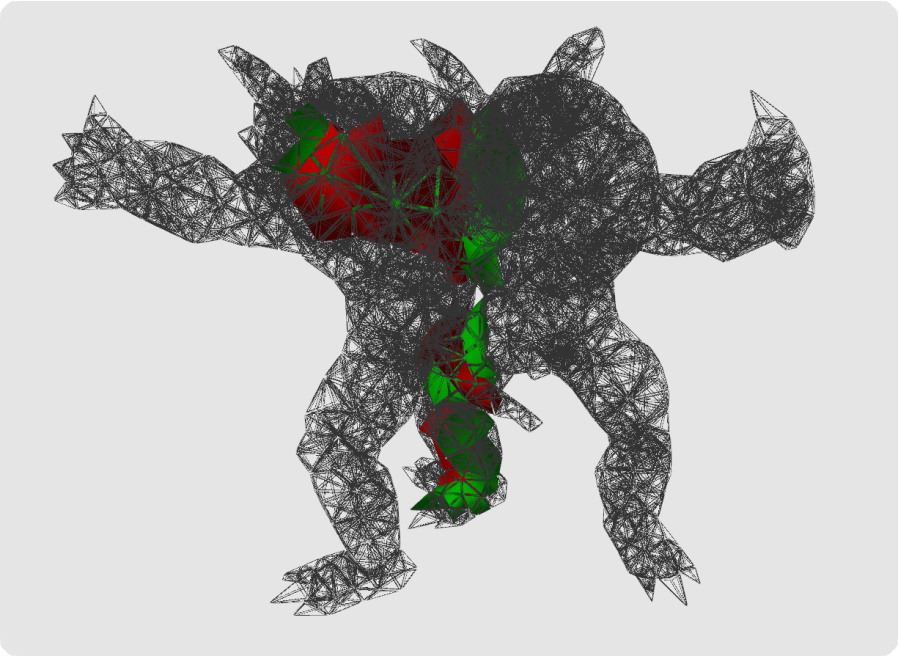
Universität Bremen

Faculty 3: Mathematics & Computer Science
Program: M.Sc. Computer Science

Master's Thesis

TETRAHEDRAL KDET

Linear Time Collision Detection for Tetrahedral Meshes



Submitted by:	Navid Mirzayousef Jadid
Enrollment ID No.:	4136247
E-Mail:	navid@uni-bremen.de

First Examiner:	Prof. Dr. Gabriel Zachmann
Second Examiner:	Dr. René Weller
Supervisor:	M.Sc. Hermann Meißenhelter
External Supervisor:	Prof. Yoshio Okamoto (UEC Tokyo)

July 25, 2023

DECLARATION

Hereby I declare that I am the sole author of this thesis. No sources or tools other than those listed were used. All references and citations to other works, be it direct or indirect, are declared as such.

Ich versichere, die Masterarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, July 25, 2023

Navid Mirzayousef Jadid

ACKNOWLEDGMENTS

I would like to first of all thank all my family and friends that offered me continuous support during my studies. Their encouragement and help consistently reached me at the right time.

Next, I would like to thank all the staff at UEC Tokyo who took great care of me during my stay abroad in 2019/2020. This especially applies to Prof. Choo who supervised the whole exchange program at UEC; Ms. Tomita who helped me in many situations and was always quick to reply to my requests; Prof. Watanabe who repeatedly moved us with his speeches; and all my Japanese language teachers, especially Ms. Nakashima who never failed to greet me with a smile and kept motivating me to do better. Furthermore, special gratitude is reserved for Prof. Okamoto who graciously welcomed me to his research lab and gave me great supervision for my work before and after my stay.

Last but not least, I want to thank my supervisors and staff at the University of Bremen: Dr. Weller, Hermann Meißenhelter, Prof. Dr. Zachmann, Sabine Dohls, and Ms. Kranz from the examination office. Ms. Kranz would swiftly answer all my inquiries without exception, especially during the last few months of my thesis. Sabine, the system administrator for the PC pool of our work group, helped me countless times with my work computer and was also fast to respond without fail. Most importantly however, were Dr. Weller and Hermann who always took time for me, even during my stay abroad, and offered me great advice during our weekly meetings here, while Prof. Zachmann had useful insights at every monthly work group meetings. It is difficult to express how thankful I am for their lasting support and patience throughout my graduate studies, and I am glad to have been able to contribute to the body of scientific knowledge under their guidance.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1 INTRODUCTION

1.1 MOTIVATION

With the steady rise of computational power and efficiency in hardware during the previous two decades, real-time collision detection of tetrahedral and polyhedral models has become much more feasible. Collision detection itself is a technology widely used in all kinds of applications, such as computer games, material simulations, etc.; yet, currently triangulated models are used most commonly. The use of tetrahedral meshes then allows for proper simulation of deformable or fracturing objects, and generally objects with an internal structure.

While for many applications collision detection can be sufficiently accelerated with data-structures like, for example, Bounding Volume Hierarchies or alike, these methods still inhibit the risk of needing $\mathcal{O}(n^2)$ time in the worst case. For time-critical applications or use-cases where a steady runtime needs to be guaranteed, other solutions have to be utilized.

In 2017, Weller, Debowski, and Zachmann [WDZ17] offered the foundational basis of this work when they proposed kDet, an efficient collision detection algorithm for the graphics-processing-unit (GPU) based on a novel geometric predicate. They proved a worst-case linear runtime $\mathcal{O}(n)$ bound to the number of intersecting pairs of polygons. When parallelized with a linear number of processors, the algorithm can theoretically perform in constant time. In the most basic sense, their method is based on the premise that for "normal" objects the number of neighbors for a given polygon is somewhat limited. Therefore, only a certain number of potential collision partners in near proximity need to be considered for each polygon.

However, this was only the first step. Since the algorithm does not require a specific kind of base primitive, it should be possible to apply these findings to other classes of objects like higher dimensional polytopes. In this work, I want to present how kDet can be applied to tetrahedral and polyhedral meshes. Fast collision detection for (deformable) tetrahedral meshes finds use in various field, such as material simulations with the finite-element-method (FEM) or real-time computations for robot vision and haptics, to only name a few.

1.2 CHALLENGES & GOALS

One immediate point of concern is the increased number of primitives necessary for volumetric aka tetrahedral meshes. This is especially problematic when considering collisions between primitives of the same object, so-called self- or inter-object collisions. This drastically multiplies the number of potential collision partners that need be considered for each primitive. Yet, detecting self-collisions is a necessity and cannot be avoided for proper simulation of deformable bodies [WC21].

Furthermore, intersection tests between tetrahedra are also more computationally expensive than for triangles. Obtaining collision responses also requires additional computations. Whether or how much this would affect the overall runtime was not clear at the beginning of my work.

Lastly, it is important that capabilities of the GPU be properly utilized, while avoiding the many pitfalls that can occur when programming on it.

In this work, I want to first extend the geometric predicate of kDet to the case of tetrahedra and general polyhedra. Once this foundation is set, the previous implementation of kDet shall be adjusted and improved upon, before being put to rigorous test in different scenarios. With that we will hopefully explore the strengths and weaknesses of the algorithm. This will also reveal us, whether the promise of linear runtime holds true in practice for the case of collision detection with tetrahedral meshes. Even if that is the case, the actual runtimes need to be evaluated to see whether kDet would be usable in time-critical real-time applications.

2 RELATED WORK

2.1 COLLISION DETECTION FOR DEFORMABLE OBJECTS WITH TETRAHEDRAL MESHES

To date, the topic of collision detection (CD) with tetrahedral models is not as extensively researched as collision detection on polygonal and triangulated bodies. Only in within the last two decades have computational capabilities, such as the emergence of programmable GPUs, allowed their use in real-time applications. Prior, only meshes with a limited resolution could be used.

Early, Teschner et al. [Tes+03] proposed the use of spatially hashed grids for CD with tetrahedra. Spatially hashed grids are data structures that model a grid, but do not fully instantiate it. Instead the grid is implicitly created by hashing occupied grid cells (and their contents) into a hash map based on their positions, as demonstrated in figure 2.1. They optimized parameters for the method using a uniform grid, with which they could overall achieve a linear running time complexity, only depending on the number of primitives. They could simulate 20k tetrahedra at around 15Hz. Eitz and Lixu [EL07] followed suit, but used a hierarchical spatial grid instead, which is able to adapt to a given scene. They put emphasis on minimizing computation times for the acceleration data structure.

Marchal, Aubert, and Chaillou [MAC04] showed an alternative approach without the use of a computational grid, instead utilizing updates of distance fields computed with fast marching [Set96] on the tetrahedral models.

As for deformable objects, Teschner et al. [Tes+05] classified applicable CD methods and gave a comprehensive review on them. They focused mainly on inter-object collisions; self-collisions were only briefly discussed. Weller [Wel13] later gave a compact summary on the same topic. More recently, Wang and Cao [WC21] did a review on state-of-the-art collision detection methods for deformable object, describing the various approaches in detail and emphasizing the importance of self-collisions in the matter. We shall discuss the most relevant of the methods mentioned in these reviews. Please note however, that most of them were not yet applied to CD with tetrahedral meshes.

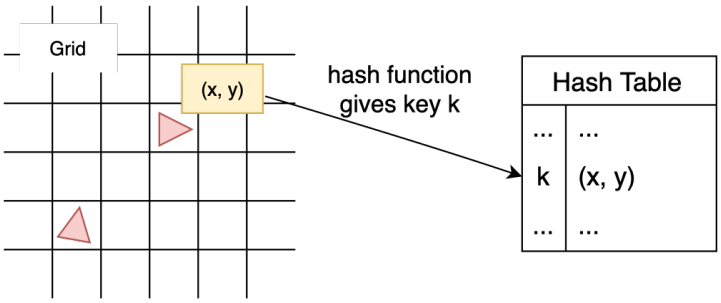


Figure 2.1: A spatial hashing grid is always only implicitly constructed, by calculating grid positions of objects, passing them to a hash function to obtain a key, before inserting them into a hash table.

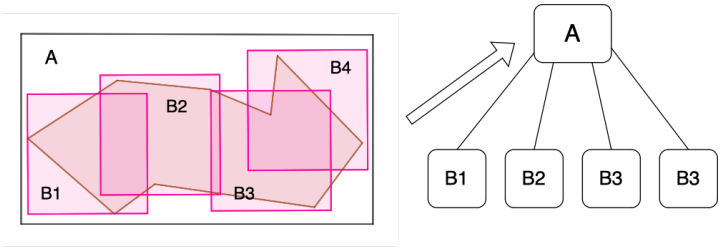


Figure 2.2: A Bounding Volume Hierarchy encloses an object with multiple sub-structures, which are then put into a hierarchy. For collision detection, the hierarchy is traversed until either no overlap of sub-structures is found or until primitives are to be tested for intersection.

Bounding Volume Hierarchies (BVH) have been used to accelerate CD for a long time [Ber97; GLM96] They divide the object into recursive bounded areas which are stored in a tree structure or hierarchy (see figure 2.2). For CD, the hierarchy is traversed until either primitive intersection tests have to be performed or no collisions of bounding volumes are found. When using BVHs on the GPU they need suitable adjustments, as the hardware does not work well with recursion. Lauterbach, Mo, and Manocha [LMM10] proposed the use linear ordering derived from Morton codes for efficient construction of BVHs for CD on the GPU. This was later improved upon by Wang et al. [Wan+18], addressing issues with culling efficiency and caching.

Spatial partitioning splits the space into sub-domains and checks for collision of objects within the same region. There exist many schemes for spatial partitioning, such as the previously mentioned spatially hashed grid [Tes+03; EL07]. This work's predecessor paper also uses methods of spatially hashed grids and was shown to also be suitable for changing topologies [WDZ17]. Meanwhile, Ye et al. [Ye+16] used similar techniques for robust and efficient CD in surgical training simulators. Other methods of spatial partitioning include: uniform grids [GASF94], octrees [WLZ14], and k-d trees [HKM95].

Sweep and prune algorithms efficiently check AABBs of shapes, before deciding to move to further computations such as intersection

tests. The method relies on the fact that AABBs only overlap in three dimensions, if their projection intervals along each of the three world axes also have overlap. For that, the projected intervals are sorted before sweeping through them. Nowadays, principal component analysis is used to determine the optimal sweeping direction. Mainzer and Zachmann [MZ15] used fuzzy clustering to subdivide objects for fast CD of deformable objects on the GPU. Capannini and Larsson [CL16; CL18] combined the sweep and prune method with BVHs and achieved great culling performance. Their algorithm runs only on the CPU as it is heavily reliant on fast and large caches.

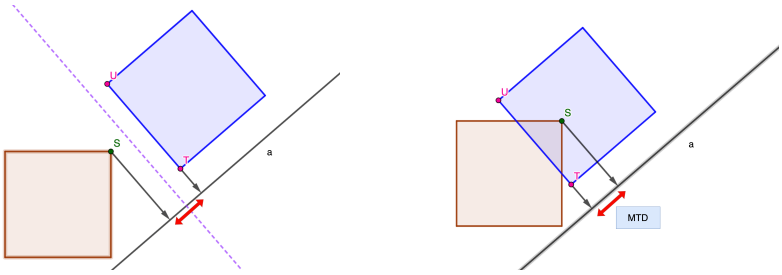
2.1.1 *Self-Collisions*

The problem with self-collisions detection is that even in cases where the mesh remains free from self intersections, the amount of computational effort required per frame is huge. This stems from the fact that each primitive in a volume mesh is close to several others and thus necessarily has several potential collision pairs every frame. Acceleration techniques like BVH, which are used effectively in inter-object collisions, often do not apply well to self-collisions or need suitable adjustments [WC21].

To reduce the number of potential collision pairs per primitive in self-collision, several approaches have been proposed. The normal cone method proposed by [VT94] is one such technique, and utilizes the mesh connectivity and topology for culling irrelevant collision partners by means of normal cones and two-dimensional contour test. Normal cone methods were commonly used with discrete self-collision detection, but Tang et al. [Tan+09] it to continuous CD for better performance. However, the additional cost of culling with normal cones scaled badly with the quadratic complexity of CCD. Wang et al. [Wan+17] later published a variation that combined BVH with the normal cone method and could be performed discretely as well as continuously. The algorithm has a linear time complexity and reliably found all self-collisions for models with triangle counts in the six digits. A year later, they improved upon the robustness and runtime of the approach [Wan+18]. Tang et al. [Tan+18b] used the normal cone method together with spatial hashing and achieved 6-8x speed compared to their old work [Tan+14].

Others utilized a fast triangle-triangle intersection test [Mö97] together with the computing capabilities of the GPU and spatial partioning [PKS10] or BVHs [Tan+18a] to great effect. Notably, this approach was also applied to triangle models undergoing topological changes [He+15].

Lastly, Tian, Hu, and Shen [THS19] presented a hybrid CPU-GPU continuous CD method for studying brain deformations with models



(a) (b)

Figure 2.3: The SAT test looks for different penetration axes. (a)) If we find a hyper-plane that separates the two objects, then they are not colliding. (b) If there is overlap, then by projecting the furthest points of each object onto the axis, we can obtain the MTV/MTD.

that had tetrahedral meshes. They achieved very interactive frame rates while also computing the deformation via finite element method at the same time.

2.2 INTERSECTION TESTS

Intersection tests are algorithms that determine whether two shapes, usually polygons or polyhedra, have any overlap. They are usually executed during the narrow-phase portion of a collision pipeline, to see which primitives of a given mesh intersect. While many tests only return a boolean answer, some will additionally calculate a collision response.

One such response is the minimum translation vector (MTV) or distance (MTD). The MTV is the smallest translation vector which when applied would separate two penetrating objects. A problem with the MTV is that it can produce sub-optimal results, especially with discrete CD methods, as shown in figure. When objects move too fast and temporal coherence is not considered, applying the MTV can place the objects at undesirable locations. Therefore, it is more frequently used with continuous CD methods and additional constraints.

