decomposition [4]. In the context of deformable objects, however, these precomputations are rather difficult to incorporate or even impossible to use.

Our contribution: We propose a novel approach to the computation of the minimum distance between arbitrarily shaped objects. Since offline precomputations are avoided, the method is particularly useful for dynamically deforming objects. We show how to efficiently use GJK to compute lower and upper bounds for the minimum distance between arbitrarily shaped objects. We further show how to employ these bounds to efficiently set up a spatial subdivision scheme that only considers a small part of the simulation domain to determine the exact minimum distance. We illustrate the performance of the proposed algorithm and discuss benefits and drawbacks compared to existing approaches.

2 Related work

A large variety of proximity query algorithms has been proposed over the last decades. They can be classified by the queries that can be performed and by the prerequisites they pose on the object representation. There exist algorithms for collision detection, separation distance computation and penetration depth computation. Excellent surveys can be found in [5, 6, 7]. Since the research on proximity queries is a huge area, we focus our discussion of the related work on approaches for the computation of separation distances.

2.1 Convex objects

Many of the early algorithms exploit the properties of convex sets to be able to formulate a linear programming problem. In [8], Gilbert et al. propose an iterative method to compute the minimum distance between two convex polytopes using Minkowski differences and a support mapping. Extensions of the algorithm handle general convex objects [9] and return a penetration distance [10]. A fast and robust implementation is given by [11] which is also incorporated in the Software Library for Interference Detection (SOLID) [1]. In [12], an algorithm that employs local search over the Voronoi regions of convex objects to descend to the closest point pair is proposed. The approach is used at the lowest level of collision detection in the software package I-Collide [2] [13].

In dynamic environments, all of the approaches mentioned above exploit geometric and time coherence to track the closest points. It is assumed that the displacement of the objects between two time steps is not too large, if the time step is small. In [12], hill climbing is employed to search the neighboring Voronoi regions and return the new closest pairs in nearly constant time. Hill climbing is also used in [10] to find new support vertices more efficiently.

2.2 Non-convex objects

The restriction to convex objects can be overcome in several ways. A non-convex object can be seen as the composition of several convex subparts [8, 12]. The algorithms are then applied to the convex pieces or subparts, respectively. Similarly, a non-convex polyhedron could be decomposed into convex subparts. In most cases, it suffices to decompose the boundary of the polyhedron into convex patches [14]. Thus, surface decomposition can be used to perform proximity queries on general, rigid bounded polyhedra [4]. The surface is decomposed into convex patches and the proximity query algorithms for convex objects can be applied to the patches. To accelerate the pairwise proximity query, the patches can be stored in bounding volume hierarchies. Different types of bounding volumes have been investigated, such as spheres [15, 16], axis-aligned bounding boxes [17], k-DOPs [18] or oriented bounding boxes [19]. Further, various hierarchy-updating methods have been proposed [20], some of them employing the underlying deformation model [21]. An algorithm that employs surface decomposition together with bounding volume hierarchies is integrated in the software package SWIFT++ [3].

Surface decomposition is a nontrivial and time consuming task. In rigid-body dynamics, the objects fortunately have to be decomposed only once. Therefore, surface decomposition is made a preprocessing step. Unfortunately, this is not the case in simulations of deformable models.

2.3 GPU-based proximity queries

Graphics hardware can be used to accelerate various geometric computations. Image-space techniques are employed for detection of collisions [22, 23, 24] as well as self-collisions [25]. Discrete Voronoi and distance fields can be efficiently computed on the GPU [26, 27] which can be used to answer penetration and distance queries [28]. Possible drawbacks of GPU-based approaches are that their accuracy is limited by image precision and depth buffer resolution. In [28], this problem is avoided by using the discrete Voronoi diagram computed in image space only as input to accelerate the proximity computation in object space. On the other hand, the time for read-back of frame buffers takes up considerable time, even on todays graphics hardware. In [24], the amount of read-back is reduced with the introduction of occlusion queries for collision detection.

In contrast to existing approaches, our algorithm focuses on deformable objects with arbitrary shape. A combination of GJK and spatial hashing is used to compute the exact distance between objects. We show that GJK can be used to efficiently compute distance bounds for non-convex objects. We further show that these distance bounds allow for an efficient setup of a spatial hashing scheme for the computation of the exact distance. Preprocessing steps are avoided. Further, arbitrary shapes and arbitrary object movements can be handled. Thus, the proposed scheme is particularly appropriate for the handling of dynamically deforming objects.

