kDet: Parallel Constant Time Collision Detection for Polygonal Objects

René Weller, Nicole Debowski and Gabriel Zachmann

University of Bremen, Germany

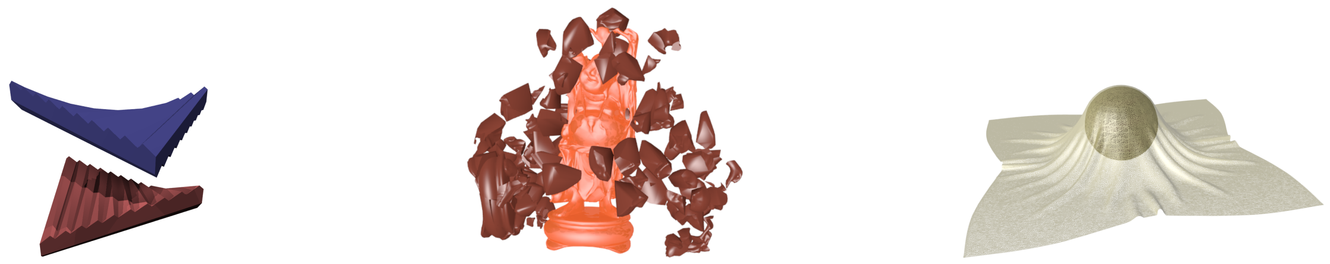


Figure 1: *Specially constructed objects like Chazelle polyhedra (left) realize a quadratic number of intersecting pairs of polygons in the worst case. Our novel algorithm supports inter- and intra-object collision detection for real-world polygon soups, including topology changes of fracturing objects (middle) and deformable objects (right), in parallel constant time.*

Abstract

We define a novel geometric predicate and a class of objects that enables us to prove a linear bound on the number of intersecting polygon pairs for colliding 3D objects in that class. Our predicate is relevant both in theory and in practice: it is easy to check and it needs to consider only the geometric properties of the individual objects – it does not depend on the configuration of a given pair of objects. In addition, it characterizes a practically relevant class of objects: we checked our predicate on a large database of real-world 3D objects and the results show that it holds for all but the most pathological ones.
Our proof is constructive in that it is the basis for a novel collision detection algorithm that realizes this linear complexity also in practice. Additionally, we present a parallelization of this algorithm with a worst-case running time that is independent of the number of polygons. Our algorithm is very well suited not only for rigid but also for deformable and even topology-changing objects, because it does not require any complex data structures or pre-processing.
We have implemented our algorithm on the GPU and the results show that it is able to find in real-time all colliding polygons for pairs of deformable objects consisting of more than 200k triangles, including self-collisions.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

1. Introduction

From a theoretical point of view, finding collisions between a pair of polygonal objects is trivial: we simply check each polygon of one object against all polygons of the other object. In practice, however, this so-called *all pairs weakness* leads to a worst-case running time quadratic in the number of polygons. Obviously, this is not an option in practice. Hence, a lot of work has been spent on developing sophisticated acceleration data structures to reduce the number of potentially colliding polygon pairs in most cases, such as bounding volume hierarchies, which can quickly prune parts of the objects that cannot collide. For many cases and configurations, these data structures work reasonably well. Unfortunately, the quadratic bound on the number of intersecting pairs of polygons is tight: it can be reached by objects such as the Chazelle polyhedron (see Figure 1) [Cha84]: each triangle of one Chazelle polyhedron can intersect all triangles of the other one. In practice, this worst case does not occur very often, but until now, it is impossible to predict in advance for what kinds of objects and in which configuration it could occur. In physically-based simulations we will probably notice a stuttering in the framerate, but in case of time-critical applications like robotics or haptics, exceeding a certain time budget can lead to serious problems.

For the first time, we define a novel geometric predicate that allows to characterize a class of polygonal objects that will not exhibit the quadratic worst-case behavior during collision detection. One nice feature of our predicate is that it depends only on the set of polygons of a single object, and not on the configuration of a pair of objects, i.e. on their position and orientation. It allows us to prove a linear worst-case bound on the number of intersecting pairs of polygons for objects that fulfill it. The main idea behind our predicate is the simple observation that for "normal" objects, the number of neighbors (in some sense, which is to be defined later) of each polygon is usually limited. This is based on a new notion of neighborhood, which takes into account not only the topology of the mesh but also its volumetric configuration. Our predicate is not only of theoretical interest: we tested it on a large database of thousands of 3D objects and found that almost all objects fulfill it, except a few pathological ones.

Moreover, our proof of the linear complexity implies a novel algorithm for collision detection that realizes a worst-case running time that is independent of the number of polygons. Our algorithm, called *kDet*, can be easily parallelized which leads to a worst-case constant complexity of parallel running time by using only a linear number of processors. Another advantage is that our algorithm does not require any complicated pre-processing or sophisticated data structures. Hence, it is also perfectly suited for collision detection of deformable objects and it can easily handle fracturing objects and even the insertion or deletion of polygons during runtime.

We have implemented kDet using CUDA and we measured the performance with a number of challenging scenarios, using both deformable and fracturing objects. Our results show that our algorithm can handle complex scenarios consisting of hundreds of thousands of polygons, including self-collision detection, in realtime. We compared it to a GPU implementation of a state-of-the-art parallel algorithm for all-pairs discrete collision detection and our algorithm outperformed it by more than a factor of four.

2. Related Work

In this section we will briefly summarize works that are related to our paper. We start with an overview on previously published theoretical results in the field of collision detection. Because a complete review on all existing collision detection algorithms would exceed the scope of this paper, we focus on the most fundamental algorithms and recent works that use the GPU.

2.1. Theoretical Results

There is a large body of literature on algorithms and acceleration data structures to reduce the running time in practice. However, there are very little theoretical results on the complexity of the problem of collision detection. There mainly exist some results on special objects such as convex polyhedra. For instance, [DK90] used a hierarchical representation of convex polyhedra to show that the distance between two of them can be computed in $\mathcal{O}(\log(\mid P \mid) \cdot \log(\mid Q \mid))$ with $\mid P \mid$ and $\mid Q \mid$ being the number of faces of P and Q respectively. If closest features of polyhedra based on Voronoi regions are considered [LC91], the worst-case running time for finding the distance is linear. If convex polyhedra undergo only translations, the running time is $\mathcal{O}(n^{\frac{8}{5}+\varepsilon})$ [ST95] and $\mathcal{O}(n^{\frac{5}{3}+\varepsilon})$ for rotational movements of at most the second degree. Later, a generalization for more flexible movements in $o(n^2)$ have been made [ST96].

For pairs of general polygonal objects, [WKZ06] showed an expected running time of $\mathcal{O}(n)$ or $\mathcal{O}(\log(n))$, depending on the overlap of the root bounding volumes and the diminishing factor of the AABB hierarchy that was used in the proof. However, the running time depends on the respective bounding volume hierarchy and on the configuration of the objects, i.e. their position and orientation, not on the object itself. When using other object representations instead of polygons, e.g. sphere packings, is possible to prove a linear complexity for the number of collisions [WFZ13]. Unfortunately, these polydisperse sphere representations only support watertight objects and thus, can be hardly applied to general scenarios.