For polyhedra, the two most commonly used intersection tests are the separating axis test (SAT) [Got96; GLM96] and the GJK algorithm [GJK88]. In the following, we will look at both, examine their differences, as well as merits and demerits.

2.2.1 *Separating Axis Test (SAT)*

The separating axis theorem (also known as hyperplane separation theorem) states that for two convex shapes that have no overlap, there exists at least one axis where the projection of the shapes onto that axis are disjoint, as shown in figure 6.7.

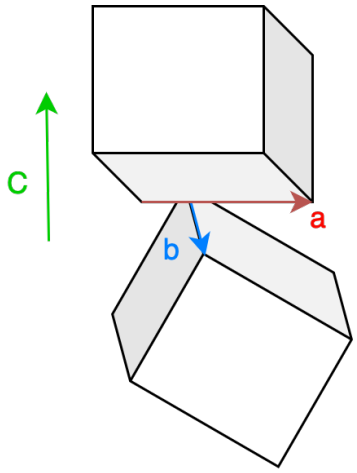


Figure 2.4: A case where the edge-on-edge test provides the separation axis. We obtain it by calculating the cross product c of a and b.

The SAT intersection test tries to find such an axis, but if it fails, we know that the two shapes must intersect. The important question is which axes should be tested. These are 1) the face normals of each shape and 2) in three dimensions also the cross products of the edges between the objects [GLM96]. These are commonly referred to as face- and edge-queries. Figure 2.4 shows a case where the edge-on-edge case is relevant. For tetrahedra, this gives a total of 44 axes to check: 8 for the face normals (4 per tetrahedron), and 36 for the edge-on-edge axes (6×6, with 6 edges for each tetrahedron).

In addition, the regular SAT test is capable of computing the MTV. For that, whenever a separation axis is tested, the shapes are projected onto it. By determining the overlap of the projections, we obtain the penetration along the tested axis, as shown in figure 2.3b. Since the SAT test only returns intersection after all relevant axes have been tested, we obtain the MTV by keeping track of the separation axis with the shallowest penetration and the depth.

In practice, we first search for so-called support points obtained from a support mapping, which is a function that takes a shape and a direction and then returns the point furthest in that direction. When probing for support points along an separation axis, each polyhedron searches in the opposite direction of the other. To obtain the penetration depth, the support points are projected onto the axis and their distance is measured (see figure 2.3b).

Improvements of the SAT algorithm mainly focus on reducing the number of edge-queries. The reasons are twofold: For one, their number scales quadratically with the edge-count per simplex. Second, they require more expensive computations, like the calculation of cross-products.

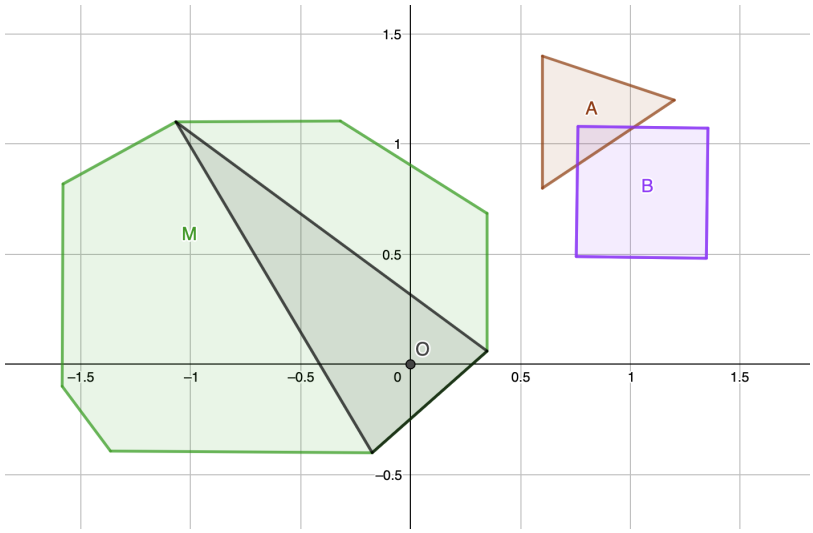


Figure 2.5: If two primitives are colliding, then their Minkowski difference (green) will contain the origin. In practical applications, only parts of the Minkowski difference are constructed (black) to check whether they contain the origin. This saves computation time.)

Following this principle, Ganovelli, Ponchio, and Rocchini [GPR02] presented a faster version of the SAT for tetrahedron-tetrahedron intersections that foregoes explicitly testing pairs of edges, by reusing quantities that were obtained during the face-queries. Another method makes clever use of concepts like Gauss maps to also trim down on edge-queries. Moreover, this approach generalizes to polytopes, and generates the contact manifold as well [Mig10].

One major downside of SAT is that it only works with convex shapes and also requires their features (edges, faces, etc.). Another point of consideration is that when searching for the MTV, SAT must check all separation axes, which is inherently slower than comparable methods that just find approximate solutions and can therefore finish earlier. On the other hand, the SAT test is relatively easy to understand and implement, especially when comparing to algorithms such as GJK. It also does not need elaborate data-structures, such as priority queues, thus on GPU's, where branching is rather slow, it might perform faster. [Mig10].

2.2.2 *Gilbert-Johnson-Keerthi (GJK)*

The Gilbert-Johnson-Keerthi (GJK) algorithm [GJK88] is frequently used for the distance computations between shapes, but can also be used for intersection queries. It takes advantage of the observation that if the Minkowski-difference of two polytopes contains the origin, then the two polytopes must intersect, as shown in figure 2.5. The Minkowski-difference can be thought of as the shape that results when

we subtract the positions of all vertices of one polytope from another. In practice, the computation of the full convex hull of the Minkowski-difference is too expensive for real-time applications; instead, the GJK algorithm iteratively constructs simplices which are contained within said hull. If the origin is found inside one of the constructed sub-shapes, then it has to also be inside the Minkowski-difference (see again figure 2.5).

GJK by itself does not compute the MTV; however, extensions of it with such capabilities have been presented, such as the popular Expanding-Polytope-Algorithm (EPA) [VDB01; VDB03]. It achieves this by expanding the simplex that was used to find the origin inside the Minkowski-difference and determining the closest feature between them.

Another variation of GJK can even compute the domain intersection of several polytopes. This is done by translating the origin back into the intersection area and constructing the convex hull of the intersection volume by solving a dual problem [TCF13]. Computing the domain intersection from a point known to be inside it had already been shown much earlier for other intersection methods [MP78; DK83].

The major advantage of the GJK algorithm is that it can handle various geometries and is not concerned with the format the geometry is stored in, so long a suitable support mapping is provided.

However, GJK is notoriously sensitive to floating point precision problems. Early on, Bergen [Ber99] presented a faster, more versatile and robust version. Montanari, Petrinic, and Barbieri [MPB17] replaced the original Johnson algorithm and Backup procedure with a distance sub-algorithm, gaining more numerical robustness. Their method is less susceptible to degenerate geometries while having a faster convergence rate and performing around 15% to 30% better than the original GJK algorithm. More recently, Montaut et al. [Mon+22] applied principles from other convex optimization algorithms and achieved up to two times faster computation times. A last noteworthy mention is XenoCollide, a "cousin" of GJK, that also claims to be more robust and require fewer branching operations overall [Sne08].

3

THEORETICAL WORK

This part of my research was conducted during my stay abroad at UEC-Tokyo[1], Japan, under the attentive supervision of Prof. Okamoto.

3.1 APPLYING THE PREDICATE OF K-FREENESS TO SETS OF TETRAHEDRA & GENERAL POLYHEDRA

The predicate of *k-freeness* was first proposed by Weller, Debowski, and Zachmann [WDZ17] with the original work behind kDet. At the heart of the predicate lies the observation that for three-dimensonal models meant to represent real-world objects, the number of polyhedra intersecting the neighborhood of another polyhedron has to be somewhat limited. Technically and mathematically speaking, it would obviously be possible to cram infinitely many objects in a space, by, for example, making them infinitesimally thin. But for our mentioned use-case this not should apply. Thus, we will see that for two sets where each object fulfills the mentioned neighborhood property, the number of possible intersecting pairs of polyhedra stays linear to the number of total polyhedra of both sets.

To demonstrate that, I will extend the existing predicate to the case of polyhedra by first adjusting the original definitions and lemmas. Then, I will follow the same exact proof for the lemmas as in the triangle/polygon case, to show that it seamlessly extends to polyhedra. The upper bound for the maximum of intersections per primitive will differ however. This property is dependent on solving a case of the sphere covering problem for tetra- and polyhedra respectively. The topic will be dealt with separately, after the general proof for the predicate has been presented.

Definition 3.1.1. Given some constant $k > 0$, a polyhedron $p \in P$, with P being a set of polyhedra, let d be the diameter of the smallest enclosing sphere of p and s a sphere with diameter $d/2$. We say p is k-free if $|\{p_j \in P \mid d \leq d_j \text{ and } p_j \cap (p \oplus s) \neq \}| < k$, with d_j representing the diameter of smallest enclosing sphere of polyhedron p_j and $p \oplus s$ the Minkowski-sum of s and p. Consequently, we can declare a set of polyhedra P k-free, if all polyhedra $p_i \in P$ are k-free.

That is to say, a k-free polyhedron p has fewer than k polyhedra of P that (1) intersect the Minkowski-sum $p \oplus s$, being the volume resulting from sweeping a sphere with diameter $d/2$ around p, and (2) have a larger minimum enclosing sphere than p itself. From here on,

1 UEC Tokyo - Homepage: https://www.uec.ac.jp/eng/

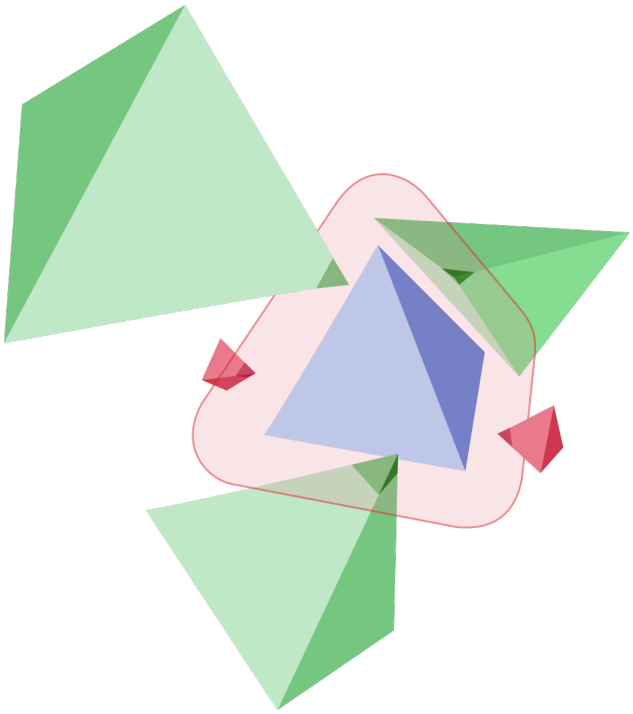


Figure 3.1: 4-*free* tetrahedron (blue); red area marks the Minkowski-sum of said tetrahedron and a sphere half the diameter of its minimum enclosing sphere; a 4-*free* polyhedron intersects at most 3 "larger" polyhedra (determined by the minimum enclosing sphere) with said Minkowski-sum

such polyhedra will simply be referred to as *larger polyhedra*. Moreover, assume two polyhedra p_i, p_j with minimum enclosing spheres s_i, s_j of diameters d_i, d_j. Then, if $d_i \geq d_j$ we say s_i is larger than s_j. Figure 3.1 shows an example of a k-free tetrahedron, where $k = 4$.

Please note, that any constant k suffices for our theoretical analysis. For sake of simplicity, the constant k will be referred to interchangeably for individual polyhedra and the complete set.

We will now show that a single polyhedron cannot intersect too many larger polyhedra of a k-free polyhedron set:

Lemma 3.1.1. Let P be a k-free set of polyhedra and let $p \notin P$ be an arbitrary polyhedron. Under those circumstances, the number of larger polyhedra $p_j \in P$ which p intersects is bound by a constant. Next, let q be the (maximum) number of covering spheres necessary for each polyhedron in question, according to our special case of the sphere covering problem. Then qk is the maximum number of intersections between p and larger polyhedra $p_j \in P$.

Before we move on, let us clarify what are "sphere covering problems" and the meaning of q. In general, they are a class of problems that deal with covering a volume or area with (usually) a minimum num-

ber of spheres of a specified quality [2]. The sphere covering problem we are interested in, is the following: Assume we have a polyhedron p and we know its minimum enclosing sphere, which has diameter d. We now want to figure out, how many spheres with diameter $d/2$ are necessary to gaplessly cover the volume enclosed by p and how they need be arranged to achieve that. The former quantity is represented by q, and varies for different classes of polyhedra.

The proof of lemma 3.1.1 is not really concerned with the exact value of q. Rather, it only requires that a class of object can be covered with a constant number of covering spheres. Later, we will determine some values for q by considering this problem in detail for two classes of polyhedra, being arbitrary tetrahedra (chapter 3.2.0.1 and arbitrary polyhedra (chapter 3.2.0.2.

Proof. Let s be the minimum enclosing sphere of p, having diameter d. We construct a sphere covering of p with spheres $s_i \in S$ each of diameter $d/2$, where $i \in 0, 1, \ldots, q$. For spheres s_j there can then be at most k larger polyhedra $p_i \in P$ that intersect it.

The proof works by contradiction: Suppose that $k+1$ larger polyhedra intersect sphere s_j, where p_a is the smallest of these polyhedra and d_a the diameter of its minimum enclosing sphere. Since p_a is larger than p, we get $d \leq d_a$, by definition. With s_j being intersected by p_a, it means that s_j is completely located inside $p_a \oplus s_a$, due to the fact that the diameter of s_j is $d/2$. Therefore, $p_a \oplus s_a$ would be intersected by k larger polyhedra, contradicting the prerequisite of Lemma 3.1.1 that p_a is k-free. □

We move on to prove a linear bound on the number of intersecting polyhedra with a constant factor for all objects that have a k-free polyhedron mesh.

Theorem 3.1.2. Presume that A and B are two k-free sets, each consisting of n polyhedra, and in collision. In such a case, the total number of colliding polyhedra is in $\mathcal{O}(n)$. More precisely, the number is at most qnk, with q being the (maximum) number of covering spheres necessary for each polyhedron, according to the aforementioned sphere covering problem.

Proof. For each of the sets A and B, we test each polyhedron of that set against all larger polyhedra of the other set. Each of these polyhedra returns at most qk intersections with larger polyhedra, according to Lemma 3.1.1. Furthermore, we are guaranteed to find all pairs of colliding polyhedra, because in a pair of intersecting polyhedra either of the polyhedra must be larger than the other. Since for each of the n

2 Technically "covering" of volumes would need to be done with balls and not spheres, as spheres only represent the boundary of balls without their interior. However, in this work I will forego this technicality and just refer to them as spheres.

polyhedra the number of intersections with larger polyhedra does not exceed qk, we have at most qnk intersecting pairs when A and B are colliding. □

3.2 SPHERE COVERINGS

In this chapter, we will deal with the topic of covering arbitrary tetrahedra and polyhedra with spheres half the size of their minimum enclosing sphere. The minimum number of spheres necessary for each covering gives us the constant q, which is a defining factor in the upper bound of intersections between k-free sets. For the original version of kDet, this q was shown to be 3 for triangles, and 7 in the case of arbitrary polygons [WDZ17]. In the following, I will prove that for tetrahedra $q = 4$, and $q = 21$ for arbitrary polyhedra.