3 Algorithm overview

In this section, we give an overview of the algorithm. It computes the minimum distance between two arbitrarily shaped objects. The objects are represented as closed non-convex poly-

[1]http://www.win.tue.nl/ gino/solid/

[2]http://www.cs.unc.edu/ geom/I_COLLIDE/

[3]http://www.cs.unc.edu/ geom/SWIFT++/

hedra in three-dimensional space. The algorithm consists of two stages.

In the first stage, we employ the Gilbert-Johnson-Keerthi (GJK) algorithm [8]. The algorithm efficiently computes the minimum distance between two convex objects. If they are not convex, GJK returns the minimum distance between their convex hulls. In this case, we use the returned distance as a lower bound to the exact minimum distance. Furthermore, the support vectors returned by GJK are part of the surface of the two objects. We use the minimum distance between them as an upper distance bound.

In the second stage, we employ the spatial hashing algorithm of Teschner et al. [29] on the transformed objects. Traditionally, spatial hashing is used to efficiently detect collisions between object primitives. Here, we use this spatial subdivision scheme to efficiently discard pairs of surface primitives with a separation distance outside the interval defined by the lower and upper distance bounds. We compute the minimum distance from the set of primitive pairs within such a hash cell and the adjacent cells. As a result, we get the exact minimum distance between the two objects.

The remainder of this paper is structured as follows: In Section 4, we summarize the most important steps from the GJK algorithm and evaluate its results. The application of spatial hashing is described in Section 5. Results are given in Section 6 to conclude the paper.

4 GJK

In this section we describe the main idea of the Gilbert-Johnson-Keerthi algorithm and evaluate the information that can be drawn from the output of the algorithm. For a more detailed description, see [8].

4.1 Terminologies and definitions

First, we would like to introduce some terminology and give definitions we use in the following sections. We consider pairs of closed non-convex polytopes P and Q in $\mathbb{R}^3$. The surface of the polytope is divided into triangles. A point position $\mathbf{v}$ is given by a three-dimensional vector $\mathbf{v} = (x, y, z)^T$. A *convex combination* is the linear combination of points where all coefficients are non-negative and sum up to one. The set of convex combinations of the points in P gives the *convex hull*, denoted as $\mathrm{conv}(P)$. In affine combinations, the coefficients sum up to one, too, but they are not restricted to be non-negative. Points are *affinely independent*, if none is an affine combination of the others. Furthermore, a *simplex* in $\mathbb{R}^n$ denotes the convex hull of a set of $(n+1)$ affinely independent points. For example, a line is a simplex in $\mathbb{R}^1$, a triangle in $\mathbb{R}^2$ and a tetrahedron in $\mathbb{R}^3$. For a vector $\mathbf{v} \in \mathbb{R}^3$, the length of $\mathbf{v}$ is given by the Euclidian norm. The distance between P and Q is given by:

$$\mathrm{dist}(P, Q) := \min\{\|\mathbf{p} - \mathbf{q}\| : \mathbf{p} \in P, \mathbf{q} \in Q\}, \tag{1}$$

with $\mathbf{p}$ and $\mathbf{q}$ being point positions. Two objects intersect, iff $\mathrm{dist}(P, Q) = 0 \Leftrightarrow P \cap Q \neq \emptyset$. We now give a short description of the algorithm.

4.2 Simplex computation

For a pair of two convex objects P and Q, the GJK algorithm iteratively computes the minimum distance between the two objects. In each iteration step k, the algorithm constructs for each object a simplex from a set of surface points and computes the minimum distance between these two simplices S_k^P and S_k^Q. The algorithm ensures that the minimum distance between the simplices in the current iteration step is smaller than in the previous iteration step: $\mathrm{dist}(S_k^P, S_k^Q)_k < \mathrm{dist}(S_{k-1}^P, S_{k-1}^Q)_{k-1}$. If they are equal or within a certain threshold, the algorithm has converged to the minimum distance between the two objects. For convex objects, the simplices from the last iteration correspond to a surface primitive of the respective objects. For non-convex objects, GJK computes the minimum distance between the convex hulls of P and Q and the simplices correspond to surface primitives only in convex regions . We evaluate this observation in the next section.