2.2. Collision Detection on the GPU

In the past, collision detection algorithms have been implemented mostly on the CPU. While the worst-case running time does not change for polygonal objects, acceleration data structures have been used to speed up collision queries for regular cases. Those acceleration data structures include bounding volume hierarchies based on axis aligned bounding boxes [vdB98] or spheres [PG95] among others, or space partition trees such as octrees [BT95], k-d trees [Ben75] or binary space partitioning trees [PY90]. For deformable objects those hierarchical data structures have to be rebuilt or at least updated. This has been done by refitting AABBs according to the surface area heuristic [WBS07], previously visited nodes [LAM06] or independently of other nodes in the case of wrapped sphere volumes [JP04]. Mesh representations based on tetrahedra have also been used with hierarchical grids [EL07] for fewer occupied grid cells per primitive.

The introduction of programmable GPUs allowed completely new algorithms. First, the depth buffer was used to determine overlapping primitives in conjunction with a color buffer for the collision response [VSC01], while the stencil buffer was used in CInDeR [KP03] to count ray-face intersections resulting from ray casts. CULLIDE [GRLM03] and its successor Quick-CULLIDE [GLM05] only used the image space for filtering potential collision candidates to reduce the errors caused by the 2.5D projection in image space. Moreover, image space algorithms have been improved by using a layered depth image per dimension to obtain a collision volume [AFC*10].

Although fully programmable, the architecture of modern GPUs is not well-suited for deep recursion. Hence, traditional BVH-based methods have been adjusted accordingly. A hybrid method has been proposed [PKS10] which executed the BVH construction and updates on the CPU and the actual collision queries on the GPU. An entirely GPU method used linear ordering with a surface area

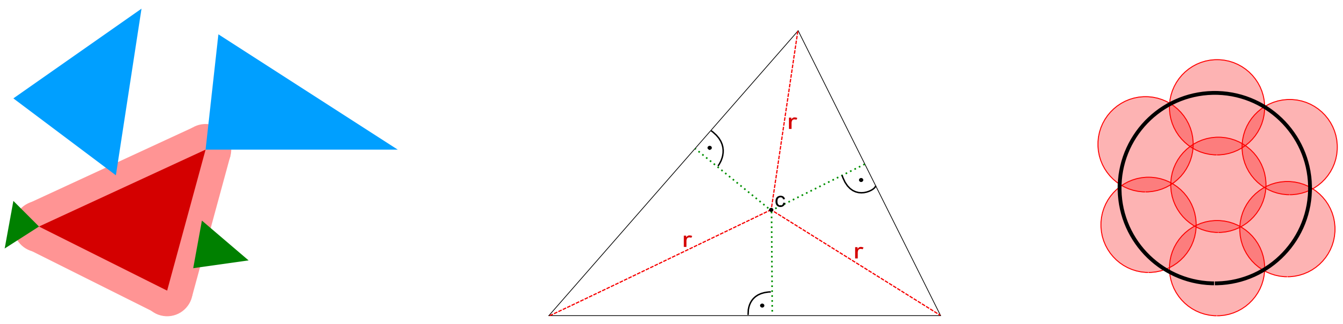


Figure 2: *Left: in this situation, the red triangle (dark red) is 3-free: the green triangles don't count, because their minimum enclosing sphere is smaller, while the two larger triangles (blue) intersect the Minkowski sum (light red). Middle: partitioning of the triangle in the acute case. The center c of the circumcircle with radius r is the intersection point of the perpendicular bisectors (green). The bisectors and lines from the center to the vertices (red), which have length r, divide the triangle into six disjoint right triangles. They can be covered by three circles that have diameter r. Right: a circle of diameter d can be covered by seven circles that have half of it's diameter.*

heuristic for BVH construction with AABBs [LGS*09] and was later adjusted for OBBs and rectangle swept spheres as well as self-collisions [LMM10]. Sweep and prune algorithms are not restricted by recursion and have been proposed using principal component analysis for an optimal sweeping direction for rigid objects and particles [LHLK10] and deformable objects [MZ15]. There also exists an approach based on adaptive octrees [WLZ14].

Recent works often concentrate on continuous collision detection (CCD) rather than all-pairs discrete collision detection (DCD). For instance [TTWM14] presented an approach that uses Bernstein sign classification on the GPU to accelerate continuous collision elementary tests. This method was later extended to support the computation of tight error bounds that typically arise when using finite-precision arithmetic [WTTM15]. CCD is often used for cloth simulation whereas DCD is preferred for CAD, path planning, Virtual Reality or force feedback. Some CCD algorithms like the work presented by Tang et. al [TMLT11], which is based on stream compaction and hierarchical culling techniques, can be also extended to DCD.

3. Our Geometric Predicate

In this section, we establish the theoretical basis of our novel linear sequential time collision detection method. First, we consider only triangulated objects. In this case, we are able to prove a slightly better bound than for arbitrary polygonal objects. However, we will extend our definitions and theorems to arbitrary polygons in the second part of this section.

3.1. The Triangle Case

We start with the definition of our geometric predicate that allows us to prove a linear number of intersections for objects that fulfill it.

What makes the analysis of the number of potentially colliding triangles so difficult is mainly the embedding of a 2-manifold object into 3-dimensional space, which allows for the stacking of an infinite number of 2D triangles into a small 3D volume. Obviously, this is an artificial scenario. Hence, our geometric predicate aims to exclude these cases. The main idea behind the definition of our geometric predicate is to ensure that each triangle shares a certain volume of its environment only with a *limited* number of other triangles. We define this environment with respect to the size of the triangle, or more precisely, with respect to the triangle's minimum enclosing sphere.

Definition 3.1 Let $t \in A$ be a triangle in a triangle set A and integer $k > 0$ some constant. Let s be a sphere with diameter $\frac{d}{2}$, where d is the diameter of the the smallest enclosing sphere of t. We call t *k-free* if $|\{t_j \in A | d \leq d_j \text{ and } t_j \cap (t \oplus s) \neq \emptyset\}| < k$, where d_j is the diameter of the smallest enclosing sphere of triangle t_j and $t \oplus s$ is the Minkowski sum of s and t (Figure 2 shows a situation where a triangle (red) is 3-free).
Accordingly, we call the whole set A *k-free*, if all triangles $t_i \in A$ are *k-free*.

In other words, a triangle t is k-free if there are fewer than k triangles of A that both (1) have a larger minimum enclosing sphere and (2) intersect the volume that results from sweeping a sphere with diameter $d/2$ around t. For sake of simplicity, we will call these triangles *larger* triangles. More precisely, let t_i and t_j be two triangles with minimum enclosing spheres s_i and s_j. Let d_i be the diameter of s_i and d_j be the diameter of s_j. Then we say t_i is *larger* than t_j if $d_i \geq d_j$.

Please note that if we want to determine the concrete number of intersections for a concrete triangle mesh we are usually interested in a *minimal* constant k. However, for the theoretical considerations in this section, it is sufficient to have *any* constant k. It improves readability if we can simply use the same constant k for the individual triangles and the complete triangle set.

The following lemma shows that our definition guarantees that a single triangle cannot intersect too many larger triangles of a k-free triangle set:

Lemma 3.1 Let A be a k-free set of triangles and let $t \notin A$ be an arbitrary triangle. Then, t intersects at most a constant number of larger triangles $t_j \in A$. More precisely, the number of intersections between t and larger triangles $t_j \in A$ is at most $3k$.

The proof of this lemma relies on the following simple geometric observation:

Lemma 3.2 Let t be a triangle in 2D and c its minimum enclosing circle with diameter d. Then we need at most three circles of diameter $\frac{d}{2}$ to completely cover t.