3.2.0.1 *Tetrahedra*

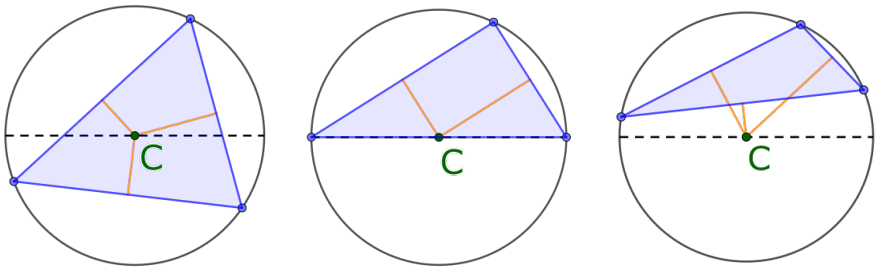
Theorem 3.2.1. Let t be an arbitrary tetrahedron with minimum enclosing sphere s of diameter d. Then t can fully be covered with 4 spheres of diameter $d/2$ (from here on, when referring to *covering spheres* this quality of the diameter is implied).

Before we get to the proof, we consider a few points and ideas:

1. First, the smallest enclosing sphere of a tetrahedron is either given by its circumscribing sphere or the smallest enclosing sphere of one of its faces [BSA17]. In the latter case, the circumscribing sphere might not touch all points of that face, e.g. if the simplex is very long and thin.

2. By looking at some properties of minimum enclosing circles of triangles, we can draw some parallels to minimum enclosing spheres of tetrahedra. For acute and right triangles, the circumscribing circle would describe the minimal enclosing circle, while for obtuse triangles the circle with diameter of the hypotenuse, placed on the center of that edge, would provide said circle [WDZ17].

   We make another observation, in that the circumcenter for acute triangles lies strictly inside the triangle, i.e. not on an edge of the triangle; for obtuse triangles it is outside the triangle; for right triangles it is strictly the midpoint of the hypotenuse (see figure 3.2). This is an equivalence relation and we can reformulate these into a predicate for determining the minimal enclosing circle of a given triangle: If the circumcenter lies inside or on an edge of the triangle, then the minimum enclosing circle has to be the circumcircle. On the other hand, if the circumcenter is found outside the triangle, the minimal enclosing circle has to be the one with diameter of the hypotenuse.



(a) (b) (c)

Figure 3.2: Circumcenters for different kinds of triangles. Orange lines represent the perpendicular bisectors. (a) The circumcenter of an acute triangle is inside of the triangle itself. (b) The circumcenter of a right triangle is located on the center of its hypotenuse. (c) The circumcenter of an obtuse triangle is located outside of itself.

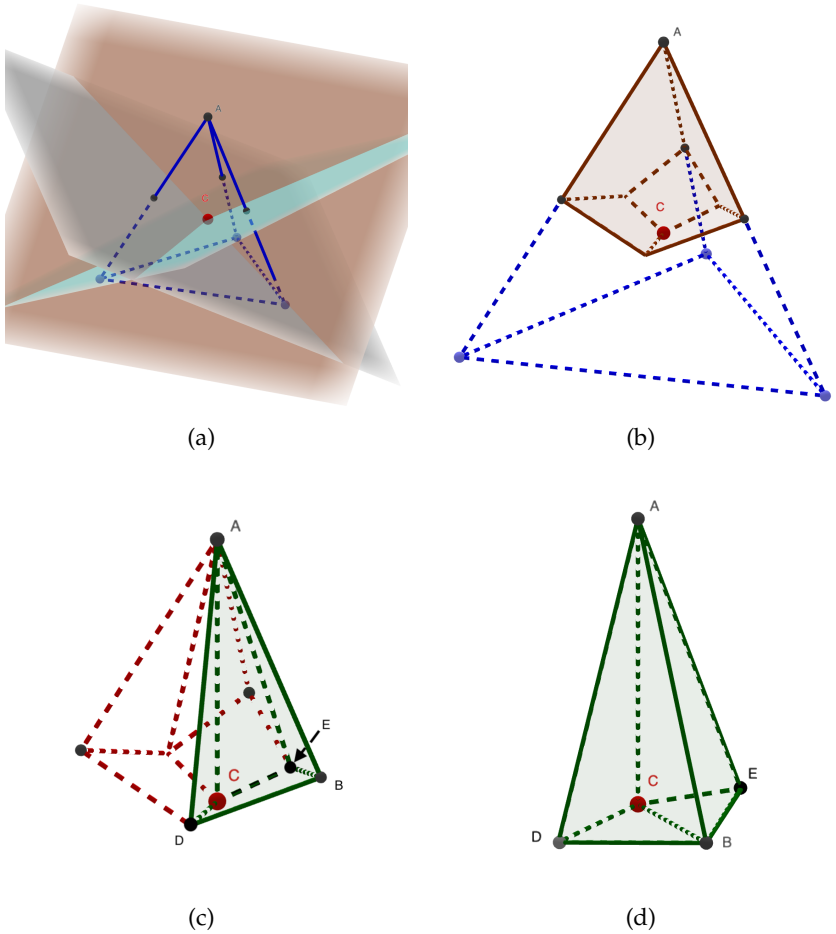
How can this insight be applied? Contrary to triangles, tetrahedron cannot be trivially put into a few categories (like acute, right, and obtuse). Instead, we can use this intuition about the position of the circumcenter, to reduce the number of classes of tetrahedra we have to look at; namely tetrahedra which have their circumcenter inside, on a face or edge, or outside.

3. We extend the idea from 2. to tetrahedra: If the circumcenter lies inside of the tetrahedron or on one of its faces or edges, then the circumscribing sphere will be the minimum enclosing sphere. This also implies that all 4 of the tetrahedron's vertices touch the minimum enclosing sphere. If the circumcenter is outside of the tetrahedron, then the minimum enclosing sphere is given by the minimum enclosing sphere of that tetrahedron's largest face [BSA17]. More specifically, if said largest face is an acute or right triangle, then 3 vertices will touch the sphere. If it is obtuse, only two vertices will touch it.

Now that we are able to classify tetrahedra and have an idea on how to find the minimum enclosing sphere for a given tetrahedron, let us look at how this knowledge can be used to cover any tetrahedron with 4 covering spheres.

Proof. Let t be an arbitrary tetrahedron with minimum enclosing sphere s of diameter d. We consider the following three cases:

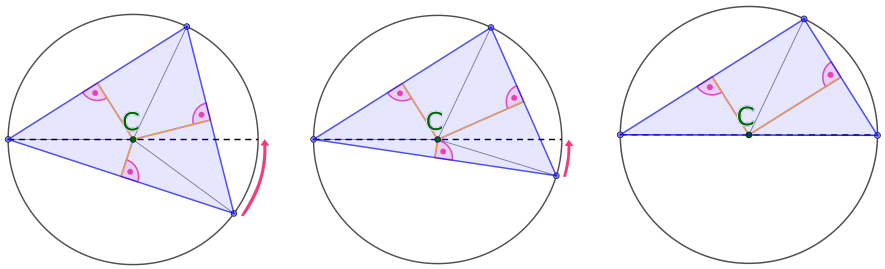
1. The circumcenter of t is inside itself, but not on either of its surfaces or edges
2. The circumcenter of t is on of its surfaces or edges
3. The circumcenter of t is outside itself



(a) (b)

(c) (d)

Figure 3.3: (a) Determining the circumcenter of a tetrahedron with bisector planes. (b), (c) Cutout from the tetrahedron using the bisector planes. (d) Pyramid cutout from the structure in (c). Point C is the circumcenter of the tetrahedron, B is the intersection point of the edge and the bisector plane, D and E are intersection points of two bisector planes and a face of the tetrahedron.



(a) (b) (c)

Figure 3.4: We subdivide a triangle into six sub-triangles by using the perpendicular bisectors and the lines connecting the circumcenter to the vertices of the triangle. When transforming an acute triangle into a right one, by moving one of the vertices along the circumcircle, two of the sub-triangles "collapse" into the hypotenuse.

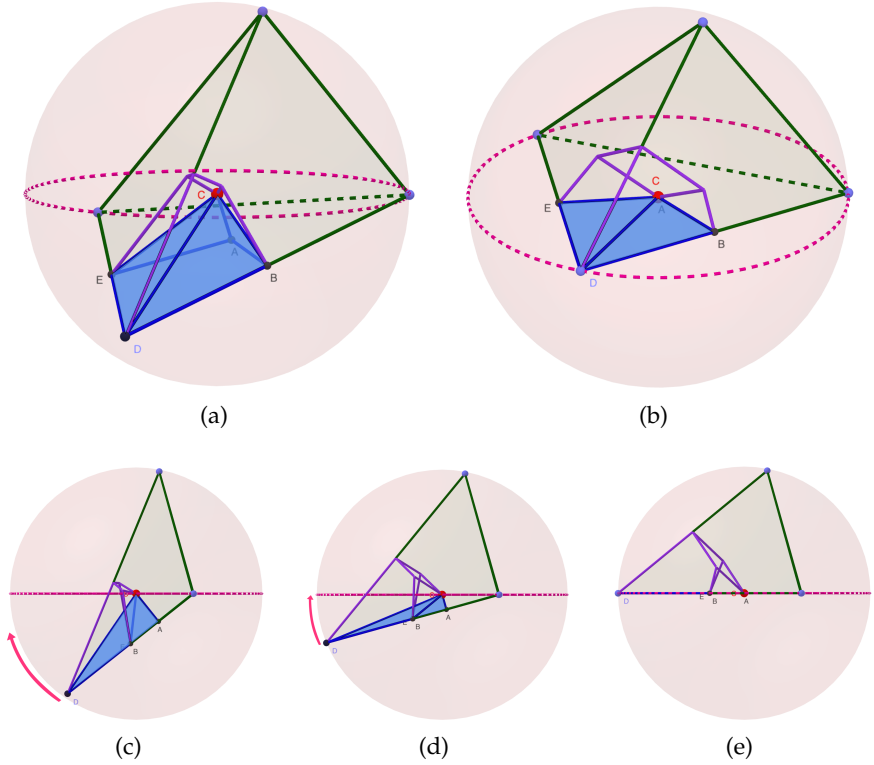


Figure 3.5: Showing how certain substructures (blue) in our tetrahedron subdivision can sometimes 'collapse' into a face of the tetrahedron. (a), (b) Before and after of the tetrahedron. We can clearly see in (b) the substructures 'collapsed' into the bottom face and A and C now coincide in position. (c), (d), (e) Transitioning point D so it is on the equator circle with the other two vertices.

Case 1 and 2: From our previous deductions, we know that in these cases the circumscribing sphere is given by the minimum enclosing sphere. The circumscribing sphere is constructed by first determining the circumcenter, using perpendicular bisector planes on three edges which share a common vertex. Figure 3.3a demonstrates this with the vertex A and the adjacent edges, yielding us the circumcenter C. By drawing all the bisector planes, the tetrahedron can be subdivided into four structures similar to the ones shown in figure 3.3b and 3.3c (basically one structure for each vertex). If we can show that each of these structures can be contained within a covering sphere, then four covering spheres will suffice for the whole tetrahedron. In one last subdivision, each of these structures can be partitioned into three double-tetrahedra as shown in figure 3.3d. Later, when discussing the edge cases, we will see why we should not classify this structure as a pyramid, even though for "regular" tetrahedra this would certainly apply.

Let us stay on figure 3.3d: C is the circumcenter, A an original vertex of the tetrahedron, B a midpoint of an original edge connected to A, while D and E are the intersection points of the faces adjacent to A and the intersection lines of the perpendicular bisector planes placed on the edges adjacent to A. We note that B, C, D, and E are co-planar since we obtained them from the bisection of the tetrahedron with the perpendicular bisector plane. In addition, we note that the angles between $\overline{AB}$ and $\overline{BC}$, $\overline{AD}$ and $\overline{CD}$, as well as $\overline{AE}$ and $\overline{CE}$ are all right angles.

For the former, it should be apparent why this is the case, because BC is the line segment co-planar to the bisector plane connecting the intersection point of the bisector plane and the edge (= point B) to the circumcenter C. For why the latter two enclosed angles are right angles, we need to take a better look at points D and E. Both of them are the intersection points of two bisector planes (of adjacent edges) and a tetrahedron face. The intersection of two planes which are not parallel is always a line, meaning that D and E are specifically the intersection points of such lines and a tetrahedron face. Note, that in our case all such lines meet in the circumcenter C and stand perpendicular to the tetrahedron's face. This is because the bisector planes were already perpendicular to the face, thus their intersection line has to be as well. Since $\overline{CD}$ and $\overline{CE}$ are segments of these lines, it means the angles in question need to be right angles as well.

We can now place a covering sphere on the midpoint of $\overline{AC}$. Triangles ABC, AEC, and ACD are then contained within that covering sphere according to Thales's Theorem, since all the important angles in question are right angles. Each of the triangles is co-planar with a "circle intersection" of the said sphere, referring to the circle we would obtain if we slice the sphere with the plane co-planar to any of the given triangles. Then $\overline{AC}$ describes the hypotenuse for all of

the mentioned triangles, and since we showed the leftover angles are all right angles, they have to be contained by the circles according to Thales's Theorem. The circles are part of the covering sphere, thus all the triangles are contained by it as well.

The argument now applies recursively: Since we showed that the mentioned triangles are contained inside the covering sphere, all the other structures than built upon them to be contained have to be inside it as well. Thus we have shown that by placing covering spheres with centers on the midpoints of the sections connecting the circumcenter and the vertices, we can cover the whole tetrahedron with exactly four spheres.

Note, that for tetrahedra which coincide with case 2, the structures in figure 3.3b and 3.3b are different, in that they contain fewer sub-structures. We can better understand this, by looking at a lower-dimensional analogy: Acute triangles can be subdivided into six sub-triangles that need to be contained within the covering circles, while for right and obtuse triangles we only get four sub-triangles, despite using the same principles. The placement of the circles remains the same though, in that we place them on the midpoints of the sections connecting the circumcenter to the vertices (for obtuse triangles we expanded the triangle to be a right one since we could do so without loss of generality, meaning again the circumcenter was the center of the minimum enclosing circle for the resulting triangle). It is just that for right triangles, two of these sub-triangles basically 'collapse' into the hypotenuse of the triangle (= they have no area; see figure 3.4), but still the same principles applied to show that the 4 remaining partial triangles are contained within the covering circles.
Similarly, some of these substructures can 'collapse' into the faces of the tetrahedron (= they have no volume) as shown in figure 3.5, but we can still use the same methods (i.e. Thales's theorem) to show that the remaining structures are contained within the covering spheres. This is the reason, we chose not to refer to the structure in figure 3.3d as a pyramid in general, since in such cases it can also be a single tetrahedron.

Case 3: Here we have to make another distinction between tetrahedra which touch the enclosing sphere with either two or three vertices:

- If two vertices touch the minimum enclosing sphere: Let C be the center of the enclosing sphere and P, Q the vertices of the tetrahedron which do not touch the enclosing sphere. We push each point to the surface of the sphere in the direction that would move C to P or Q respectively (see figure 3.6a and 3.6b).