4.3 Lower and upper distance bound

In this section, we consider the case where GJK returns the minimum distance between the convex hulls of non-convex objects, not the minimum distance between the objects themselves. In this case, we derive a lower and upper distance bound for the minimum distance between P and Q from the information returned by the GJK algorithm, namely the closest pair of points $(\mathbf{p}, \mathbf{q})$ on the convex hulls of P and Q and the simplices S^P and S^Q. We denote the vector that connects $\mathbf{p}$ and $\mathbf{q}$ by $\mathbf{d} := \mathbf{p} - \mathbf{q}$. It is the normal of the maximum-margin hyperplane h, which maximally separates P and Q. Two parallel hyperplanes can be constructed that separate $\mathrm{conv}(P)$ and $\mathrm{conv}(Q)$, the *margins* (see Fig. 1). The points in the S^P and S^Q lie on these margins. They are the support points and lie on the boundaries of P and Q, respectively. Note that $\mathbf{d} = 0$, if the two convex hulls intersect, i. e. $\mathrm{conv}(P) \cap \mathrm{conv}(Q) \neq \emptyset$. In this case, the algorithm cannot compute the minimum distance.

The space delimited by the two margins contains no points from P and Q. We conclude, that GJK returns a lower distance bound $\mathrm{dist}_{lower}(P, Q)$. If the simplices constructed by the support vectors not only lie on the convex hulls, but on the boundaries of the objects as well, it is also the minimum distance between the two objects. This can be verified quite easily and is a termination criterion for our overall distance algorithm (see Fig. 1). Otherwise, at least one of the simplices lies in a concave region of P or Q.

Now, we search for an upper distance bound $\mathrm{dist}_{upper}(P, Q)$. We consider the support points in S^P and S^Q from the last iteration of the GJK descend. Unlike the points of the closest point pair, the support points are guaranteed to lie on the object boundaries. Therefore, the distance between any pair of support points is also a distance between P and Q. We take the pair of support points with the smallest distance and set it as the upper bound (see Fig. 2). Obviously, $\mathrm{dist}_{lower}(P, Q) \leq \mathrm{dist}_{upper}(P, Q)$ holds.

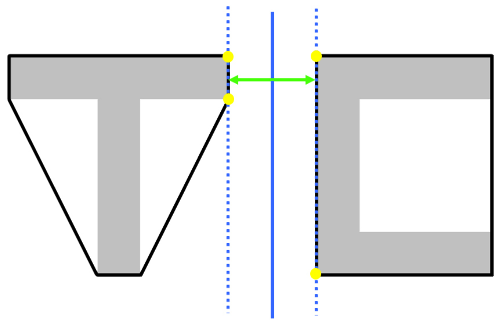

Figure 1: GJK computes the minimum distance (green arrow) between the convex hulls (black) of two objects (grey area). The support points (yellow dots) are returned by GJK in the form of two simplices. They lie both on the margins (dashed blue lines) and on the surfaces of the objects, respectively. The margins are parallel to the maximum-margin hyperplane (blue line).

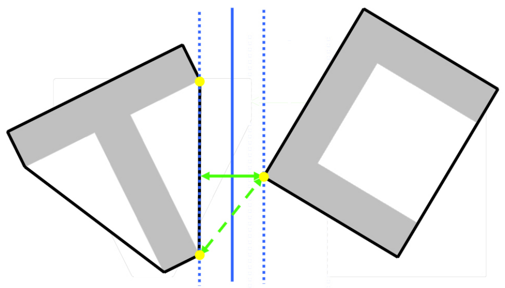

Figure 2: In comparison to Fig. 1, the simplex returned for the left object (dotted black line) does not lie on the surface of the object. Thus, the returned distance (green arrow) is a lower bound of the exact minimum distance. We compute an upper distance bound (dotted green arrow) as the minimum distance between the support points (yellow dots).

With the lower and upper distance bound established, the goal is to efficiently cull away primitive pairs with a distance outside those bounds. Only the distances between the remaining primitive pairs have to be computed to find the minimum distance. We describe the culling algorithm in the following section.

5 Spatial subdivision

In this section, we show how to use the lower and upper distance bounds computed in stage one to efficiently set up a spatial hashing scheme that only considers a small part of the simulation domain to determine the minimum distance. This small part denotes the region of interest. We employ the spatial hashing algorithm described in [29].