Proof We consider two cases, acute and obtuse triangles. In the case of acute triangles, we can use the Circumcenter Theorem, which gives the circumcircle as the smallest enclosing circle. With the perpendicular bisectors, we can partition the triangle into six *right* triangles, all meeting at the circumcenter (see Figure 2 middle). The length of the hypotenuses of all the sub-triangles is obviously $\frac{d}{2}$. On each of the three hypotenuses, we place a circle of diameter $\frac{d}{2}$ on the midpoints. Due to Thales' Theorem, the opposite vertices will lie on these circles. Thus, all six sub-triangles are completely covered by those three circles.
In the case of obtuse triangles, the minimum enclosing circle c is not the circumcircle. Instead, its diameter is the length of the longest side, and its center lies on the midpoint of the longest side. Without loss of generality, it is sufficient to consider right triangles. If t is obtuse, we simply enclose it by a larger triangle t' by moving its vertex opposite the longest edge outwards onto the minimum enclosing circle. In the following, let t be a right triangle. Similar to the case above, we construct a partitioning of t by the perpendicular bisectors and the circumcenter. In this case, there will be four right sub-triangles. Again, the hypotenuses of all these sub-triangles have length $\frac{d}{2}$. Hence, we obtain a circle covering by placing circles of diameter $\frac{d}{2}$ on the midpoint of h and at the positions $\frac{1}{4}$ and $\frac{3}{4}$ of the length of the hypotenuse of t. □

Now that we have proven Lemma 3.2, we can use it to prove Lemma 3.1:

Proof Let c be the minimum enclosing circle of t in its supporting plane; let d be its diameter. We construct a circle covering of t according to Lemma 3.2. We claim that for each of these circle c_j of diameter $\frac{d}{2}$ there can be at most k larger triangles $t_i \in A$ that intersect it.
We use a proof by contradiction: assume that circle c_1 is intersected by $k+1$ larger triangles. Let t_a be the smallest of these triangles, and let d_a be the diameter of its minimum enclosing sphere. Then, by definition, $d \leq d_a$, because t_a is larger. Since t_a intersects c_1 and the diameter of c_1 is $\frac{d}{2}$, c_1 is completely located inside $t_a \oplus s_a$. Hence, there have to be k larger triangles in $t_a \oplus s_a$. This is a contradiction to t_a being k-free by prerequisite of Lemma 3.1. □

Note, if a set of triangles is k-free for a sphere of diameter $\frac{d}{2}$, it is also k-free for smaller spheres in the range of $0 < \delta \leq \frac{d}{2}$ which could result in smaller factors k'. Nevertheless, the choice of $\frac{d}{2}$ is not arbitrary. Actually, smaller spheres would require more circles for the circle covering in the proof of Lemma 3.2. The best choice between the sphere diameter and the number of circles in the covering remains an open question. This is mainly because, to our knowledge, the general sphere covering problem for triangles is still open.

However, the result of Lemma 3.2 allows us to prove a linear bound on the number of intersecting triangles with a decent constant factor for all k-free triangulated objects:

Theorem 3.3 Let A and B be two k-free sets of triangles. Then the total number of colliding triangles of A and B is in $\mathcal{O}(n)$, where n is the number of triangles in A and B. More precisely, the number of intersections is at most $3nk$.

Proof We test each triangle of A against all larger triangles of B and vice versa. For each of these tests we get at most $3k$ intersections according to Lemma 3.1. Moreover, we find all pairs of colliding triangles, because either of the triangles in a pair of intersecting triangles must be larger than the other. Overall, we get at most $3nk$ intersections. □

3.2. Extension to Arbitrary Polygons

Similarly to Definition 3.1, we can define k-freeness for arbitrary polygons:

Definition 3.2 Let $p \in A$ be a polygon in a polygon set A and $k > 0$ some constant. Let s be a sphere with diameter $\frac{d}{2}$, where d is the diameter of the the smallest enclosing sphere of p. We call p *k-free* if $|\{p_j \in A | d \leq d_j \text{ and } p_j \cap (p \oplus s) \neq \varnothing\}| < k$, where d_j is the diameter of the smallest enclosing sphere of polygon p_j and $t \oplus s$ is the Minkowski sum of s and t.
Accordingly, we call the whole set of polygon A *k-free*, if all polygons $p_i \in A$ are *k-free*.

Unfortunately, Lemma 3.2 does not necessarily hold for arbitrary polygons. Obviously, we could simply use a triangulation of an arbitrary polygon and apply Lemma 3.2 to each of the triangles in the triangulation. However, in this case, the constant would depend on the triangulation and thus, the number of edges of the polygons.

In order to get a factor that is independent of the actual polygon we use another approach. The main idea of Lemma 3.2 is to cover a triangle with circles of half of the diameter of the circumcircle of the triangle. Instead, we could also cover an upper bound of the triangle, this would still result in a linear number of intersections. In case of polygons we simply chose to cover the complete circumcircle of the polygon as an upper bound. We can cover it with seven spheres that have half of its' diameter (see Figure 2). This is a well known circle covering theorem [Ker39]. This results in an only slightly worse constant factor for arbitrary polygon soups, but the maximum number of intersections remains linear. The rest of the proof of the following theorem remains exactly the same as for Theorem 3.3.

Theorem 3.4 Let A and B be two k-free sets, each consisting of n polygons. Then the total number of colliding polygons of A and B is in $\mathcal{O}(n)$. More precisely, the number of intersections is at most $7nk$.

4. Our Algorithm

The proofs of Theorem 3.3 and Theorem 3.4, respectively, lead to an algorithm based on the following idea: when we want to check two objects A and B for collisions, we simply have to check each polygon of A against *larger* polygons of B and vice versa. Obviously, a naive implementation would still result in a quadratic running time. The challenge is to reduce the number of potentially colliding triangles that we have to check to a constant number for each polygon. In other words, we have to identify polygons in a

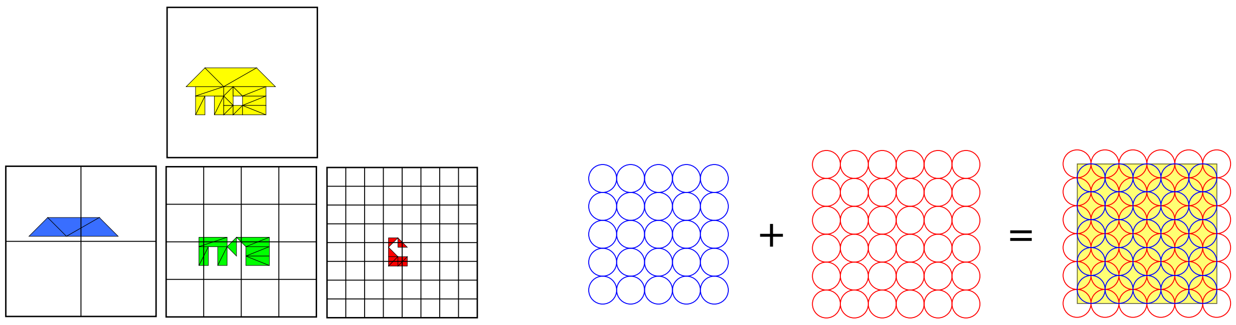


Figure 3: *Left: three consecutive levels of the hierarchical grid with distributed polygons based on their circumcircle. Right: covering of a cell in the uniform grid with two shifted regular circle lattices.*

certain neighborhood and we have to show that there are not too many polygons in this neighborhood.