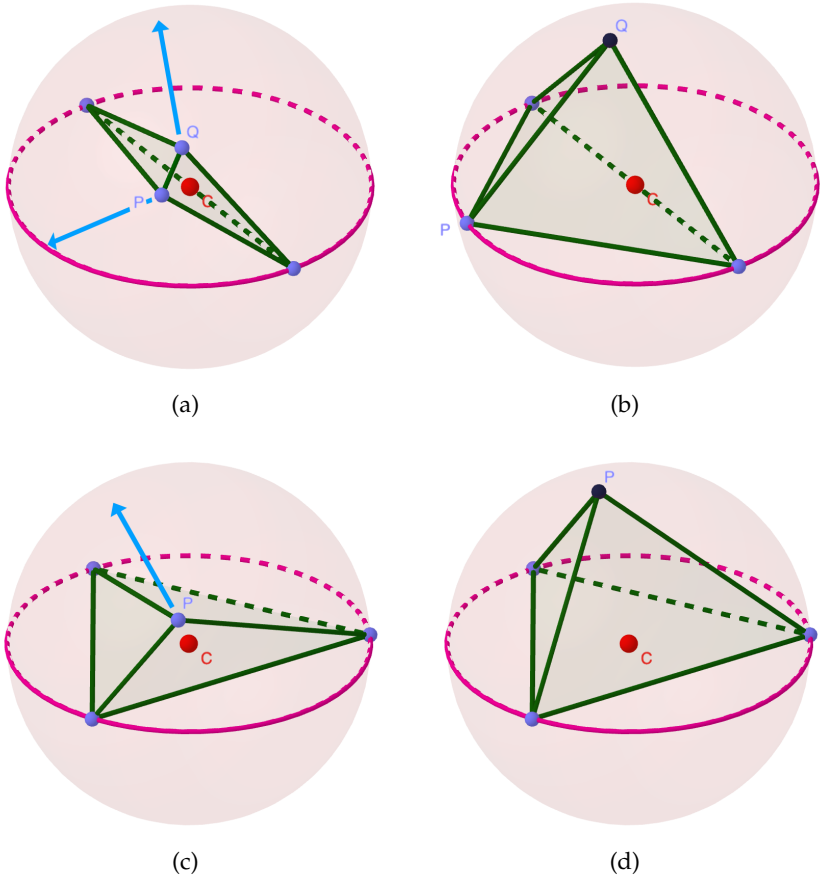


Figure 3.6: (a) tetrahedron with obtuse triangle as its largest face and its minimum enclosing sphere (b) result of moving P, Q to the surface of the minimum enclosing sphere (c) tetrahedron with acute triangle as its largest face and its minimum enclosing sphere (d) result of moving P to the surface of the minimum enclosing sphere

- For the case of three vertices touching the enclosing sphere, we do the same as the previous case, but only for the single point not on the surface of the sphere (see figure 3.6c and 3.6d).

In both cases we now have a tetrahedron with the same minimum enclosing sphere as the original one, but with the following additional properties: 1) It encloses the original tetrahedron, and 2) The circumcenter is also the center of the minimum enclosing sphere (= four vertices on the minimum enclosing sphere), therefore the circumsphere is the minimum enclosing sphere

We have already proven that tetrahedra which have the circumsphere as their minimum enclosing sphere can be covered with four covering spheres. Therefore, we have shown that this covering extends to all kinds of tetrahedra. □

3.2.0.2 *Spheres*

Unlike tetrahedra, arbitrary polyhedra require more than only 4 covering spheres. Since the covering depends on the specific type of polyhedron in question, it is not possible to suggest a general solution in this matter. Instead, we aim for a covering of the minimum enclosing sphere of the arbitrary polyhedron and thus reach a upper bound for the number of covering spheres.

Verger-Gaugry [VG05] proposed that the volume enclosed by a sphere $s_{r,d}$ with radius $1/2 < r \leq 1$ in dimension $d \geq 2$ could be covered with spheres of radius $1/2$ as follows: Put the first covering sphere at the center of $s_{r,d}$; we will call this the *inside sphere*. Next, place covering spheres at equivalent distance from the center and such that spherical caps created by the intersection of the covering spheres with $s_{r,d}$ have bases that are of radius $1/2$; these spheres will from here on referred to as *outer spheres*. Additionally, the outer spheres overlap with the inside sphere, leaving no gaps, giving a full covering of $s_{r,d}$. This simplifies the problem to determining how many spherical caps of the mentioned quality cover the surface of $s_{r,d}$.

Following the proposed strategy, we can prove that a sphere in $\mathbb{R}^3$ can be covered with at most 21 spheres half its radius [Wyn12].

Theorem 3.2.2. A sphere of radius r can be covered with of 21 spheres of radius $r/2$.

We prove this theorem for the unit sphere, as this will trivially extend to spheres of any radius $r \in \mathbb{R}$.

Proof. Let us take a look at figure 3.7: Let the black sphere be the unit sphere ($r = 1$) and the orange ones be covering spheres. First, we place the inner sphere with its center C aligning to the unit sphere's center. We continue to place outer spheres to cover the surface of the

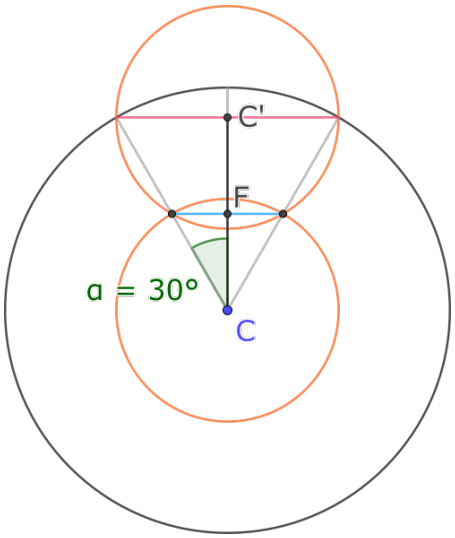


Figure 3.7: Covering the unit sphere (black, $r = 1$) with covering spheres (orange) of radius $r = 1/2$. We have an *inside sphere* whose center aligns with the unit sphere, and *outer spheres* that are placed such that the bases of their spherical caps are of radius $1/2$.

unit sphere, such that the bases of the spherical caps are of radius $1/2$. With this strategy, we will cover the maximum amount of area from the unit sphere's surface for each outer sphere.
We want to know how many of these spherical caps are needed to cover the surface of the unit sphere. For that we first need to determine the angle α of these spherical caps using the basic *sin*-law:

$$\sin(\alpha) = \frac{opposite\ side}{hypotenuse} .$$

We know all the gray line segments have length 1 (as they represent the radius of the unit circle). The orange spheres are half as big, thus their radius is $1/2$, which is half the length of the red line. We insert the values and take the *arcsin*:

$$\Leftrightarrow \sin(\alpha) = \frac{(1/2)}{1} = \frac{1}{2}$$

$$\Leftrightarrow \alpha = \arcsin(\frac{1}{2}) = 30^\circ .$$

We can also use cos to determine the length of the black line segment connecting C and $C\prime$, i.e. the distance from the center at which the outer spheres are placed:

$$\cos(\alpha) = \frac{adjacent\ side}{hypotenuse}$$

$$\Leftrightarrow \cos(30) = \frac{adjacent\ side}{1}$$

$$\Leftrightarrow \frac{\sqrt{3}}{2} = adjacent\ side = \overline{CC\prime}\ .$$

What's more, is that this implies that the inside and outside spheres overlap, since the distance of their centers is less than their combined radii:

$$\frac{\sqrt{3}}{2} < 1/2 + 1/2 = 1\ .$$

According to the computed table provided by Hardin, Sloane, and Smith [HSS95], in order to fully cover a sphere with spherical caps of 30° one needs exactly 20 spheres. In addition, the outer spheres intersect the inside sphere with the same spherical cap of 30° (we can confirm that by using sin and arcsin again, knowing that $\overline{CF}$ is half of $\overline{CC\prime}$ and then solve for α). So covering the surface of the unit sphere will at the same time cover the surface of the inside sphere which provides us a gapless covering of the unit sphere using $20 + 1 = 21$ spheres of radius $r = 1/2$. □

4

ALGORITHMS & IMPLEMENTATION

4.1 BASE IMPLEMENTATION

The base implementation was ported by Hermann Meißenhelter from the original kDet implementation and is written in C++. Figure 4.1 shows the collision pipeline of said implementation. It contains only two steps: 1.) populate the spatial grid, and 2.) check for collisions. The first step inserts all the simplices of the objects into the hashed spatial grid. The second step traverses the grid for each object and checks for collision against the primitives of the other. To do so, it searches for all potential collision partners in a tetrahedron's vicinity, and executes the intersection tests for the simplices.

The hash table that implicitly stores the grid uses bucket hashing. Each bucket has 133 entries: 128 for storing tetrahedron ids, and 5 entries that are used as flags. These flags include the frame counter, the spatial-id of the currently hashed grid-cell, and the hash of the next bucket if there is overflow. The flag for the frame counter is beneficial in that spares us the time for resetting the entries of the hash table each frame. When visiting a bucket while populating the grid, we can just check the flag and if it is not the current frame, we can treat it like an empty bucket.

With this, we can look at algorithm 4.1 presenting the procedure for populating the grid:

First we check if the tetrahedron is in the overlap are of the bounding boxes of the collision objects, and if not, we can already return. This reduces the number of tetrahedra that need be inserted into the grid and accelerates traversal during the collision checks. This method can however not be applied, when self-collisions have to be detected. We continue by determining the "size" of the tetrahedron and the grid layer it should be inserted in. Next, we obtain all grid cells that are intersected by the tetrahedron. For all those cells we then continuously calculate a hash and inspect the hash table until we either find a) a bucket that is already used for this grid cell or b) an empty bucket. In either case, we then try insert the tetrahedron into the bucket. If the bucket in case a) is full however, we need to jump to the next bucket that is designated to the same grid cell.

Next, algorithm 4.2 demostrates how the grid is traversed when checking for collisions:

We start by determining all cells that are intersected by the tetrahedron in its own grid layer and all higher layers. For each of these cells,

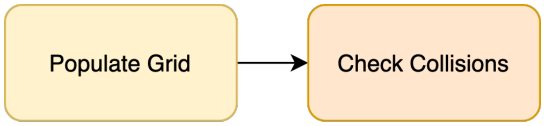


Figure 4.1: Collision Pipeline of the old Version of kDet

Algorithm 4.1 Populate Grid

Input: object A with tetrahedral mesh
bbox overlap area of both collision objects A, B
Result: inserts all tetrahedra of the mesh into the hashed spatial grid accordingly

```
1:  for all t ∈ O do in parallel
2:      if t is not in bbox overlap area then
          return
3:      end if
4:      determine size s of t
5:      determine grid layer l of t based on s
6:      find set C of all grid cells intersected by ton layer l
7:      for all cells c ∈ C do
8:          calculate hash key k based on grid position g of c
9:          while k does not point to bucket b that is either empty
                or has the same hashed grid position g do
10:             increase hash misses m by 1
11:             recalculate hash k with g and m
12:         end while
13:         if bucket b is empty then
14:             atomic: set grid position of b to g
15:         else if bucket b is full then
16:             recalculate hash key k until a suitable overflow
                    bucket b_o is found
17:             atomic: set grid position of b_o to g
18:             set b's reference of overflow bucket to b_o
19:             atomic: increase tetrahedron counter of b_o by 1
20:             insert t intro bucket
21:             continue
22:         end if
23:         atomic: increase tetrahedron counter of b by 1
24:         insert t intro bucket
25:     end for
26: end for
```

Algorithm 4.2 Check Collisions Grid

Input: objects A, B with tetrahedral meshes
Output: all tetrahedra $t_A \in A$ that collide with larger tetrahedra $t_B \in B$

```
1:  for all t_A ∈ A do in parallel
2:      determine size s of t
3:      determine grid layer l of t based on s
4:      find set C of all grid cells intersected by t on all layers l_i ≥ l
5:      for all cells c ∈ C do
6:          calculate hash key k based on grid position g of c
7:          while k does not point to bucket b in hash table
                with same hashed grid position g do
8:              increment hash misses m by 1
9:              recalculate hash k with g and m
10:         end while
11:         for all tetrahedra t_i ∈ B hashed into bucket b do
12:             if t and t_i are both already marked as colliding then
13:                 continue
14:             else if bbox of t and t_i do not overlap then
15:                 continue
16:             end if
17:             if t and t_i have intersection then
18:                 set t & t_i to colliding
19:             end if
20:         end for
21:         if bucket b is has reference to overflow bucket b_o then
22:             update b to b_o
23:             go-to line 10
24:         end if
25:     end for
26: end for
```

we, again, find the all hash buckets that correspond to them. We will then try to check for collisions against each tetrahedron that is hashed into those buckets. To reduce the number of intersection tests, two techniques are applied beforehand: 1) A bounding box test between the tetrahedra is executed, and 2) if both tetrahedron have a flag set, that they already have another collision partner, then the intersection test is skipped. Only if both measures do not apply, do we execute the intersection test. If it returns an intersection, we set the according flag for each tetrahedron. A consequence of the second acceleration trick is that the original kDet would not find all collision pairs, only all simplices that were part of the collision set.

4.2 FINDING ALL PAIRS

As explained in the previous chapter, the original implementation merely finds all simplices which are part of the collision set, but importantly not all collision pairs between these simplices. This might be a problem when computing collision responses, as it might be necessary to compute the response on a per collision pair basis. Therefore, it was necesseray to remove this acceleration trick for the new implementation. Now, after removing the acceleration trick, kDet will suddenly find the same collision pair several times. The is caused by the fact that during the insertion of a simplex into the grid, it can be hashed into multiple adjacent grid cells. We would like to avoid these duplicate collision pairs, the reason being twofold: First, this prevents the physics pipeline from applying the force responses of a collision pair multiple times. This is especially useful, in case we do not want to explicitly store, sort, and filter the collision responses, before applying them after the collision detection is finished, which would require additional computational effort. Second and more importantly, we want to avoid the additional computations for the intersection tests of the duplicate pairs.

Due to parallelization and the possible race conditions, it is rather difficult and expensive to check during run-time whether another pair has already been checked for collision with a naive approach. One such naive approach could be to simply write all checked pairs into a list and check before the execution of the intersection test if the pair is already included in the mentioned list. We immediately see why this would be inefficient: Imagine multiple threads for the same collision pair checking the list at the same time and not finding their pair. They would all proceed to execute the intersection test. Afterwards, to avoid duplicate insertions into the list, the list would need to be checked once again, before the insertion is done as an atomic operation. One way to accelerate this, is to not check the list beforehand, but to always execute the intersection test and only check the list afterwards. Another way would be to insert all pairs into the

	B1	B2	...	Bm
A1	x		...	x
A2		o	...	o
...	...	...	...	...
An		x	...	

Figure 4.2: Example of a Collision Table: If a pair of primitives is to be checked, we first look whether there exists an entry within the table. If objects have not been checked yet, they have no entries. Otherwise, we fill in whether they are intersecting or not.

list and sort afterwards. But this would mean that the intersection test was computed for the same collision pair several times. For these reasons, several approaches were considered to mitigate this issue.

4.2.1 *Collision Table*

The simplest idea that comes to mind is a table that tracks the collisions of all possible combinations of tetrahedra. An example for such an collision table is presented with table 4.2. The table entries store information about whether a potential pair has been (or is in the process of being) checked or not. So each time, before running the intersection test for a potential collision pair, we check the respective entry in the table. If it is set to *not-checked*, we change the entry to *checked* and continue with the intersection test. Otherwise, we know the pair has already been checked or is in the process of getting checked, and we can continue. Importantly, the check and set of the collision table has to be done as an atomic operation.

Having said that, we face one important problem with this approach, being that the size of the table grows proportionally to the number of tetrahedra in the scene squared. Imagine the table for two collision objects with 100.000 tetrahedra each. The resulting collision table would have 10^{10} entries. Let us assume the table entries are represented with booleans, which in C++ need 1 byte of memory. Then the resulting table would have a size of ~9,3 GiBi [1]. Additionally, a consequence of the table size would be higher numbers of random memory accesses, which are rather expensive on the GPU.

1 Technically, the size of the table can be halved because entry (x, y) would correspond to the same pair as (y, x). Still, the size of the table remains in the same order of magnitude.