The spatial hashing algorithm subdivides our simulation domain using a regular grid. A grid cell is represented by an axis-aligned bounding box (AABB). The size of the grid cell along every direction of the coordinate frame is given by the vector $\mathbf{s}^{cell} = (s_x^{cell}, s_y^{cell}, s_z^{cell})^T$.

As we will detail in the following, $\mathbf{s}^{cell}$ is chosen, such that the points of the closest point pair lie in the same or adjacent grid cell. We utilize the lower and upper distance bounds, the support vectors and the maximum-margin hyperplane which have been derived from the result of the GJK algorithm. First, we transform the objects into a new coordinate frame in order to align the maximum-margin hyperplane with the xy-plane and its normal with the z-axis. Thus, the grid cells are in alignment with the region of interest and every point in P has a z-value greater than $\frac{1}{2} \cdot \text{dist}_{lower}(P, Q)$ and every point in Q has a z-value smaller than $-\frac{1}{2} \cdot \text{dist}_{lower}(P, Q)$, or vice versa. This is due to the margins that are now parallel to the xy-plane. To determine s_z^{cell}, we consider the point pair $(\mathbf{p}, \mathbf{q})$ with $\mathbf{p} \in P$ and $\mathbf{q} \in Q$. If $|p_z| > \text{dist}_{upper}(P, Q) - \frac{1}{2} \cdot \text{dist}_{lower}(P, Q)$, the distance between the points is greater than the upper bound: $\|\mathbf{p} - \mathbf{q}\| \geq \text{dist}_{upper}(P, Q)$. The same holds for $|q_z|$. Thus, we set $s_z^{cell} = 2 \cdot \text{dist}_{upper}(P, Q) - \text{dist}_{lower}(P, Q)$. To determine s_x^{cell} and s_y^{cell}, let $\mathbf{t} = (t_x, t_y, t_z)^T := \mathbf{r} - \mathbf{s}$ be the vector that connects the support points $\mathbf{r} \in P$ and $\mathbf{s} \in Q$ with $\|\mathbf{t}\| = \|\mathbf{r} - \mathbf{s}\| = \text{dist}_{upper}(P, Q)$. As $\mathbf{r}$ and $\mathbf{s}$ are support points and lie on the margins, we know that $t_z = \text{dist}_{lower}(P, Q)$. We now investigate which point pairs$(\mathbf{p}, \mathbf{q})$ with $\mathbf{p} \in P$, $\mathbf{q} \in Q$ can be excluded from the exact distance computation. As $\|\mathbf{t}\| = \text{dist}_{upper}(P, Q)$, $(\mathbf{p}, \mathbf{q})$ can be discarded if $\|\mathbf{p} - \mathbf{q}\| > \|\mathbf{t}\|$. Since $|t_z| = \text{dist}_{lower}(P, Q)$, we know that $|p_z - q_z| \geq |t_z|$ for every point pair $(\mathbf{p}, \mathbf{q})$. Thus, if we postulate

$$\sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} > \sqrt{t_x^2 + t_y^2}, \quad (2)$$

we get $\|\mathbf{p} - \mathbf{q}\| > \|\mathbf{t}\|$, and we can discard the point pair $(\mathbf{p}, \mathbf{q})$. Therefore, we choose $s_x^{cell} = s_y^{cell} := \sqrt{t_x^2 + t_y^2}$. As triangles are generally not aligned to the hash cells, we always have to consider a cell together with its eight neighbors in x- and y-direction.

With the dimension of the cell computed along every axis, we can discretize the AABBs of the triangles of P and Q. In the first step, we loop over the triangles of P. If none of the extreme points of the AABB of the triangle has a value smaller than $\frac{1}{2} \cdot s_z^{cell}$, we discard the triangle and no hash entry is generated (see Fig. 3). Otherwise, we generate a hash cell entry for the triangle in the hash table. In the second step, we loop over the triangles of Q. We invert the z-values of the extreme points of the AABB and check, whether the z-value of one of the extreme points is smaller than $\frac{1}{2} \cdot s_z^{cell}$. If this is the case, we compute the hash index of the triangle in the hash table. For each triangle of P that has been stored under the same index and its neighbors in step one, we check for a new minimum distance (see Fig. 3).