A widely used approach for neighbor searching are uniform grids. However, choosing an appropriate cell size is challenging: if the cell size is too large, there may be many polygons assigned to the same cell. On the other hand, if the cell size is chosen too small, large polygons occupy a large number of cells. Fortunately, in our case, we are not interested in finding *all* neighbors, but we have to find only *larger* polygons. This means, we can use a hierarchy of grids with different cell sizes and assign each polygon to a certain level in this hierarchy where it does not occupy too many cells.

4.1. Populating the Hierarchical Grid

Before performing a collision query, we have to assign the polygons to their particular grid cells. To do that, we apply a simple rule: let A be a set of polygons. For each polygon $p_i \in A$ let d_i be the diameter of the circumcircle and let $d_{min} = \min\{d_i\}$. We set the cell size of the finest grid in our hierarchy to d_{min}. Coarser levels are derived by successively doubling the cell size. The hierarchy level l_i of each polygon p_i can be computed by

$$2^{l_i} \cdot d_{min} \leq d_i < 2^{l_{i+1}} \cdot d_{min}.$$

In other words, each polygon is assigned to the level so that the cell size is at most the diameter of the circumcircle. Then we simply add the polygon to all cells in the level l_i that are intersected by the polygon (see Fig 3).

4.2. Collision Queries

If we want to check two objects A and B for collision, we simply test all polygons $p_i^A \in A$ against all *larger* polygons $p_j^B \in B$ and vice versa. In detail, for each $p_i^A \in A$ we compute its level l_i and all cells in B's hierarchical grid that are intersected by p_i at this level. For each of these cells we test all included polygons p_j^B with at least the size of the circumcircle, i.e. $c_i^A \leq c_j^B$. In order to check also larger triangles, we ascend in the hierarchy until we reach the maximum level and again, test p_i^A against all included polygons of B for an intersection (see Algorithm 1 and 2).

This algorithm guarantees that we find for each polygon $p_i \in A$ all intersecting polygons $p_j \in B$ with at least the same diameter of the circumcircle.

Overall, we will find *all* colliding pairs of polygons if we test A against B and vice versa, because either of the polygons has a larger circumcircle, assuming general position of the polygons. Obviously, for real-world tests we cannot assume general positions. Here we avoid to double check polygon pairs by simply testing only strictly larger polygons in one direction.

Algorithm 1: checkCollisions(object A, object B)

```
forall the polygons $p_i \in A$ do
    checkCollisions( $p_i$, $B$)
forall the polygons $p_j \in B$ do
    checkCollisions( $p_j$, $A$)
```

Algorithm 2: checkCollisions(polygon p_i, object B)

```
Get hierarchy level $l_i$ for $p_i$
forall the hierarchy levels: $l_i \cdots l_{max}$ do
    forall the cells $c_k \cap t_i \neq \varnothing$ do
        forall the polygons $p_j \in c_k$ do
            polygonIntersection($p_i$, $p_j$)
```

4.3. Parallelization

This algorithm can be easily parallelized. For the population of the hierarchical grid, we assign all polygons independently to their particular cells. Simple atomic operations avoid race conditions if two polygons are assigned to the same cell. During the queries, we can also check all polygons for each object in parallel. Algorithm 3 shows the complete parallel algorithm. It uses Algorithm 2 that will be executed as the kernel for the collision check per polygon.

In case of rigid objects, the assignment to the grid cells does not have to be computed before each collision check, but it can be done once at the beginning of the simulation as a pre-processing step. However, even if the assignment is computed before each collision check, as it would be required for deformable objects, it does not affect the theoretical complexity of our algorithm.

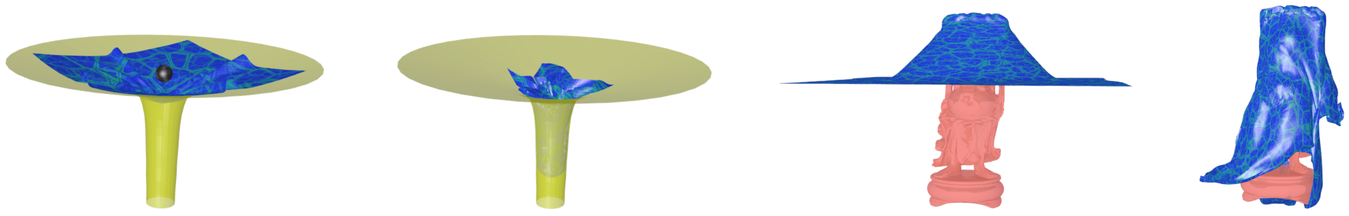


Figure 4: *Scenes from the benchmarks we used. Left: funnel benchmark, a ball presses a cloth into a funnel. Right: Buddha cloth benchmark, a cloth wraps around a Buddha statue.*

Algorithm 3: checkCollisionsParallel(object A, object B)

```
In Parallel forall the polygons $p_i \in A$ do
    assignPolygonToGridcell( $p_i$ )
In Parallel forall the polygons $p_j \in B$ do
    checkCollisions( $p_j, A$)
In Parallel forall the polygons $p_j \in B$ do
    assignPolygonToGridcell( $p_j$ )
In Parallel forall the polygons $p_i \in A$ do
    checkCollisions( $p_i, B$)
```

4.4. Analysis

The construction of our hierarchical grid can be done in linear time in case of sequential processing and constant time in the parallel case: computing the level of each triangle takes $\mathcal{O}(1)$ time. Due to the choice of the level – the cell size is at least the diameter of the circumcircle of each polygon – the polygon can intersect only a constant number of cells on its level. Hence, it has to be inserted into at most a constant number of cells. More precisely, each polygon can intersect at most eight cells in its level. Overall, we get a linear time for the construction of the hierarchical grid.

The query time consists mainly of two factors: the height of the hierarchy and the maximum number of polygons in a cell. First, we consider the number of polygons per cell. Due to the construction of the hierarchy, the minimum diameter d of the circumcircle of any polygon inside a cell with length c is at most $\frac{c}{2}$. We can cover the complete cell with spheres of diameter $\frac{c}{4}$, for instance by overlaying two regular sphere packings (see Figure 3 for an 2D example). Obviously, the number of spheres is constant and independent of the particular cellsize, because the diameter of the spheres is a fraction of the length of the cell. This number can be improved by using a better sphere covering. If we have such a sphere covering and if the object is k-free, there can be at most $k+1$ polygons intersecting such a sphere of diameter $\frac{c}{4}$, following the same argumentation as in the proof of Lemma 3.1. Summarizing, we have a constant number of spheres that are required to cover a cell c and we have at most a constant number of polygons intersecting each of these spheres; Consequently, the total number of polygons inside a cell is constant. Please note that this holds only for k-free objects. For the Chazelle polyhedron we would get a linear number of intersecting polygons for a sphere.

It still remains to show that the height of the hierarchy is independent of the number of polygons. Actually, the height of the hierarchy depends only on the ratio between the largest and the smallest polygon of each individual object. Let $d_{min} = \min\{d_i\}$ and $d_{max} = \max\{d_i\}$ where $p_i \in A$ are the polygons of a set of polygons A and d_i is the diameter of the circumcircle of each polygon p_i. Then the height h of the hierarchy is $h = \log(\frac{d_{max}}{d_{min}})$. Obviously, h is independent of the *number* of polygons in A, it only depends on their size distribution.