4.2.2 *Prime Factorization*

Another idea that was suggested to me, was to make use of the prime factorization theorem, which states that any positive integer > 1 can be represented as a product of prime numbers and that the factorization of that product is unique. For example, $500 = 2^2 \times 5^3$.

By assigning each tetrahedron a unique prime number and keeping track of the product of the primes of all other tetrahedra that the tetrahedron already checked against in a frame (from here on referred to as *prime-product*), it is possible to detect duplicate potential collision pairs during runtime with some divisibility checks. To be more specific, if we have two tetrahedra A and B, and they have previously been checked for collision, then one factor of each of their respective prime-product's has to be the assigned prime of the other, making them divisible by said prime. Algorithm 4.3 demonstrates the procedure for such a check.

Algorithm 4.3 Determine If Already Checked - Primes

Input: two tetrahedra t_A, t_B
Output: returns whether t_A, t_B have already been checked for intersection

Require: all $t_i \in$ A $\cup$ B are assigned a unique prime p_i, and a prime-product q_i initialized with the value of p_i

1: **if** q_{tA} is divisible by p_{tB} **then**
    return t_A, t_B have already been checked for intersection
2: **else**
3: **atomic** $q_{tA} \leftarrow q_{tA} \times p_B$ and $q_{tB} \leftarrow q_{tB} \times p_A$
4: proceed to intersection test between t_A, t_B
5: **end if**

To start off, the algorithm requires us to give each tetrahedron of the scene a different prime number in pre-computation. In addition, each tetrohedron keeps a prime-product, which on each frame is initialized with the value of the tetrahedron's prime. When checking if two tetrahedra have already collided, we determine if the prime-product of tetrahedron A is divisible by the prime of tetrahedron B. This can be done with a simple modulo calculation: If $a \mod b = 0$, then b is divisible by a. If the prime-product of tetrahedron A is not divisible, we multiply the prime-product of both tetrahedron A and B with the assigned prime of the other and proceed with the intersection test. When the prime-product is divisible however, we know that tetrahedron A and B have already been checked, and we exit. This check and swap of the prime-product, once again, has to be done as an atomic operation to avoid race conditions.

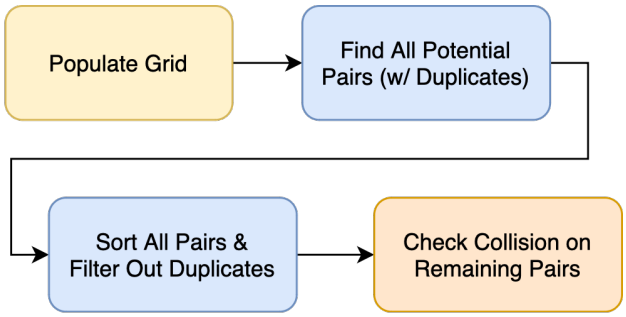


Figure 4.3: Collision Pipeline of new version of kDet with multiple phases.

While elegant and theoretically possible, this approach fails in practice simply due to the limited size of integers. The growth of the prime-product for each tetrahedron is quasi factorial. Consequently, even with integers of type `long long int`, which usually can take a maximum value of up to $2^{63} - 1 \approx 9,2 \times 10^{18}$ (depending on the compiler), overflow occurs frequently. For example, if we take the following list of primes, which are all within the first 100,000 prime numbers, and multiply them, we get:

$$7 \times 13331 \times 73961 \times 268789 \times 755033 \approx 1,4 \times 10^{21}$$

, which is already three orders of magnitude larger than the maximum value for `long long int`'s. We will see another possible application of the prime factorization theorem in chapter 7.1.

4.2.3 *Multiple Phases*

The solution I settled on, actually does not check for duplicate pairs during the initial traversal of the grid. Instead, an approach with multiple phases is used, as presented in figure 4.3.

In the first phase, we traverse the grid and write all potential collision pairs to a list, including duplicate ones. As shown in algorithm 4.4, the procedure for that is almost identical to algorithm 4.2, except for where the insertion of the pair into the list replaces the intersection test and acceleration trick. This is done by incrementing a global counter and then inserting the pair into the list at the returned position. The incrementation of the counter, again, has to be done as an atomic operation, to avoid race conditions. Note, that again the traversal of the grid has to be done for the first object against the second, and vice versa, or else not all collision pairs can be found later.

In the second step, we then filter out all duplicate pairs, before moving to the final phase where we execute intersection tests for the remaining potential pairs.

Algorithm 4.4 Find All Potential Collisions Pairs

Input: objects A, B with tetrahedral meshes
Result: writes all potential collision pairs (t_A, t_B) to a list, with $t_A \in A$, $t_B \in B$ and t_A being smaller than t_B; can find the same pair multiple times

```
1:  for all t_A ∈ A do in parallel
2:      determine size s of t
3:      determine grid layer l of t_A based on s
4:      find set C of all grid cells intersected by t_A on all layers l_i ≥ l
5:      for all cells c ∈ C do
6:          calculate hash key k based on grid position g of c
7:          while k does not point to bucket b in hash table
                with same hashed grid position g do
8:              increment hash misses m by 1
9:              recalculate hash k with g and m
10:         end while
11:         for all tetrahedra t_B ∈ B hashed into bucket b do
12:             if bbox of t_A and t_B overlap then
13:                 atomic: write pair (t_A, t_B) into list of potential
                      collision pairs
14:             end if
15:         end for
16:         if bucket b has reference to overflow bucket b_o then
17:             update b to b_o
18:             go-to line 10
19:         end if
20:     end for
21: end for
```

The list is implemented as a device (= GPU) vector from the CUDA `thrust`-library [Dev23], which offers a similar API to that of `std::vector`. The list uses CUDA's `int2` as its entry data type, which is essentially just a wrapper for 32-bit integers. The vector can be resized, but not during CUDA kernel execution. This means that the size of the vector has to be declared beforehand. Additionally, the vector has to be large enough to hold all potential collision pairs with duplicates, while not exceeding the memory capacities of the GPU. For the current implementation, I chose a fairly arbitrary heuristic to determine the size of the device vector. Currently the vector size is calculated with this formula:

$$size_{vector} = c\ (\#tetrahedra_A \times \#tetrahedra_B), \text{ where } c = 1/10.$$

The rationale was that the size of the vector should be proportional to the maximum combinations of possible collision pairs multiplied by some constant, which in this case was meant to reduce the vector size since I did not expect to have that many collision pairs. I also tested smaller values for c, such as $c = 1/50$ or $c = 1/100$,. While these would work in some cases, $c = 1/10$ was the smallest value that would not crash the program for any of the provided scenes, even with self-collisions turned on.

During development I did not spend much time refining this heuristic, as it was sufficient for my work. However, we can already note that memory demands for the vector would increase by a square-law, similar to the collision table. Ideally the size of the vector would grow linearly. Determining a suitable size of the device vector is its own topic of discussion, which we will further explore in chapter 7.3.

Finally, for the filtering of the duplicate pairs, I use a built-in function of the `thrust`-library, called `unique()`. This method reorders a specified range of the vector such that all unique elements are put to its front. It then returns the index of the last unique element, which gives us the range of all non-duplicate items in the list. The `unique()` function requires the vector to be sorted prior. For that `thrust` offers an adequately named `sort()` method as well. Note, that we never need to sort the whole list, only the first n entries, with n being the number of potential collision pairs (with duplicates) detected that frame. For the sorting and filtering to work properly, the pairs that are placed into the list need to follow a strict ordering. One example of such an ordering would be that the first entry of the pair always corresponds to a tetrahedron of first object, while the second entry always corresponds to a tetrahedron of the second object.

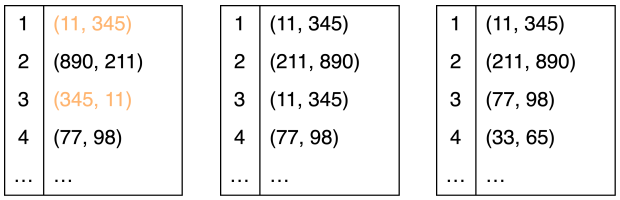


Figure 4.4: The list of potential pairs during self-collision detection normally would include duplicate pairs, since the pairs (11, 345) and (345, 11) reference the same tetrahedra (a). By enforcing a strict ordering of the entries (b) before sorting (c), we can forego this problem.

4.3 SELF-COLLISIONS

The procedure for finding all self collision pairs is implemented almost identically to the normal multi-phase method for finding all pairs. The differences are minute, but important.

First of all, when checking for self-collisions, populating the grid cannot use the acceleration trick described in chapter 4.1 with the bounding box overlap anymore. That trick would cause tetrahedra outside of the bounding box overlap area of the two models to not be placed into the hashed grid. Since all tetrahedra are relevant for self-collisions, we need to hash them all into the grid instead.

Second, when searching for all potential pairs, we (obviously) traverse the grid for the object against itself.

Lastly and most importantly, we need to add an additional step before filtering out duplicate pairs. Consider the situation demonstrated in figure 4.4: The list of potential pairs includes the pairs (11, 345) and (345, 11). When checking for inter-object collisions, the order of the pair encodes which object an tetrahedron belongs to. But for self-collisions, both tetrahedra of the pair are from the same object; therefore, the pairs (11, 345) and (345, 11) are identical in this case and would be considered duplicates, even after filtering. This can be solved by just reordering the entries of each pair, such that the first entry is always smaller than the second one [2], as shown in figure 4.4b. Anew, the `thrust`-library has a suitable function for this situation: With `for_each()` it is possible to apply a function to all entries in a specified range. This allowed me to easily reorder the entries of all pairs in the relevant range of the list.

2 At the time of writing, it also occurred to me that this ordering could just be applied before inserting into the list.

4.4 HASHING METHODS

As for hashing functions, the following algorithms are implemented:

- DJB2 [RRM07]
- FNV-1a [FNV91]
- Simple-Hash [Eri04]
- Morton Bit Interleaving [BLD13]

The hash functions are used both when populating the grid with tetrahedra and when traversing it. Each method takes a position, grid layer, and number of hash misses as input to generate a table hash. The hash is then taken module the table size, obtain a hash key of appropriate size. Also, some of these methods were originally hash algorithms for strings, thus they were slightly adjusted.

The first three methods all utilize prime numbers to some degree. DJB2 utilizes bitshifts for fast multiplications by a constant factor, while FNV-1a uses mainly XOR operations. Simple-Hash just takes three prime numbers and multiplies each position components with one of them. Morton Bit Interleaving generates Morton codes by taking the integer coordinates of the cell that is to be hashed, shifting the bits of these integers, before applying XOR operations on them.

Note: The base implementation ported by Hermann Meißenhelter included DJB2 hashing as well as Morton Bit Interleaving. I implemented the remaining two hashing methods.

4.5 INTERSECTION TESTS

These intersection tests were implemented:

- SAT [Got96; GLM96]
- SAT-No-Edges
- SAT-MTV
- GJK [GJK88]

The SAT test is implemented in its most rudimentary form. The summary of the high-level steps is presented in algorithm 4.5. Basically all the relevant axes described in chapter 2.2.1 are queried and if a separation axis is found, we return that there is no intersection. If all axes have been checked and no separation axis was found, we return an intersection.

Algorithm 4.5 SAT Intersection Test

Input: two tetrahedra t_A, t_B
Output: whether t_A, t_B intersect

1: **if** face normals of t_A contain separation axis **then**
    return No Intersection
2: **end if**
3: **if** face normals of t_B contain separation axis **then**
    return No Intersection
4: **end if**
5: **if** edge cross-products from t_A and t_B contain separation axis **then**
    return No Intersection
6: **end if**
    return Intersection

Meanwhile, SAT-No-Edges omit the edge tests of the normal SAT test. Doing so causes the algorithm to find more false positives for collision, because we are skipping some relevant axes. The rationale was to compare the false positive rate with the saved computational time, to determine if such an intersection test could be useful in certain applications where accuracy might be less important.

On the other hand, the basis of the SAT-MTV is almost identical to the normal SAT test, except that during the tests of the separation axes, we also keep track of the current minimum penetration depth and the axis which this penetration occurred on. This method was included in order to offer at least one intersection test that would compute some kind of collision response, similar to what we would expect from a real use-case. However, the collision response is not applied to the simplices in the actual implementation, only calculated to see its effects on performance.

For my implementation of GJK, I used `libccd` [dan10], an commonly open-source collision detection library for the CPU that offers several intersection tests, and other resources [Win20] as foundation. I also considered implementing the EPA algorithm [VDB01; VDB03] to compute the MTV with GJK., but later decided against it. This was mainly due to time constraints, but another reason was that GJK had issues with accuracy, i.e. finding all collision pairs.

Note: The SAT (separating axis theorem) test was included with the base implementation ported by Hermann Meißenhelter. The other intersection tests were added by me.

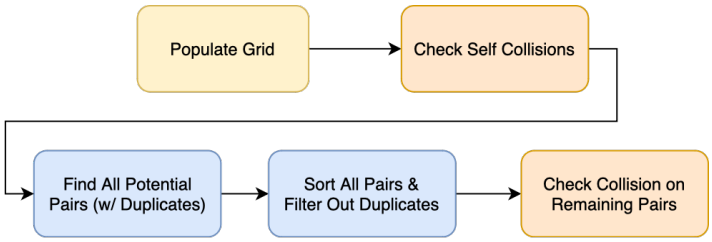


Figure 4.5: Collision Pipeline of new version of kDet with Self Collisions.

4.6 COLLISION PIPELINE

We saw the final collision pipeline for kDet already in figure 4.3. To summarize again: In the first step, we calculate the axis aligned bounding box for each of the collision objects. Then we check if the bounding boxes have any overlap. Only if they do, do we populate the grid and check for inter-object collisions. Otherwise, said steps are skipped and the program exits, since the objects cannot possibly intersect if their bounding boxes do not either.

If we also check for self-collisions, we need to populate the grid and check for self-collisions every frame, no matter if the bounding boxes of the geometries overlap or not. Furthermore, as previously discussed, with self-collisions we cannot use the acceleration trick when populating the grid. Said trick only inserted tetrahedra into the grid that were inside the bounding box overlap of the models. But when checking for self-collisions, we need to consider all tetrahedra of the scene, not only those inside the bounding box overlap area.

The adjusted pipeline with self-collisions is shown in figure 4.5. Note that the bounding box overlap test only determines whether to check for inter-object collisions.

Lastly, for the integration of kDet into a proper physics pipeline, it is recommended to execute as many of the pipeline stages as possible on the GPU. This saves the computational effort and time of copying data to and from the GPU.

5 BENCHMARKING SCENES

For the evaluation, I was suggested to run the presented algorithms against some benchmarks, ideally established ones. Surprisingly, animated scenes with tetrahedron meshes are hard to come by. SCI Utah, for example, offers their animation repository [Wal13], but their models use triangle meshes. An early idea was to take those animations and simply run a tetrahedron mesh generator on each frame. The problem is that this does not deliver desirable results because, for one, meshing would not be coherent over consequent frames. Second, certain kinds of animations (e.g. explosions) do not leave the mesh closed, making them unusable for (most) tetrahedron mesh generators. Therefore, I proceeded to create some scene benchmarks which we will proceed discuss in detail.

To offer other researchers a better starting point, I published all the created benchmark scenes as well as scene configurations & details in the kDet benchmark repository [MJ23]. Furthermore, I included all the data from benchmarking kDet as a point of reference for runtime comparisons.