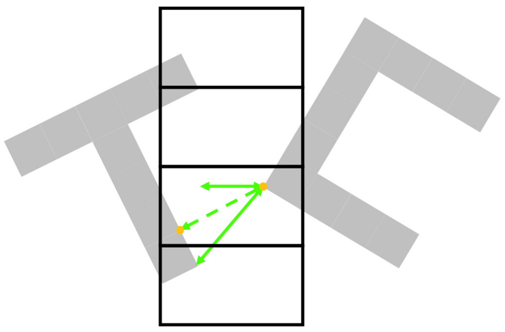
Figure 3: The lower and upper distance bounds (green lines) determine the size of the hash cells (black rectangles). Only the pairs of primitives inside adjacent cells are tested for the minimum distance (dashed green line).

5.1 *Optimizations*

The grid cell size determines the number of triangles that are hashed to the same hash index. If the cells are large, i. e. significantly larger than the size of a triangle, along any coordinate axis, possibly more triangles are hashed to the same index. Conversely, if the cells are small, i. e. significantly smaller than the size of a triangle, along any coordinate axis, a triangle is hashed to a large number of cells, producing many hash entries.

We propose two improvements to adjust the grid cell size along any coordinate axis without the loss of triangle pairs that may support the pair of closest points. The first addresses the number of hash entries of a triangle. The average edge length l_{avg}^e of all triangles in the scene is the empirically optimal size to produce a minimal number of hash entries [29]. As such, we do not allow the cell size to be smaller than a certain threshold δ with $\delta = l_{avg}^e$ along the x- and y-axes. Thus, a triangle produces a number of hash table entries within reasonable magnitude. On the other hand, the number of triangles that are hashed to the same cell due to the increased threshold increases only slightly, since the hash cell size is still smaller than the average edge length. For the second improvement, we keep track of the current upper distance bound while we compute distances between pairs of hashed and queried triangles. As soon as we find a new upper bound, we adjust the grid cell size and hash the triangles of P again. The triangles of Q that were not tested up to this point are then queried using the new grid cell size.

Thus, the number of triangle pairs is constantly reduced. This rehashing can be done very fast (see Sec. 6).

6 Results

We have implemented a variety of scenarios to evaluate and compare the performance of our approach. We have integrated the algorithm into a deformable modeling approach for tetrahedral meshes [30]. The distance queries are performed on the surface faces of the tetrahedral mesh. For comparison, we use the SWIFT++ [4] software package. All timings have been performed on an Intel Core 2 PC, 2.13 GHz with 2 GB of memory. The code is not parallelized.

We highlight the performance of our algorithm on various benchmarks with multiple arbitrarily shaped objects, similar in the sense of [28]. The set of benchmarks include: (1) a pair of mushrooms (see Fig. 4), (2) a pair of cows (see Fig. 5), (3) a pair of teddy bears and (4) a set of four deforming teddies (see Fig. 6). Our algorithm involves no offline preprocessing and computes the minimum distance between all object pairs. In scenes (1) - (3), each of the objects randomly rotates around its center of mass. The objects vary in the number of surface triangles. In scene (4), distances are computed for the six object pairs. Table 1 provides the results of the benchmarks. The average computation time was taken from the distance computation of 1000 consecutive frames.

Benchmark	# of triangles	avg. [ms]
(1) Mushroom	32000	90
(2) Cow	12000	29
(3) Teddy	4400	1.9
(4) Teddies	8800	7.5

Table 1: Timings of the benchmarks: The average distance computation time per frame is given in milliseconds. The objects consist of up to 16000 surface triangles.

The minimum computation time is governed by the GJK algorithm. It varies between 0.08 and 0.6 milliseconds in the first three benchmarks, depending on the number of surface triangles. Thus, finding the lower and upper distance bounds (see Fig. 4) takes only a very small portion of the computation time. Most of the

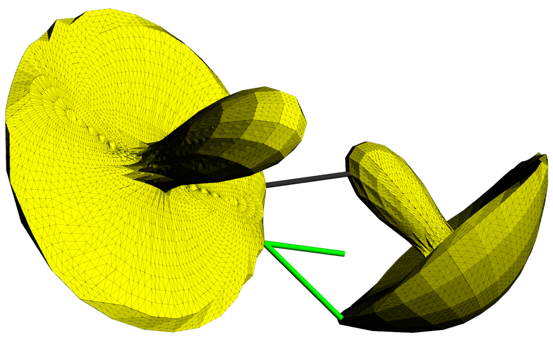

Figure 4: Lower and upper distance bound (green lines) are computed between the convex hulls and the support vectors of the two mushrooms. The exact minimum distance (black line) lies within these bounds.

computation time is spent on the distance computation between the triangle pairs. Therefore, the upper distance bound specifies the overall computation time, since it determines the grid cell size and, thus, the culling of triangle pairs that can be omitted. The triangle culling and the insertion of the remaining triangle pairs into the grid cells of interest is very efficient (see Fig. 5). Thus, we constantly resize the grid cells and rehash, after a new upper bound is found (see Sec. 5.1).