To summarize: in case of k-free sets of polygons, we get at most a constant number of polygons in each cell of the hierarchy, each polygon intersects at most a constant number of cells and the number of levels in the hierarchy is independent of the number of polygons. Overall, we get a running time of $\mathcal{O}(\log(\frac{d_{max}}{d_{min}})n)$ for a collision query which is almost linear in the number of polygons.

In the parallel case, we process all polygons at the same time for both the hierarchy construction and the collision queries. The construction requires an atomic operation when inserting several polygons into the same cell. However, the number of polygons per cell is constant, consequently, also the number of atomic operations is constant per cell. This means, we get a constant running time for the construction. In the query algorithm, all steps are constant per polygon except the height of the hierarchy. Hence, we get a parallel running time of $\mathcal{O}(\log(\frac{d_{max}}{d_{min}}))$ for the query which is independent of the polygon count and thus, almost constant. For both algorithms we need only a linear number of parallel processors.

The factor $\log(\frac{d_{max}}{d_{min}})$ somewhat blemishes the analysis and it is easy to construct artificial worst-case objects that would produce a linear height of the hierarchy. However, objects with such a wide spread in polygon sizes can be easily identified, in contrast to objects that produce a quadratic number of collisions, and moreover, they are typically avoided in real-world scenarios. In Section 6 we present the results of measuring a large object database. The typical height of the hierarchy is about eight for most objects.

In case of deformable (or fracturing) objects, $\frac{d_{max}}{d_{min}}$ could change due to the deformations. In our experiments (see Section 6) we did not observe such a behavior. We are positive that most deformation methods will not change $\frac{d_{max}}{d_{min}}$ much, because an extremely varying polygon size is usually unwanted, if only for reason of numerical stability and high-quality rendering. However, the theoretical proof of a constant $\frac{d_{max}}{d_{min}}$ for existing deformation schemes or the development of novel deformation schemes that keep this within certain bounds is an interesting question for future works.

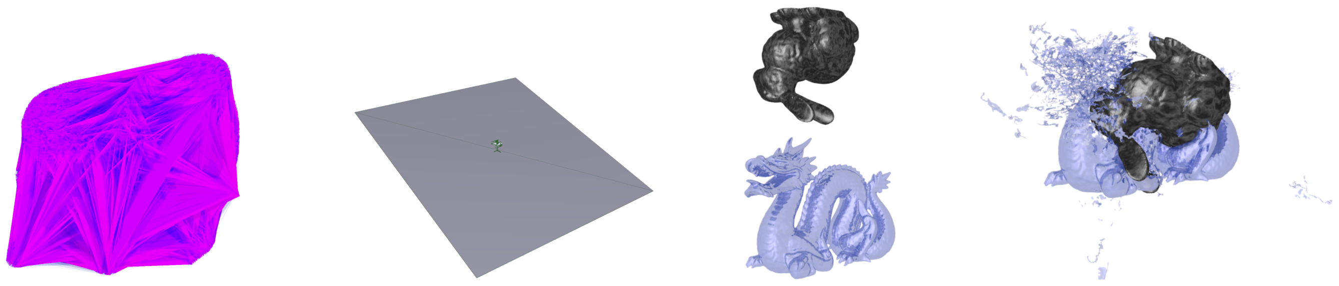


Figure 5: *Left: the worst object we found in the database has a constant k of more than 6000 according to Definition 3.1. Middle left: the object that realizes the maximum height of the hierarchy that we have found. It is a highly detailed object placed on a plane formed by two triangles. Middle right and right: scenes from the exploding dragon we used in one of our benchmarks.*

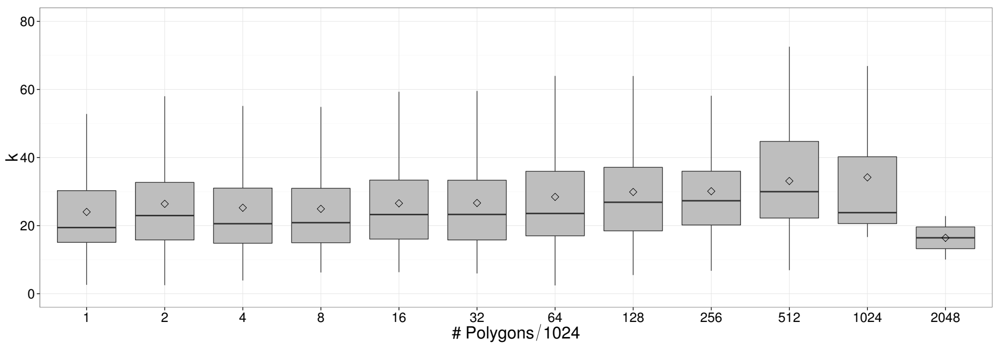


Figure 6: *Tukey boxplots of the factors k according to Definition 3.1 that we computed for 8000 objects from the 3D Meshes Research Database of the INRIA GAMMA group. We binned the objects with respect to the polygon count. The resolution was not equally distributed over all objects, hence, the number of objects with a high polygon count is relatively small. This results in larger standard deviations. Overall, we did not observe a relationship between the factor k and the number of polygons of the object.*

5. Implementation Details

The high-level description of our algorithm from the previous section is useful to understand the underlying concepts and for the theoretical analysis. An actual implementation should also consider details of current computer architectures like memory consumption or the memory access of current massively parallel processors like GPUs. In this section, we will describe our current implementation we used for our tests (see Section 6).

5.1. Spatial Hashing

A major drawback of the naive algorithm is the high memory consumption that is required to maintain a hierarchy of uniform grids. Usually, most of the cells remain empty, even if we restrict the grids' extents to the bounding boxes. In order to overcome this drawback, we use hash tables instead of real grids.

Hash tables are a widely used data structure that already have been applied successfully to represent uniform grids in the past [Tur89]. Hash tables achieve almost constant insertion and query time while reducing the memory overhead. The main challenge when using hash tables is to find an appropriate hash function. We assumed different hashing functions like DJB2 hashing [EL07], that spreads the triangles relatively uniformly in the hash table and, thus, minimizes hash collisions. Additionally, we tested 3D Morton codes [Mor66] that generate a neighborhood-preserving distribution of the triangles that should help to maximize coalesced memory access in our GPU implementation. In order to further improve the memory access, we initially pre-sort the triangles with respect to the Morton codes and the hierarchy levels using a bitonic sorter [Bat68]. However, we do this only once at the beginning of the simulation, not before each individual collision check. Consequently, this pre-processing heuristic does not affect the constant running time.