5.1 CHOICE OF TOOLS & RESOURCES

5.1.1 *Simulation Software*

For the making of the animations, I required an animation program with a FEM solver (or comparable procedures, such as mass spring systems) to compute deformations with the internal mesh. Unfortunately, many common programs, such as Blender [BF23], do not (yet) offer such capabilities, The choice of software was mostly limited by this requirement. The two other important requirement were a) the ease of use, due to the amount of time that was left before thesis submission, and b) the option to export tetrahedron meshes each frame. The following were considered:

1. NVidia PhysX [NVI23]
2. SideFX: Houdini [Sid23]
3. SOFA FrameWork [Fau+12]

NVidia PhysX programming framework would have been a solid option, since it offers many capabilities aside from the necessary FEM solver. But given the remaining time, learning the framework and the necessary skills for this use-case seemed too much of a hurdle. This

was only consolidated by my inexperience with setting up C++ builds. Therefore, this option was discarded.

SideFx: Houdini was at first my animation software of choice, being an industry grade, feature rich animation program. *SideFx* even graciously offered me a key for their professional suite upon request for my research. Learning the software would have been manageable, because I had some prior practice with similar 3D modeling and animation program. On top of that, there exist lots of accessible material online which teaches how to create soft body animations using the *Houdini*'s FEM solver.

The reason I decided against using *SideFx: Houdini*, was that during my initial research I came to the conclusion that the program would not offer sufficient export options for the model geometries on a per frame basis. Later I came to find out however, that this was a research mistake on my behalf, because exporting geometries per frame was indeed possible. This was regrettably only realized by me within the last month before thesis submission, where I had no more time to create new scenes with the program.

Consequently, the choice fell on the *SOFA Framework*. *SOFA* offers various tools and solvers to simulate physical systems. The application accepts scene descriptions in `.scn`-format (which is basically `.xml`) that are fairly customizable.

In addition, scenes in *SOFA* can be written in `python` [1]. This would enable even more possibilities than with the `.scn`-format. However, the python scripts would not work with the Windows executable provided on their website. For that reason, I was restricted to using the `.xml`-style `.scn` scene declarations. *SOFA*'s documentation for these scene declaration file formats can be a bit lacking at times, but with the provided examples [Sof06] and a YouTube tutorial [SF21] explaining most of the components that were necessary for this use-case, I was able to construct the scenes for the animations.

Before we move on, a few remarks on the limitations of *SOFA*. First of all, *SOFA*'s solvers enforce a contact distance for the collision detection. This means that the models in the exported animations would never penetrate each other. But we have to consider that in real applications models possibly penetrate each other before force penalties are calculated and applied to correct for said penetration. Based on that, I corrected for this detail in the benchmarking by slightly translating the objects positions to have some penetration, in

1 Technically C++-scripts can be used as well, however this requires to build the program from source and on the programmer's end much more insight into the architecture of the *SOFA framework*

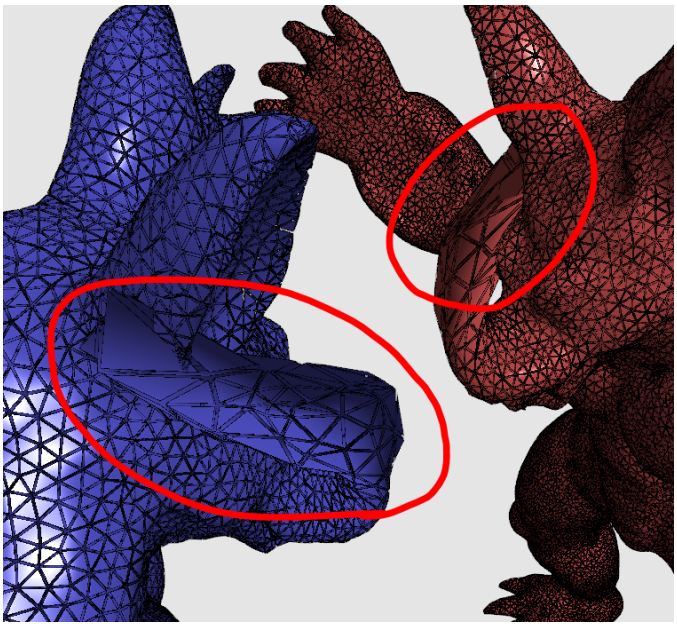


Figure 5.1: Since the animated scenes were not computed with self-collision detection, some frames have deformations where the models self penetrate.

order to obtain more realistic runtimes. We will discuss this later in more detail.

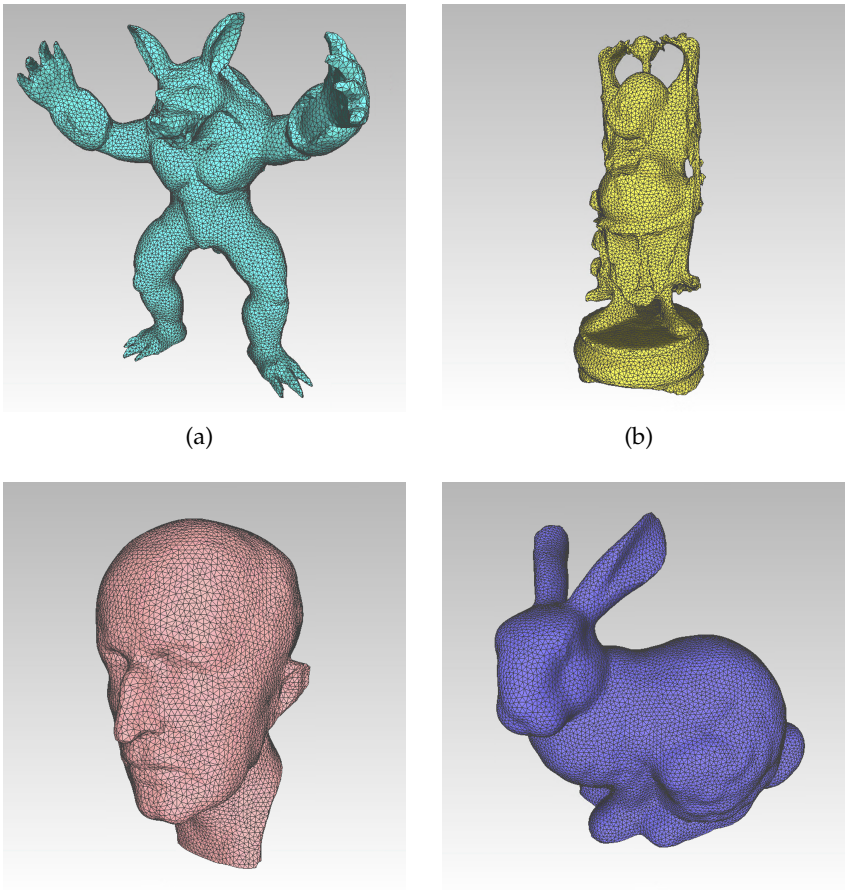
Additionally, *SOFA* did not calculate self-collisions. Their website occasionally mentions the option to check for them, but in my testing this did not work. Some frames therefore have the models penetrating themselves, for example the Armadillos' snouts penetrating their ears as shown in figure 5.1. An effort was made to minimize the occurrence of such frames.

Lastly, *SOFA* `.scn`- or `.xml`-scenes do not offer the proper capabilities to by themselves compute topological changes to the models such as cuts, breaks, or explosions. There exists an option to define a list of simplices that shall be added or removed at specific points of time during the simulation. But in light of the large number of simplices for our models, the effort required made this method simply unfeasible.

5.1.2 *Models*

The following models were sourced for the animations:

1. Stanford Armadillo [KL96] (figure 5.2a)
2. Stanford Happy Buddha [CL96] (figure 5.2b)
3. Max Planck head from MPI [Rus+] (figure 5.2c)
4. Stanford Bunny [TL94] (figure 5.2d)



(a) (b)

(c) (d)

Figure 5.2: Models used for benchmarking scenes (a) Stanford Armadillo (b) Stanford Happy Buddha (c) Max Planck Head from MPI (d) Stanford Bunny

model resolution	# tetrahedra
1	~10k
2	~20-30k
3	~44-57k
4	~85-100k

Table 5.1: Tetrahedron Counts for Different Model Resolutions

model \ resolution	1	2	3	4
Armadillo	10,121	20,052	49,218	83,804
Happy Buddha	10,433	28,584	49,616	87,158
Max Planck	10,093	21,240	43,648	92,387
Stanford Bunny	10,308	29,274	56,924	99,201

Table 5.2: Exact Tetrahedron Counts for Each Model and Resolution

Each model was first resized and prepared with Meshlab [Cig+08]. This included the covering of any holes, to make the models watertight, before removing any self intersecting triangles. To obtain the model in the various resolutions, Meshlab's *Surface Reconstruction: VGC* procedure was applied to the models with according parameters. Finally, the generation of the tetrahedron mesh was done using `tetgen` [Si15], with the `-pq` options.

As previously stated, each model is provided in different resolutions, four to be exact. Table 5.1 shows the approximate number of tetrahedra for each model resolution. Models with resolution one will have around 10k tetrahedra; roughly 20-30k for resolution two; between 44-57k for resolution three; and finally 85-100k for resolution four. Table 5.2 presents the exact number of tetrahedra per model for each of the four resolutions, while table 5.3 shows the number of vertices for the models.

5.2 SCENES

5.2.1 *Scenes created with SOFA*

With SOFA, I created the following three scenes using the FEM solver:

1. *Bunny & Planck*: a stiff model of Max Planck's head falling onto a deformable Stanford bunny
2. *Buddha & Armadillo*: a deformarble armadillo falling onto a non-deforming buddha model

model \ resolution	1	2	3	4
Armadillo	3,051	5,912	15,233	25,159
Happy Buddha	3,380	8,647	15,046	26,450
Max Planck	3,079	6,290	12,673	26,507
Stanford Bunny	3,272	8,969	17,205	29,536

Table 5.3: Exact Vertex Counts for Each Model and Resolution

3. *Armadillos*: two deformable armadillo models moving towards and colliding with each other

Figure 5.3 shows some screenshots for each of the listed animations.

Each scene contains two colliding models, though the forces, poses, and elastic properties of the models differ for each scene. The two model "restriction" is not due to kDet being unable to handle collisions between multiple objects, but only due to the implementation of our application.

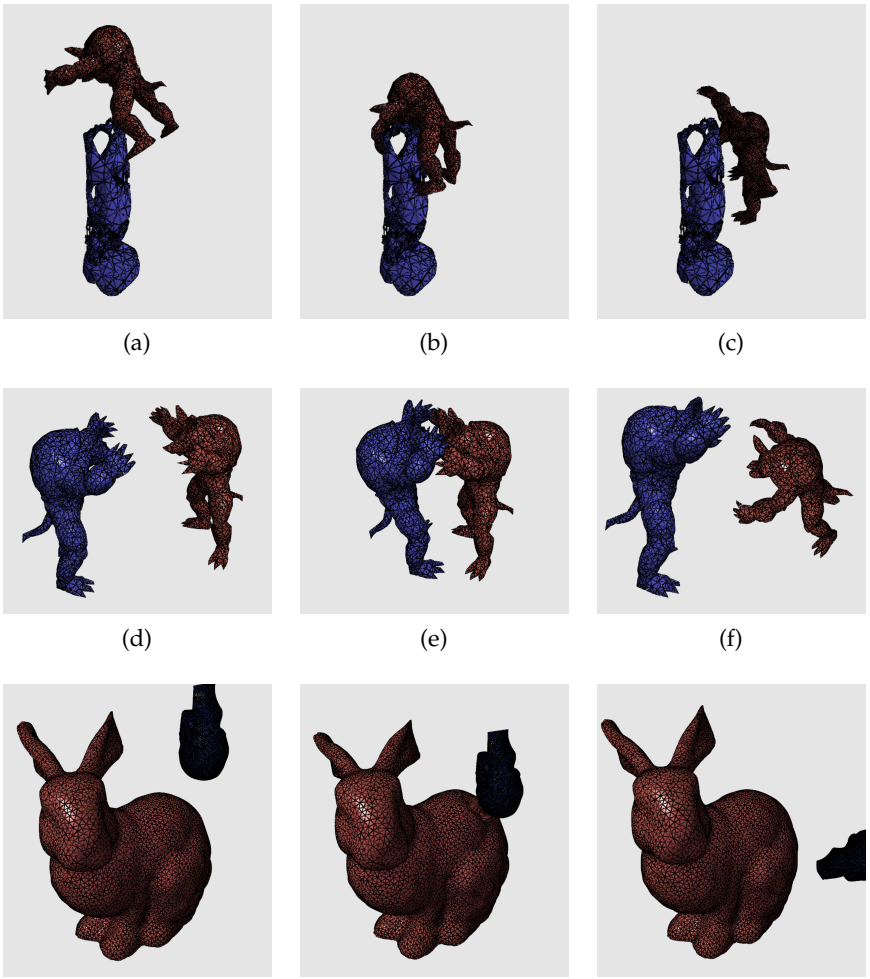
In addition, each scene was computed for each of the model resolutions. The goal was to keep the same animations consistent over the different model resolutions for better comparisons. However, while using the same scene configuration for the different resolutions, the parameters of the FEM solver had to be adjusted in each case. Thus, the same animation came out slightly different for each model resolution.

More info about the scene configurations and the FEM parameters is included together with the source code (see appendix A.2).

5.2.2 *Pass Through Armadillos*

As previously discussed, SOFA enforces a contact distance and already applies correction forces to each frame. This means, that models of a scene would usually not have any overlap or penetration. Even though we will later correct for this in the actual benchmarking, it might be of interest to observe runtimes for larger penetration depths. For that reason, I created this fourth scene of an armadillo phasing through another, as shown in figure 5.4. No force penalties or interactions were calculated or applied for this scene. This was done without SOFA, using a simple `python` script which took the initial starting position for one armadillo and applied a constant translation along the x-axis. The other armadillo remains stationary.

The scene is only available with the armadillos in resolution 2 (~20k tetrahedra), because of the concern that for higher resolutions either the GPU memory or the vector holding the potential pairs might overflow.

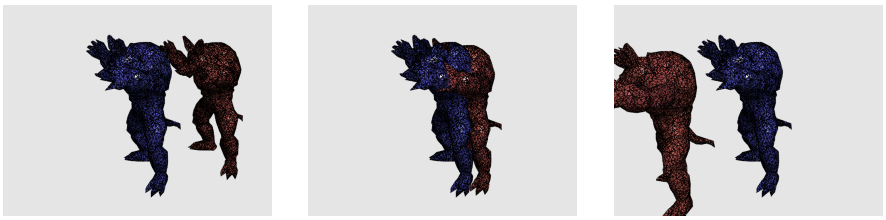


(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 5.3: Screenshots of the Benchmarking Scenes created with SOFA (order is left to right). (a)-(c) Flexible Stanford Armadillo falling on rigid Happy Buddha model. (d) - (f) Two deformable Armadillos accelerating towards each other. (g)-(i) Head of Max Planck falling onto Stanford Bunny, with both being deformable.



(a) (b) (c)

Figure 5.4: Screenshots of the Pass Through Armadillos Scene (left to right)

5.3 SHORTCOMINGS

The created benchmarks unfortunately exhibit several flaws which need to be acknowledged before moving to the evaluation.

The major problem of this benchmarking suite is that the scenes are too similar overall. Even though each scene uses different models and material properties, all scenes are essentially just two deformable objects moving towards each other and colliding for brief moments of time.

Ideally the benchmark suite would cover various scenarios to help determine challenging as well as favorable conditions for the algorithm. Among other things, one could include scenes with changing topologies, like tears, breaks, or explosions. Especially the former two occur frequently in material simulations. To that one could add scenes where models have large contact areas, such as cloth simulations, or scenes with higher penetration depth. For the former, I actually tried to create a cloth simulation benchmark, but it turned out unsuccessful because the modeled cloth just phased through the sphere, instead of colliding and deforming.

Finally, for future benchmarks the use of tetrahedron meshes that contain tetrahedra of varying sizes might be interesting. To my knowledge, software such as Gmsh [**gmsh**] would allow for creating meshes with such constraints.