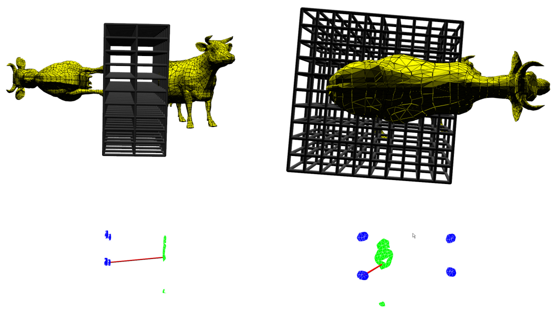

Figure 5: A front and a side view of Benchmark (2). Upper row: Distance pairs are computed between triangles pairs that lie in the same grid cell (black boxes) or one of its neighbors. Bottom row: Visualization of the triangles that have been hashed (blue) and queried (green). One of the triangle pairs gives the minimum distance (red line).

For comparison, we have executed Benchmark 2 in the software package SWIFT++. The package decomposes the surface of a non-

[4]http://www.cs.unc.edu/ geom/SWIFT++/

convex object into convex parts and constructs a bounding volume hierarchy (BVH) for this decomposition. For the cow model, the decomposition took 240 milliseconds and the construction of the BVH took 630 milliseconds. The minimum distance computation took one millisecond. For deformable objects, all three steps have to be performed in every simulation step, which gives a total of 871 milliseconds. Please note that the surface decomposition and the construction of the BVH are preprocessing steps in SWIFT++. Therefore, they are probably not optimized. Nevertheless, the timings indicate that the decomposition is less suitable for online computations in the context of deformable objects.

Since our algorithm is executed on the CPU, potential bottlenecks of GPU-based approaches are avoided. In e. g. [27], it is stated that the data readback from the GPU can be 20-30 ms. Since data readback is an issue in all GPU-based approaches, this indicates that our algorithm outperforms GPU-based approaches for scenarios with a geometric complexity of several thousand triangles.

In Benchmark 4, we demonstrate the applicability of our algorithm to deformable objects. Four teddy bears are thrown into the scene and the minimum distances are computed for the six object pairs. The scene consists of a total of 8800 surface triangles and the average distance computation time is 7.5 milliseconds. This indicates that the algorithm is suitable for interactive applications (see Fig. 6).

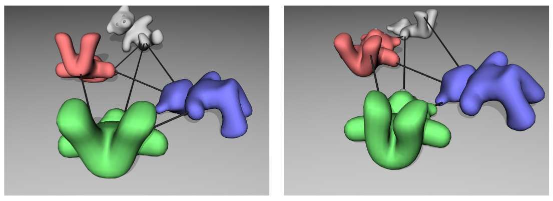
Figure 6: Minimum distances (black lines) are computed between a set of deformable teddy bears. The overall distance computation time per frame is 7.5 milliseconds on average.

6.1 Limitations

As stated in Section 4.3, the algorithm does not compute the exact minimum distance in the case of overlapping convex hulls. This issue is addressed in our current research.

7 Conclusion

We have presented an approach for the minimum distance computation between pairs of objects that proceeds in two stages. In the first stage, we employ the results of the GJK algorithm to compute lower and upper bounds for the minimum distance. We use these bounds to define the grid cell size for the spatial hashing algorithm we employ in the second stage. This enables the efficient culling of large parts of the simulation domain. Thus, only a small portion of triangle pairs has to be considered in the actual pairwise distance computation. We have illustrated the efficiency of the proposed algorithm in a set of benchmark scenarios.

8 Acknowledgments

This work has been supported by the German Research Foundation (DFG) under contract number SFB/TR-8.

References

[1] J.-C. Lombardo, M.-P. Cani, and F. Neyret. Real-time collision detection for virtual surgery. In *Proc. of Computer Animation*, pages 82–90, 1999.