Another question that arises when using hash tables is the resolution of hash collisions that appear when several polygons have the same hash value. Closed hashing would result in extremely non-coalesced memory access. Open hashing, on the other hand, would require dynamical memory allocation if we would use lists for instance. Fortunately for our algorithm, we already know the maximum number of polygons per cell that we could use to pre-allocate memory for the hash buckets, at least as long as there is only one grid cell assigned to each hash value. However, this constant factor is only an upper bound that is rarely met in real applications and, hence, simply using such a large number of entries for each bucket would result in a large memory footprint of the hash table. To overcome these drawbacks, we decided to use a hybrid hashing

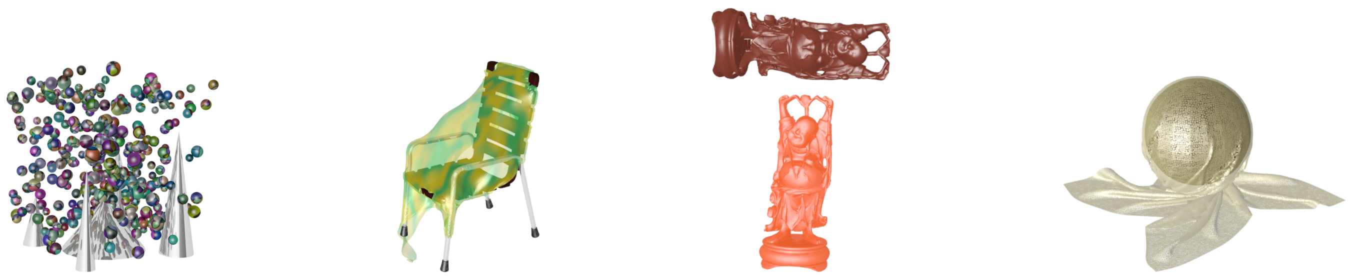


Figure 7: *Scenes from the benchmarks we used. Left: N-body benchmark, dozens of balls flying around. Middle left: chair benchmark, a cloth wraps around a chair. Middle right: breaking Buddha benchmark, a Buddha statue falls down and breaks into pieces when hitting another Buddha statue (see also Figure 1). Right: whirling cloth benchmark, a cloth is whirled by a ball (see also Figure 1)*

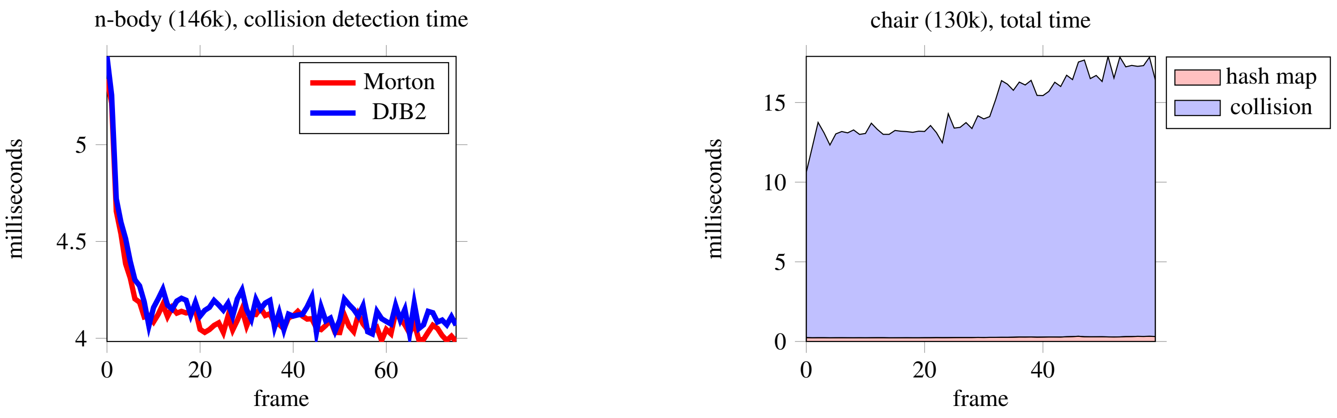


Figure 8: *Left: the collision detection times of kDet for different hashing functions, namely, Morton codes and DJB2 in the n-body scene that consists of 146k polygons. Right: stacked plot of our kDet's collision detection time (blue) and the hierarchical hash map's population time in the highest resolution chair scene (130k polygons).*

strategy: we reserve a certain, relatively small, number of entries for each bucket and in case of a bucket overflow, we simply search for an empty bucket and link it to the overflown bucket.

6. Results

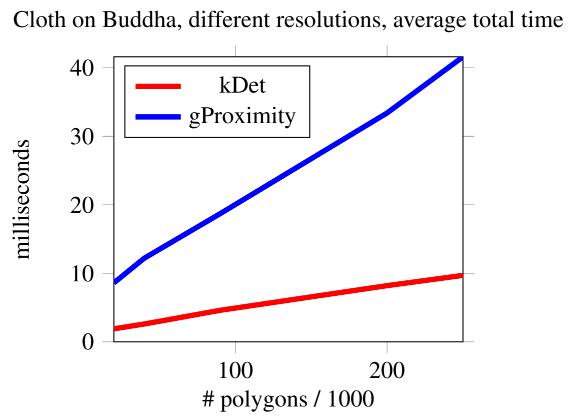
In this section we present results for both our theoretic considerations and the practical implementation of our new algorithm. More precisely, we checked how good our geometric predicate fits to real-world 3D objects, and we applied our algorithm to challenging scenarios including deformable object simulations and self-collision detection.

6.1. Analysis of Real-World Objects

First we tested, whether our geometric predicate is reasonable or not. This means, we checked whether typical real-world 3D objects are k-free and we derived their (minimum) constant factors k. We used objects from the 3D Meshes Research Database of the INRIA GAMMA group. This database covers a wide variety of different 3D objects with different polygon counts. Overall, we chose randomly 8000 objects from all categories (like cars, cloth, architectural models,...). The polygon count ranges from 10 to 2 millions (M=20657, SD=58168). For all objects we computed the minimum constant factor according to Definition 3.1. The average factor is 30.79 with a standard deviation of 32.82 (see Figure 6 for more details) but only a relatively small amount of this number (M=6.44, SD=6.14) results from topologically adjacent polygons. However, there are only very few objects with very high constant factors of up to 6000 that disturb the average (see Figure 5). Actually, a representative sample of these objects appears to be defective in our object viewer. Due to the large number of objects, it was impossible to filter out the defective ones, because this would have to be done by inspecting manually all objects.

In addition, we investigated the heights of the hash map hierarchies, which depends on the ratio between the largest and the smallest object of the scene. The average hierarchy height of all tested objects is 8.17 (SD=4.03). Again, we identified some objects with very large hierarchies. Most often, these are relatively small objects that are placed on large planes that are only modeled by two triangles (see Figure 5). Scenes like that hardly ever occur in real-time applications due to a number of detrimental effects. The total maximum height measured over all objects was 29 for a scene consisting of 40k polygons.

Overall, our geometric predicate defines factors that allow us



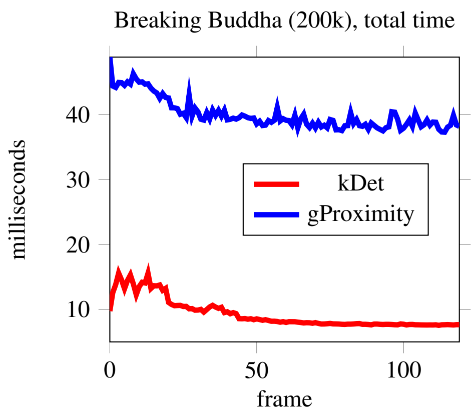


Figure 9: *Left: the average total time for kDet and gProximity, i.e. collision detection and hash map population or hierarchy refitting time, respectively in relation to the polygon count for the different resolutions in the cloth on Buddha scene. Right: the total time per simulation step for kDet and gProximity in the highest resolution breaking Buddha scene (250k polygons).*

to predict a much smaller number of possibly colliding polygons than the worst-case bound of n^2 for all real-world 3D objects in the database. Moreover, the height of the hash map hierarchy used by kDet is small compared to the number of polygons for all objects. The theoretical worst-case object with a linear height of the hierarchy does not seem to occur in practice.