6 EVALUATION

For the benchmarking, I ran and compared the following four methods:

- *kDet-Thrust*: The new version of kDet that I worked on. The name was chosen, because the list that stores all the potential pairs uses CUDA's `thrust` device vectors. This version find all collision pairs.
- *kDet-No-Pairs*: The old version of kDet. This version does not find all collision pairs, but only all collision simplices, hence the name.
- *GPU-BF-Loop*: GPU brute-force method that launches a thread for each tetrahedron of the first object and checks against all tetrahedra of the second object in a loop. This method finds all collision pairs.
- *GPU-BF-No Loop*: GPU brute-force method that launches a thread for each possible pair of tetrahedra to be checked. This method finds all collision pairs.

Every presented method includes a bounding box pre-check, similar to kDet-Thrust. This means, that if no bounding box overlap of the collision objects is detected, then inter-object collisions are skipped; self-collisions might still be computed. We can observe the effects of this in figure 6.1: The huge jumps in runtime indicate where bounding boxes of the collision objects overlap, and therefore inter-object collisions were computed.

All scenes, except for the pass through armadillos, were benchmarked with and without self-collisions. Benchmarking results with self-collisions consequently never include data for the pass through armadillos scene. Also please keep in mind, that when not explicitly mentioned, a benchmark was executed without self-collisions.

The benchmarks were run on a 64-Bit Windows 10 system with an Intel(R) Core(TM) i7-7800X with six cores and 3.50 GHz, 64 GB of RAM, and an NVIDIA GForce RTX 3080 with 10 GB of GPU memory.

6.1 SELECTING BENCHMARKING PARAMETERS

Before running the "actual" benchmarks, I ran some tests to determine favorable parameters (e.g. CUDA block size, choice of hashing method). We will consequently go through the results of these tests as well as some measures that I needed to employ for the benchmarking.

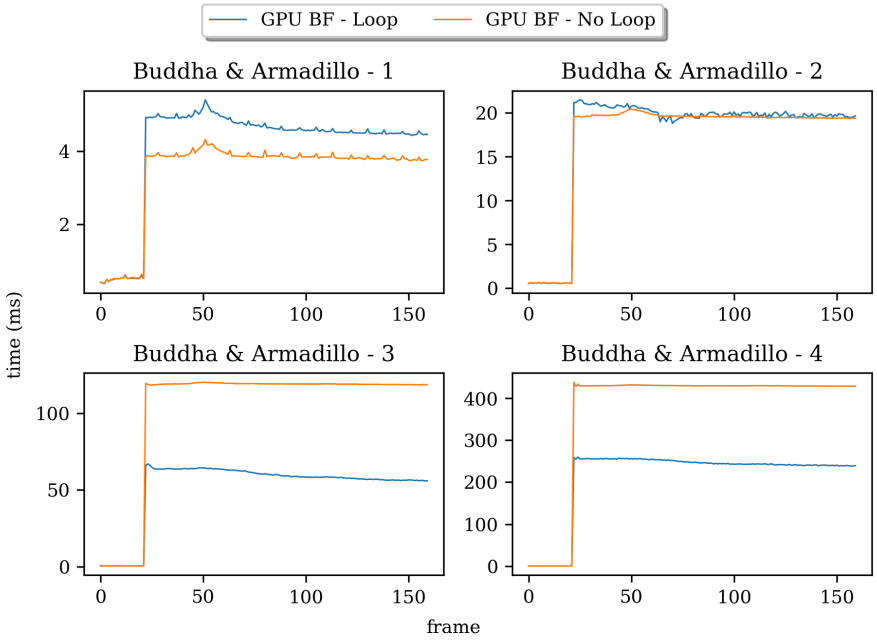


Figure 6.1: The plots show sample frame times for the GPU brute-force methods. Note the sudden jump in frame times at certain positions: This indicates whether the bounding box pre-check passed and let CD be computed or not.

Animation	Model	Translation
Bunny & Planck	Planck	(0, 0.1, 0)
Buddha & Armadillo	Armadillo	(0, -0.1, 0)
Armadillos	Armadillo B	(-0.1, 0, 0)

Table 6.1: Applied translations on the models of each respective scene to create overlap during CD.

6.1.1 *Correcting for SOFA Contact Distance*

As previously eluded to, the objects in the animated scenes do not have any collisions or penetration, because of the enforced contact distance by the *SOFA framework*. This could possibly distort runtimes and does not accurately depict actual use-cases, where we would expect some overlap [1]. For that reason, the decision was made to slightly translate the position of one object per scene. Table 6.1 lists which objects were translated for each scene and the translation vector in world coordinates for every respective case.

1 Of course, after the collision detection is finished and force penalties have been applied, the overlap should be resolved.

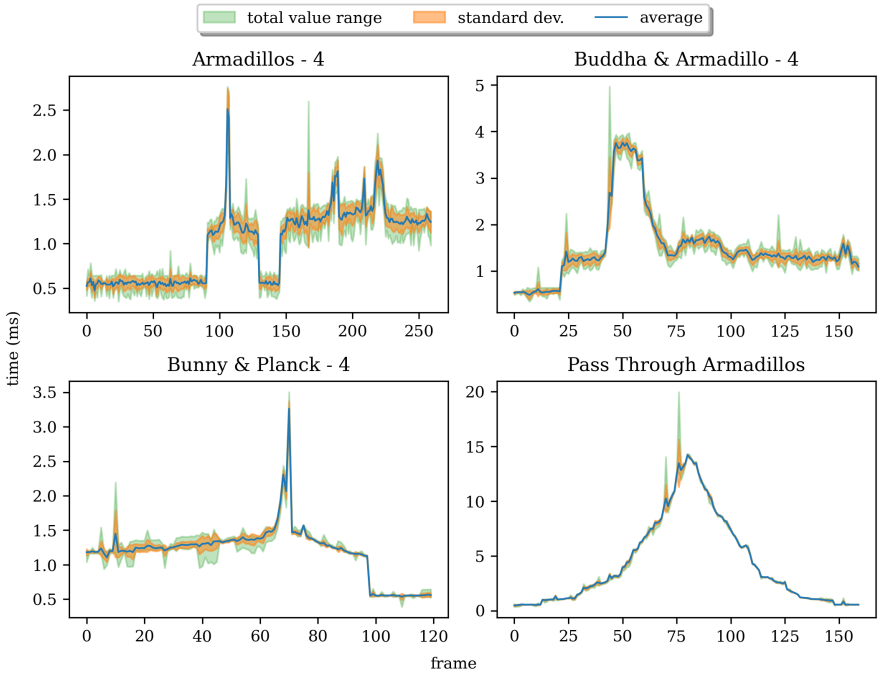


Figure 6.2: Test of Average Frame Times with kDet-Thrust: Total range of values per frame is green; Standard deviation per frame is given in orange.

6.1.2 *Use of 10 Run Averages*

To minimize variance for the runtimes, I decided to run every benchmark ten times and average over the results for each frame. For this to be a good measure however, it required that the averaged runtime does not drastically differ from the runtime of individual runs. To inspect whether this is the case, I plotted for each frame the averaged runtime, the range of total values, and the standard deviation for all scenes in the highest resolution with kDet-Thrust and kDet-No-Pairs. When inspecting figure 6.2 and 6.3, we can see that the standard deviation usually stays below 1/5 th to 1/4 th of a millisecond. There are some occasional spikes in the total range of values, but in most cases these are only in the order of a few milliseconds. A notable spike was recorded for kDet-Thrust and the pass through armadillos near frame 80.

Overall the averaged runtime seems to depict the general runtime of the algorithm decently. The same can be said for the GPU brute-force methods as well as for the kDet methods with self-collisions. Although it should be mentioned, that the standard deviation per frame was higher, yet roughly scaled to the runtimes of each method.

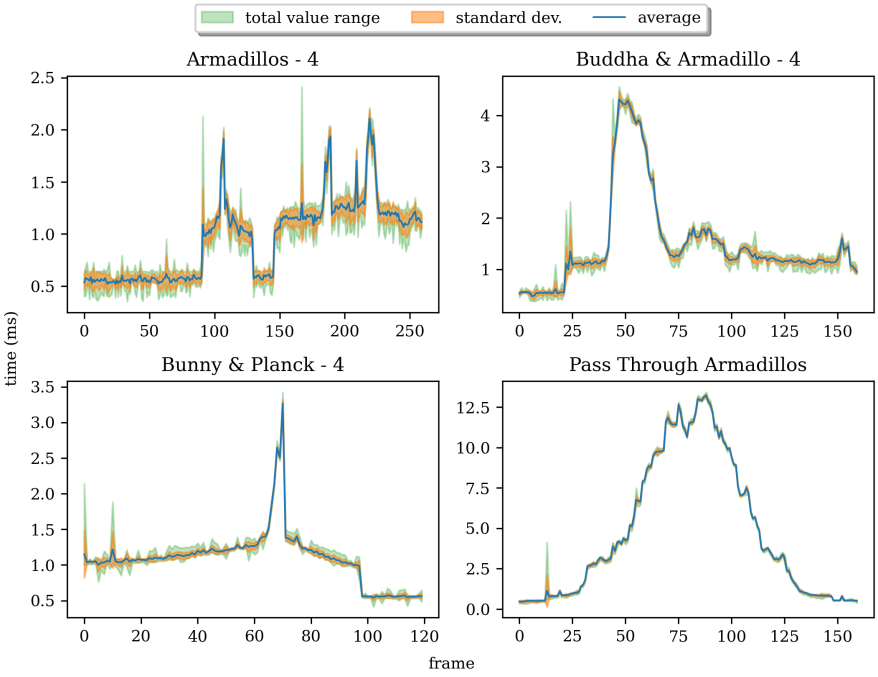


Figure 6.3: Test of Average Frame Times with kDet-No-Pairs: Total range of values per frame is green; Standard deviation per frame is given in orange.

Based on these results, all graphs from here on out will depict averaged runtimes. However, the raw data for all runs is included with the source code (see appendix, A.2).

6.1.3 *Host Time vs Device Time*

The benchmarking recorded both host (CPU) and device (GPU) times. Since kDet runs mostly on the GPU, our main interest lies in the device times. Beforehand, let us confirm that host and device times do not differ by a significant amount. We expect only a few ms difference per frame, caused by overhead to start the GPU kernels and some operations that are carried out by the CPU.

Figure 6.4 depicts host and device times for kDet-Thrust in all scenes on highest resolution. We notice a high host runtime on the first frame of the animation, as well as the frame where inter-object collisions seem to be checked for the first time. These are artifacts of an implementation detail, being the initialization of the grid and other data structures for kDet. In an actual implementation these can all be done as pre-computation. Other than that, host and device times seem to follow the stated expectations, in that they only vary by a small and seemingly set amount per frame.

A small note that I would like to add, is that from what I can deduct, the computation times for the initialization of the grid do not seem to

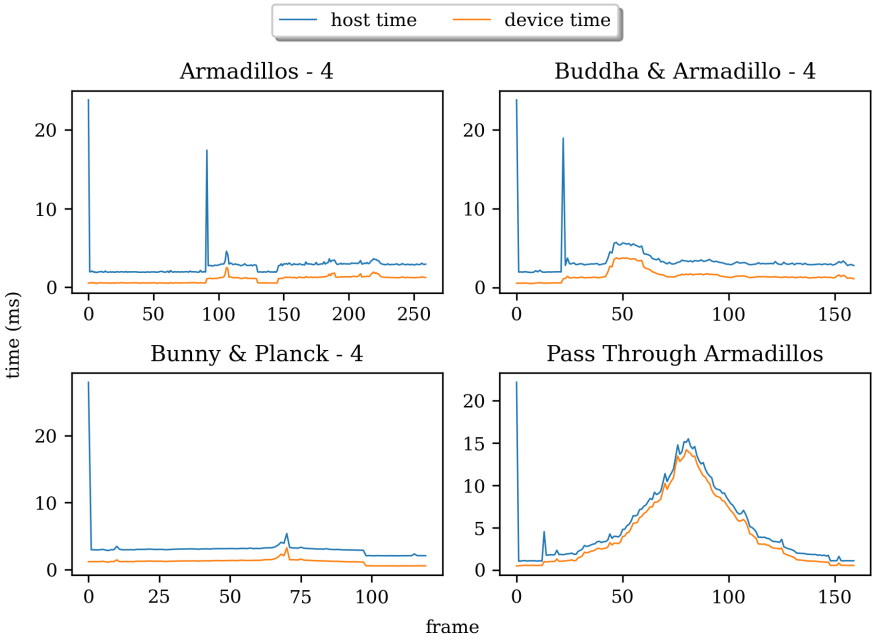


Figure 6.4: Host vs Device Time with kDet-Thrust: The spike in host times occur when data-structures on the GPU are initialized. The initaliziations could just be done as pre-computation, this was an oversight on my part.

exceed 100 ms in the benchmarking data.

All reported runtimes henceforth are device runtimes.

6.1.4 *CUDA Block Size*

Next, I intended to explore what effect different block sizes of the GPU kernels would have on runtime. CUDA block sizes reflect how many threads are in a single thread block of the GPU kernel grid. As shown in figure 6.5, according to the NSIGHT Compute statistics for the RTX 3080, powers of two seem to offer the best warp occupancy. Thus, I decided to test block sizes of 16, 32, 64, 128, and 256 with kDet-Thrust for each scene on the highest resolution. Block sizes higher than 256 could not be tested because the intersections tests would not execute.

Figure 6.6a presents the sums of the total runtimes of all benchmarked animations for each block size. Figure 6.6b does the same for kDet-Thrust with self-collisions. Overall, we cannot observe significant difference between the various block sizes. Overall, we only see that that block sizes of 16 and 256 perform (slightly) slower than the rest. These results extended to the kDet-No-Pairs method as well.

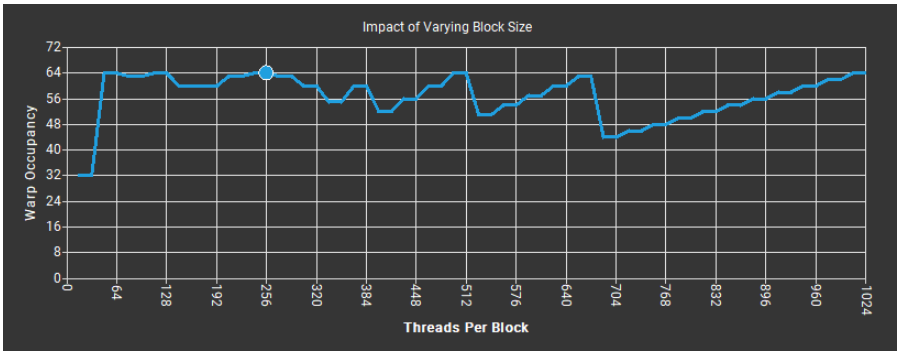
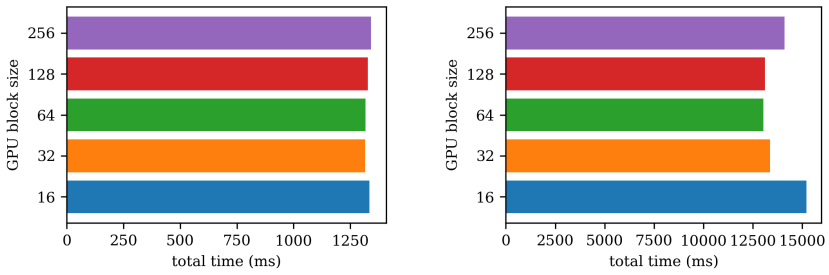
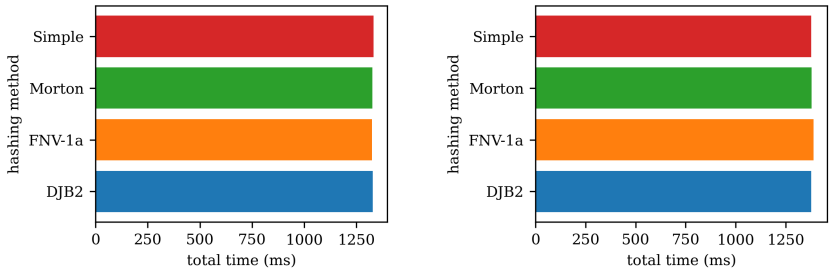


Figure 6.5: Warp Occupancy for different GPU Block Sizes for a NVIDIA RTX 3080 (source: NSIGHT compute)



(a) with self-collisions (b) without self-collisions

Figure 6.6: Summation of runtimes for all resolution 4 animations with different GPU block-sizes for kDet-Thrust without self-collisions (a) and with self-collisions (b).