[2] J.-H Youn and K. Wohn. Realtime collision detection for virtual reality applications. In *IEEE Virtual Reality Anual International Symposium*, pages 415–421, 1993.

[3] I. Lotan, F. Schwarzer, D. Halperin, and J.-C. Latombe. Efficient maintenance and self-collision testing for kinematic chains. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*, pages 43–52, New York, NY, USA, 2002. ACM Press.

[4] S.A. Ehmann and M.C. Lin. Accurate and fast proximity queries between polyhedra using surface decomposition. *Computer Graphics Forum (Proc. of Eurographics'2001)*, 20(3):500–510, 2001.

[5] M. C. Lin and D. Manocha. *Handbook of Discrete and Computational Geometry*, chapter 35, pages 787 – 806. CRC Press, 2004.

[6] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann (The Morgan Kaufmann Series in Interactive 3-D Technology), 2004.

[7] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 24(1):61 – 81, 2005.

[8] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation*, 4(2):193–203, 1988.

[9] E.G. Gilbert and C.-P. Foo. Computing the distance between general convex objects in three-dimensional space. *Robotics and Automation, IEEE Transactions on*, 6(1):53–61, 1990.

[10] S. Cameron. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. *IEEE International Conference on Robotics and Automation*, 4:3112–3117, 1997.

[11] G. van den Bergen. A fast and robust GJK implementation for collision detection of convex objects. *J. Graphics Tools*, 4(2):7–25, 1999.

[12] M.C. Lin and J.F. Canny. A fast algorithm for incremental distance calculation. In *IEEE International Conference on Robotics and Automation*, pages 1008–1014, 1991.

[13] J.D. Cohen, M.C. Lin, D. Manocha, and M. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–196, New York, NY, USA, 1995. ACM Press.

[14] B. Chazelle, D.P. Dobkin, N. Shouraboura, and A. Tal. Strategies for polyhedral surface decomposition: An experimental study. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 297–305, New York, NY, USA, 1995. ACM Press.

[15] S. Quinlan. Efficient distance computation between non-convex objects. *IEEE International Conference on Robotics and Automation*, 4:3324–3329, 1994.

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

[17] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *J. Graphics Tools*, 2(4):1–13, 1997.

[18] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[19] S. Gottschalk, M.C. Lin, and D. Manocha. OBB-Tree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180, New York, NY, USA, 1996. ACM Press.

[20] T. Larsson and T. Akenine-Moeller. Collision detection for continuously deforming bodies. In *Eurographics*, pages 325 – 333, 2001.

[21] J. Spillmann, M. Becker, and M. Teschner. Efficient updates of bounding sphere hierarchies for geometrically deformable models. *J. Visual Communication and Image Representation*, 18(2):101–108, 2007.

[22] Y.J. Kim, M.A. Otaduy, M.C. Lin, and D. Manocha. Fast penetration depth computation for physically-based animation. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 23–31, New York, NY, USA, 2002. ACM Press.

[23] D. Knott and D.K. Pai. CInDeR: Collision and interference detection in real-time using graphics hardware. In *Proc. of Graphics Interface*, pages 73–80, 2003.

[24] N.K. Govindaraju, S. Redon, M.C. Lin, and D. Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 25–32, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[25] B. Heidelberger, M. Teschner, and M. Gross. Detection of collisions and self-collisions using image-space techniques. In *WSCG*, pages 145–152, 2004.

[26] K. Hoff, J. Keyser, M.C. Lin, and T. Manocha, D. andCulver. Fast computation of generalized voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, New York, NY, USA, 1999. ACM Press.

[27] A. Sud, N. Govindaraju, R. Gayle, and D. Manocha. Interactive 3d distance field computation using linear factorization. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 117–124, New York, NY, USA, 2006. ACM Press.

[28] A. Sud, N. Govindaraju, R. Gayle, I. Kabul, and D. Manocha. Fast proximity computation among deformable models using discrete Voronoi diagrams. *ACM Trans. Graph.*, 25(3):1144–1153, 2006.

[29] M. Teschner., B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Vision, Modeling, Visualization VMV'03, Munich, Germany*, pages 47 – 54, 2003.

[30] M. Müller and M. Gross. Interactive virtual materials. *Proceedings of the 2004 conference on Graphics interface*, pages 239–246, 2004.