6.2. Practical Results

We have implemented the parallel version of our algorithm in NVIDIA's CUDA. Our testing system consists of a 64 bit Windows 10 PC with a Nvidia GTX 1080 with 8 gigabytes of GDDR5X memory. The code was compiled with CUDA 8.0 for a device capability of 6.1.

We use different test scenes including classic benchmarks for deformable and fracturing objects like the funnel, the swirling cloth, the n-body and the exploding dragon benchmark from the UNC dynamics benchmarking suite [YCM07, PKS10] (see Figures 1, 5, 4 and 7). Additionally, we created own scenes in order to stress our algorithm and to test the performance with respect to the polygon count. Therefore, we built animations with different polygon resolutions for the chair, the cloth on Buddha and the breaking Buddha scene (see Figures 1, 5, 4 and 7). Since all our scenes contain deforming objects we inserted all polygons in the same hierarchical hash table. All our timings include all intra- and inter object collisions. We repeated each test run 10 times and always took the average from these runs in order to avoid the influence of system-specific caching effects. However, we did not observe significant differences between the test runs.

Table 1 summarizes all results from our timings. In addition to the timing for collision detection and the hash table population, we also included the average constant factors and standard deviations according to Definition 3.1 and the average heights of the hierarchies. First, we investigated the influence of the hashing function on the performance. We used DJB2 hashing and Morton codes as described in Section 5.1. For the DJB2 version we chose a size of 131101 elements as a large prime number and for the Morton codes a grid size of 64^3, which results in a table size of 64 and 128 MByte, respectively. Because of the more uniform spread of the polygons by DJB2, we decided to choose a smaller hash table. The results show that both methods perform equally well. Figure 8 shows the timings for the n-body simulation.

The insertion time of the polygons in the hierarchical grid does not differ significantly across all our scenes. Actually, it can be done very fast in less than two milliseconds in all our examples, including the breaking Buddha scene with more than 200k polygons. Compared to the collision detection time, the population time is negligible (see Figure 8 for an example in the highest resolution chair scene with 130k polygons). Our hash table build times are not only faster than typical refitting procedures for bounding volume hierarchies but they also avoid the typical deterioration: the quality of refitted bounding volume hierarchies usually decreases over time in case of heavy deformations, which usually results in decreased culling efficiency during the collision queries. kDet completely rebuilds the hash table hierarchy every time, hence, there is no quality loss. Moreover, kDet supports the removal or the addition of new polygons during the simulation.

We compared the performance of kDet to gProximity, a state-of-the art all-pairs collision detection method that also works completely on the GPU [LMM10] and whose sourcecode is available for download on the authors' website. gProximity relies on bounding volume hierarchies and it supports continuous as well as discrete collision detection and proximity queries. We used the OBB discrete version for all-pairs collision of deformable and fracturing objects. Like kDet, gProximity can also compute intra- as well as inter-object collisions.

We measured the smallest speed-up, 2.7, of kDet compared to gProximity in the chair scene. The reason for this is the geometry of the scene: the pipes forming the legs and the arms of the chair consist of long triangles in close proximity. In contrast, the triangles of the cloth are relatively small. This results in a large height of the hierarchy as well as a large factor k. However, even in

Scene	# polys	*k* (stdev)	height (stdev)	kDet col det	kDet update	kDet total time	gProx. col det	gProx. update	gProx. total time	speed-up
Dragon	252k	8.4 (0.9)	7.8 (0.0)	8.3	1.0	9.3	30.2	3.1	33.3	3.6
Whirling cloth	92k	6.6 (0.2)	3.4 (0.1)	5.2	0.2	6.1	21.9	2.1	24.0	3.9
N-body	146k	9.7 (0.5)	4.4 (0.0)	4.1	0.3	4.4	17.2	2.1	19.3	4.4
Funnel	18k	9.5 (2.0)	6.7 (1.0)	1.6	0.1	1.7	12.3	1.0	13.3	7.8
Chair	40k	19.0 (0.3)	10.7 (0.0)	4.6	0.1	4.7	11.8	1.0	12.8	2.7
	90k	20.6 (0.5)	12.3 (0.0)	10.0	0.2	10.2	23.1	1.4	24.5	2.4
	130k	21.3 (0.6)	11.8 (0.0)	14.3	0.3	14.6	32.1	2.1	34.2	2.3
Breaking Buddha	38k	16.8 (6.1)	8.3 (0.0)	4.9	0.2	5.1	18.6	0.9	19.7	3.8
	68k	16.8 (6.8)	8.3 (0.0)	6.2	0.2	6.3	27.1	1.4	28.5	4.5
	100k	13.3 (1.8)	8.3 (0.0)	7.6	0.3	7.9	38.0	2.1	40.1	5.1
	120k	13.3 (1.8)	8.3 (0.0)	11.8	0.4	12.2	59.2	2.5	61.7	5.1
Cloth on Buddha	20k	11.6 (2.0)	8.3 (0.0)	1.8	0.1	1.9	8.3	0.3	8.6	4.5
	40k	11.1 (2.0)	8.0 (0.0)	2.5	0.1	2.6	11.2	1.0	12.2	4.9
	90k	10.0 (1.4)	7.9 (0.0)	4.4	0.2	4.6	17.0	1.7	18.7	3.8
	200k	8.3 (0.9)	7.6 (0.0)	7.7	0.5	8.2	31.3	2.0	33.4	4.1
	250k	8.2 (0.6)	7.5 (0.0)	9.2	0.5	9.7	39.1	2.5	41.6	4.2

Table 1: *The results from our timings for the different benchmarks and the different algorithms. The table shows the polygon count of the scenes we used in our benchmarks, the average factors k according to Definition 3.1, including the standard deviation, the average height of our hierarchical hash map, and the timings for kDet and gProximity. The timings are divided into the population time of the hash map for our algorithm and the BVH refitting for gProximity, respectively (update), and the collision detection times for inter- and intra-object collisions (col det). Additionally, the total times for the update and the collision detection are shown (total time).*

this case kDet outperforms gProximity by at least a factor of two. The biggest challenges for BVH-based algorithms are scenes with heavy deformations or even changes in the topology (like the cloth on Buddha or the breaking Buddha scene). In these scenarios we achieved a speed-up factor of more than four for our algorithm.

Overall, the results show that kDet is able to achieve real-time performance, i.e. less than 15 msec, in all our test scenarios, even for large polygon counts of up to 250k polygons in the exploding dragon and the cloth on Buddha scene (see Table 1). These times include the population of the hierarchical grid, the computation of the collisions between different objects but also self-collisions. Moreover, our algorithm outperforms gProximity in all scenarios by at least a factor of two. However, not only the average time is much better but also the individual collision detection times per frame (see Figure 9 for results of the highest resolution breaking Buddha scene).

Although the theoretical running time of kDet is constant assuming an ideal PRAM model, it does vary in practice. This is mostly due to the limited number of processors of current GPUs. Consequently, our algorithm scales linearly with the number of polygons on real GPUs, as expected (see Figure 9 for results of different resolutions of the cloth on Buddha scene).

7. Conclusions and Future Work

We have defined a novel geometric predicate for arbitrary polygonal models that allows us to prove a worst-case linear number of intersecting polygon pairs for all objects that fulfill it. The predicate can be easily tested in advance and it is sufficiently general that almost all "normal" 3D object fulfill it. Our proof provides a theoretical basis for the common wisdom that "normal" shapes usually exhibit worst-case collision detection times linear in the number of triangles.

Additionally, we have presented a new algorithm, kDet, that is able to find all intersecting polygons in almost linear sequential time for objects that match our predicate. A parallel version can even achieve a constant worst-case running time. kDet is suitable for all kinds of polygon soups and it can be applied to deformable and even topology-changing objects at no extra costs, because no complicated pre-processing steps or acceleration data structures are necessary. Our results show that complex deformable objects consisting of hundreds of thousands of polygons can be checked in less than 15 msec including self-collision detection.

Our work presented here opens up a lot of interesting avenues for future work: a natural next step would be the development of an algorithm that optimizes the constant factor for real-world objects by improving the meshing. Obviously, also the height of the hash map hierarchy should be considered for this optimization. We believe, our predicate could be also used to improve existing approaches. For instance, it can lead to new construction methods for optimized bounding volume hierarchies. The development of simulation methods that maintain the predicate and the height of the hierarchy during deformations is another challenge. Finally, it would be interesting to apply our geometric predicate also to other problems in computer graphics, for instance to quality measurement of polygonal meshes or for object classification problems.

Acknowledgements

This work was partially funded by the German Federal Ministry of Economics and Technology (BMWi) under grant 16KN036252 and the "Creative Unit – Intra-Operative Information".

References

[AFC*10] Allard J., Faure F., Courtecuisse H., Falipou F., Duriez C., Kry P. G.: Volume contact constraints at arbitrary resolution. *ACM Transactions on Graphics 29*, 4 (July 2010), 82:1–82:10. 2

[Bat68] Batcher K. E.: Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, pp. 307–314. 7

[Ben75] Bentley J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept. 1975), 509–517. 2

[BT95] Bandi S., Thalmann D.: An adaptive spatial subdivision of the object space for fast collision detection of animated rigid bodies. *Computer Graphics Forum 14*, 3 (1995), 259–270. 2

[Cha84] Chazelle B.: Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm. *SIAM J. Comput. 13*, 3 (July 1984), 488–507. 1

[DK90] Dobkin D. P., Kirkpatrick D. G.: Determining the separation of preprocessed polyhedra: A unified approach. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming* (New York, NY, USA, 1990), Springer-Verlag New York, Inc., pp. 400–413. 2

[EL07] Eitz M., Lixu G.: Hierarchical spatial hashing for real-time collision detection. In *Shape Modeling and Applications, 2007. SMI '07. IEEE International Conference on* (June 2007), pp. 61–70. 2, 7

[GLM05] Govindaraju N., Lin M., Manocha D.: Quick-cullide: fast inter- and intra-object collision culling using graphics hardware. In *Virtual Reality, 2005. Proceedings. VR 2005. IEEE* (March 2005), pp. 59–66. 2

[GRLM03] Govindaraju N. K., Redon S., Lin M. C., Manocha D.: Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), HWWS '03, Eurographics Association, pp. 25–32. 2

[JP04] James D. L., Pai D. K.: Bd-tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 393–398. 2

[Ker39] Kershner R.: The number of circles covering a set. *American Journal of Mathematics 61*, 3 (1939), 665–671. 4

[KP03] Knott D., Pai D. K.: Cinder: Collision and interference detection in real-time using graphics hardware. In *Graphics Interface* (2003), pp. 73–80. 2

[LAM06] Larsson T., Akenine-Möller T.: A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics 30*, 3 (2006), 450 – 459. 2

[LC91] Lin M., Canny J.: A fast algorithm for incremental distance calculation. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on* (Apr 1991), pp. 1008–1014 vol.2. 2

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D. P., Manocha D.: Fast bvh construction on gpus. *Computer Graphics Forum 28*, 2 (2009), 375–384. 3

[LHLK10] Liu F., Harada T., Lee Y., Kim Y. J.: Real-time collision culling of a million bodies on graphics processing units. In *ACM SIGGRAPH Asia 2010 Papers* (New York, NY, USA, 2010), SIGGRAPH ASIA '10, ACM, pp. 154:1–154:8. 3

[LMM10] Lauterbach C., Mo Q., Manocha D.: gproximity: Hierarchical gpu-based operations for collision and distance queries. *Comput. Graph. Forum 29*, 2 (2010), 419–428. 3, 9

[Mor66] Morton G. M.: *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*. Tech. rep., IBM Ltd., Mar. 1 1966. 7

[MZ15] Mainzer D., Zachmann G.: *Collision Detection Based on Fuzzy Scene Subdivision*. Springer Singapore, Singapore, 2015, pp. 135–150. 3

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

[PKS10] Pabst S., Koch A., Strasser W.: Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. *Computer Graphics Forum 29*, 5 (2010), 1605–1612. 2, 9

[PY90] Paterson M., Yao F.: Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete & Computational Geometry 5*, 1 (1990), 485–503. 2

[ST95] Schömer E., Thiel C.: Efficient collision detection for moving polyhedra. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry* (New York, NY, USA, 1995), SCG '95, ACM, pp. 51–60. 2

[ST96] Schömer E., Thiel C.: Subquadratic algorithms for the general collision detection problem. In *12th European Workshop on Computational Geometry* (March 1996), pp. 95–101. 2

[TMLT11] Tang M., Manocha D., Lin J., Tong R.: Collision-streams: Fast GPU-based collision detection for deformable models. In *I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (2011), pp. 63–70. 3

[TTWM14] Tang M., Tong R., Wang Z., Manocha D.: Fast and exact continuous collision detection with bernstein sign classification. *ACM Transactions on Graphics 33*, 6 (Nov. 2014), 186:1–186:8. 3

[Tur89] Turk G.: *Interactive collision detection for molecular graphics*. Tech. rep., 1989. 7

[vdB98] van den Bergen G.: Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools 2*, 4 (Jan. 1998), 1–13. 2

[VSC01] Vassilev T., Spanlang B., Chrysanthou Y.: Fast cloth animation on walking avatars. *Computer Graphics Forum 20*, 3 (2001), 260–267. 2

[WBS07] Wald I., Boulos S., Shirley P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (Jan. 2007). 2

[WFZ13] Weller R., Frese U., Zachmann G.: Parallel collision detection in constant time. In *Virtual Reality Interactions and Physical Simulations (VRIPhys)* (Lille, France, Nov. 2013), Eurographics Association. 2

[WKZ06] Weller R., Klein J., Zachmann G.: A model for the expected running time of collision detection using AABB trees. In *Eurographics Symposium on Virtual Environments (EGVE)* (Lisbon, Portugal, 8–10 May 2006), Hubbold R., Lin M., (Eds.). 2

[WLZ14] Wong T. H., Leach G., Zambetta F.: An adaptive octree grid for gpu-based collision detection of deformable objects. *Vis. Comput. 30*, 6-8 (June 2014), 729–738. 3

[WTTM15] Wang Z., Tang M., Tong R., Manocha D.: Tightccd: Efficient and robust continuous collision detection using tight error bounds. *Computer Graphics Forum 34* (September 2015), 289–298. 3

[YCM07] Yoon S.-E., Curtis S., Manocha D.: Ray tracing dynamic scenes using selective restructuring. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2007), EGSR'07, Eurographics Association, pp. 73–84. 9