(a) kDet-Thrust (b) kDet-No-Pairs

Figure 6.7: Total summed runtimes over all resolution 4 scenes for different hashing methods with kDet-Thrust (a) and kDet-No-Pairs (b)

Based on these results, I picked a block size of 128 for all following benchmarks [2].

6.1.5 *Hashing Methods*

To see if the hashing method had an effect on the runtime, we can first look at figure 6.7a which displays the total runtimes of all benchmarked animations for each hashing method executed with kDet-Thrust. Figure 6.7b presents the same info but for kDet-No-Pairs. Both graphs tell the same story, being that the hashing method does not have any noticeable impact on the runtime. Presumably, the number of hashed entries in contrast to the hash table size is too small to observe any meaningful differences in hash collisions or other characteristics of the hash functions, that could affect the runtime. Table 6.2 displays the total number of tetrahedra for the resolution 4 scenes and the corresponding occupancy of the spatial grid in memory. The occupancy is calculated by dividing the number of tetrahedra per scene by the total available entries in the hash table of the spatial grid. The hash table has a hard-coded size of roughly ~3.8×10^6, with every bucket having 128 entries, resulting in 480 million entries overall. Because multiple tetrahedra can get hashed into the same bucket, it is reasonable to argue that a lower percentage of the available buckets is occupied than total entries in the hash table. Despite the bucket occupancy being dependent on the object poses, the general point holds and underlines our assumption.

From here onwards, benchmarks for kDet-Thrust and kDet-No-Pairs all use *DJB2* hashing [3].

2 It should be noted, that the benchmarks from the previous sections were also executed with a block size of 128.

3 *DJB2* hashing was also used for all benchmarks in the previous chapters.

scene	total #tetrahedra	grid occupancy
Armadillos - 4	167,608	0.03%
Bunny & Planck - 4	191,588	0.04%
Buddha & Armadillo - 4	170,962	0.04%

Table 6.2: Hash Grid Occupancy for Benchmarking Scenes; observe the low utilization of the available hash cells.

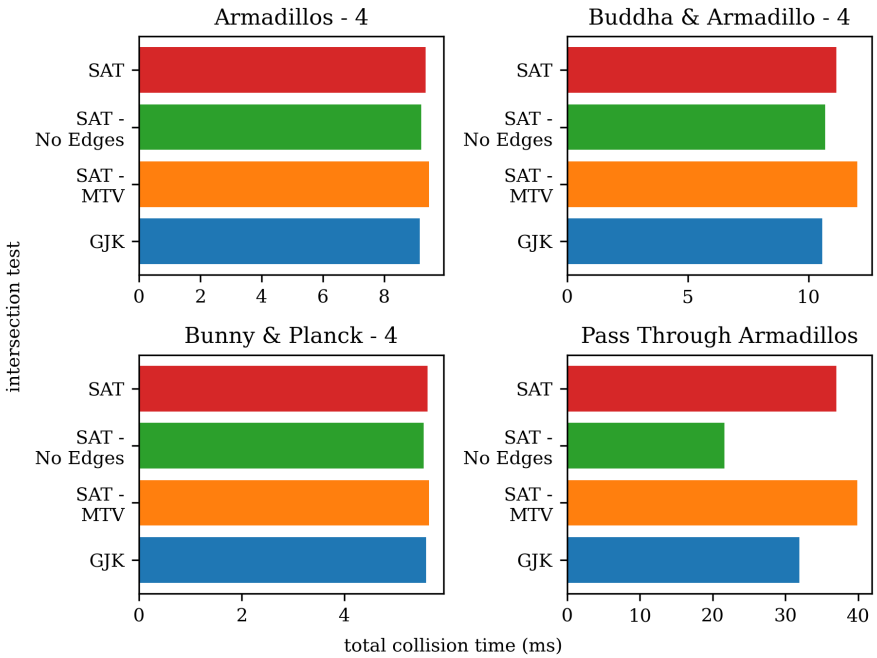
6.1.6 *Intersection Tests*

The last parameter that was tested for, were the different intersection tests. I compared runtimes and detected collision pairs for the implemented intersection tests from chapter 4.5, which were:

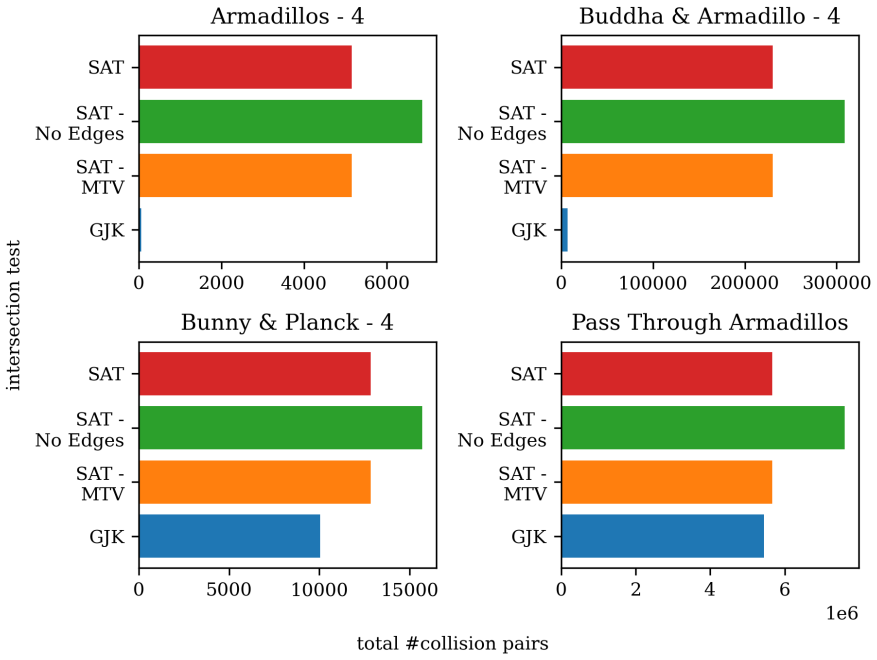
- *SAT*: traditional SAT intersection test
- *SAT - No Edges*: SAT test without edge-on-edge tests
- *SAT-MTV*: SAT that also computes the MTV
- *GJK*: Gilbert-Johnson-Keerthi algorithm

First things first, figure 6.8a shows the total runtime of the intersection tests for each animation (in highest resolution) when benchmarked with kDet-Thrust. Please note, that these are not the total frame times, but only the time that was spent in the GPU kernel for intersection tests. *SAT - No Edges* and *GJK* run slightly faster for most of the scenes, except for the pass through armadillos where they considerably pull ahead of the other methods. Here, *SAT - No Edges* is almost 50% faster than the normal *SAT* test. Since the pass through armadillos are the scene where the most penetration occurs, as the collision objects literally pass through each other, differences in the runtimes of the intersection tests manifest here most noticeably. On the other hand, the *SAT-MTV* is only slightly slower than the standard *SAT* test in all scenes.

Let us now contrast that with the total number of collision pairs detected for each animation with every intersection test, as presented in figure 6.8b. The *SAT* and *SAT-MTV* always find the same number of collision pairs. However, *SAT - No Edges* will find between 15 - 30 % additional collision pairs, which is quite significant. *GJK* on the other hand, finds almost no pairs for some of the scenes, and misses a noteable amount in the others. To further investigate this, I plotted the collision pairs of *GJK* and *SAT* for all resolutions of the Armadillos scene in figure 6.9. We observe that for resolutions 1 & 2, *GJK* still finds the majority of the collision pairs (even though it still misses a considerable number of them), before collapsing with resolutions 3 &



(a) total runtimes



(b) total collision pairs

Figure 6.8: Total summed runtimes and collision pairs for all resolution 4 scenes with kDet and different intersection tests

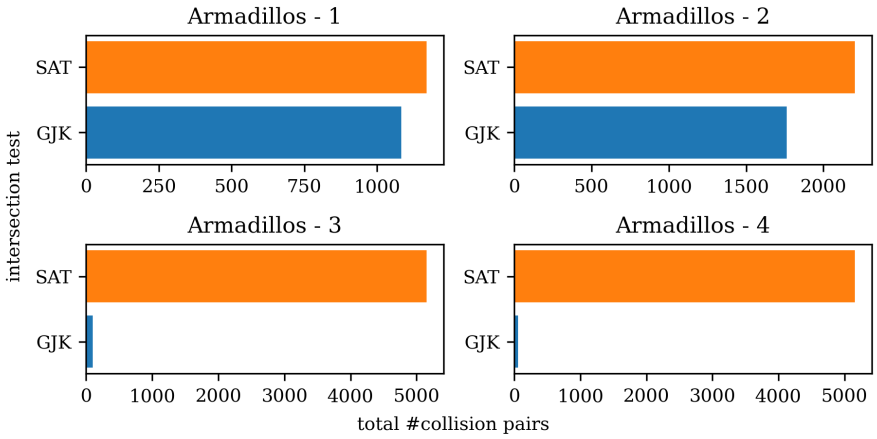


Figure 6.9: Comparison of Detected collision pairs between SAT and GJK in Armadillo scenes. GJK cannot properly detect collision pairs from resolution 2 onwards.

4. I suppose that my implementation of *GJK* cannot handle simplices below a certain size. Since the different resolutions of the scenes are the same scale, it means that the simplices of the meshes have to become smaller for higher resolutions. Perhaps this issue could also be attributed to the numerical inrobustness of the algorithm. This remains speculation on my behalf though.

From this selection, only the *SAT* and *SAT-MTV* are reasonably accurate, even if they are slower in certain scenes. Accuracy has to be valued higher here, because as we will soon see, intersection tests do not take a considerable amount of time per frame with kDet-Thrust. Hence my choice of the *SAT* test for all benchmarks moving forward [4].

6.2 GPU BRUTE-FORCE VS KDET

A keen reader might have noticed, that for the choice of parameters the GPU brute-force methods were only sparingly mentioned. The reason being, that for scenes with higher resolutions the brute-force methods run several orders of magnitude slower than both versions of kDet. No choice of parameters would meaningfully change this outcome (see appendix.

Figure 6.10 shows the runtime per frame for the pass through armadillos with all our collision methods. The armadillos in this scene are only of resolution two, which explains the "mere" 10-15 ms runtime difference per frame. The rise and fall of the runtimes can be explained with the increasing and decreasing penetration depth of

4 The *SAT* was also used for all benchmarks in the previous chapters.

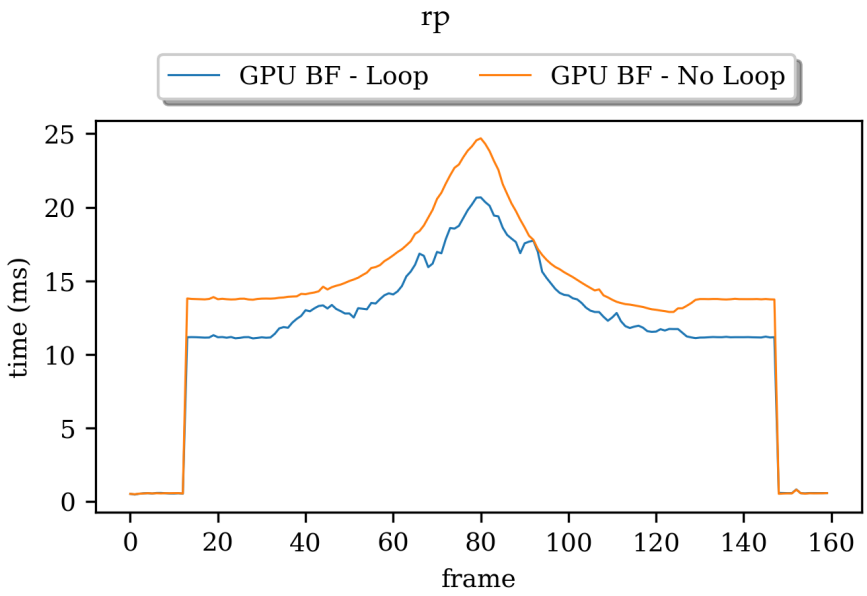


Figure 6.10: Runtime comparison between kDet and GPU brute-force methods in Pass-Through-Armadillo scene

the armadillos as they phase through each other (we will discuss this in detail in chapter 6.3.1). Notice, that as soon the bounding box test passes and inter-object collisions are calculated, the brute-force methods already have quite the jump in runtimes to about 12 ms, while the kDet methods seem to scale with the penetration depth.

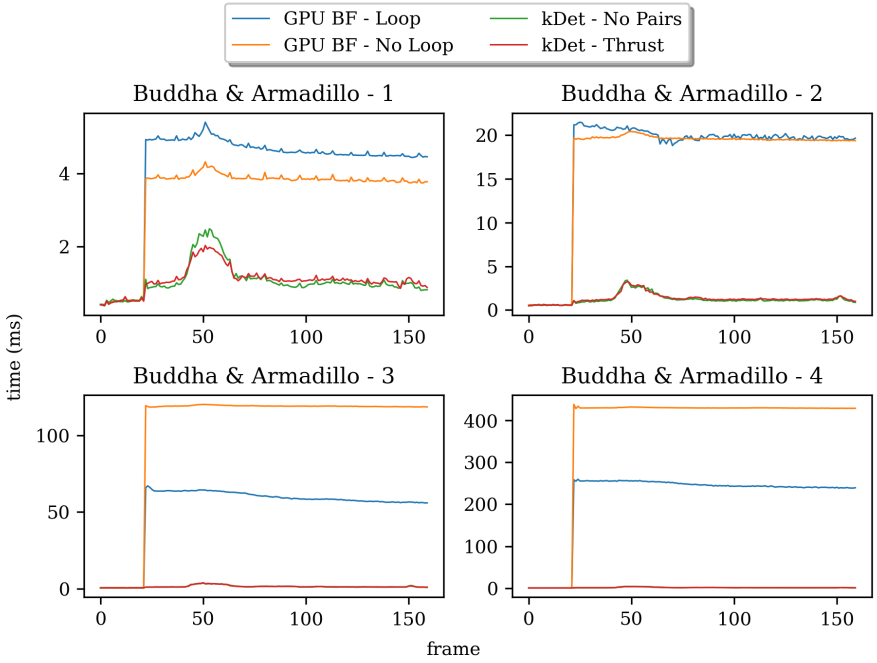
When we move to figure 6.11, we can deduct what was eluded to earlier. For scenes with smaller resolutions, the runtimes of brute-force methods are again slower, but still somewhat comparable to that of kDet as they are only differ a few ms. With increasing scene resolution however, the runtimes of kDet are dwarfed in contrast, as they basically appear flat for resolutions 3 and 4. The brute-force methods need several hundred ms, while the kDet methods are in the single digits. Results for all other scenes painted the same picture, with and without self-collisions.

Based on these results, I will only compare kDet-Thrust against kDet-No-Pairs from here on.

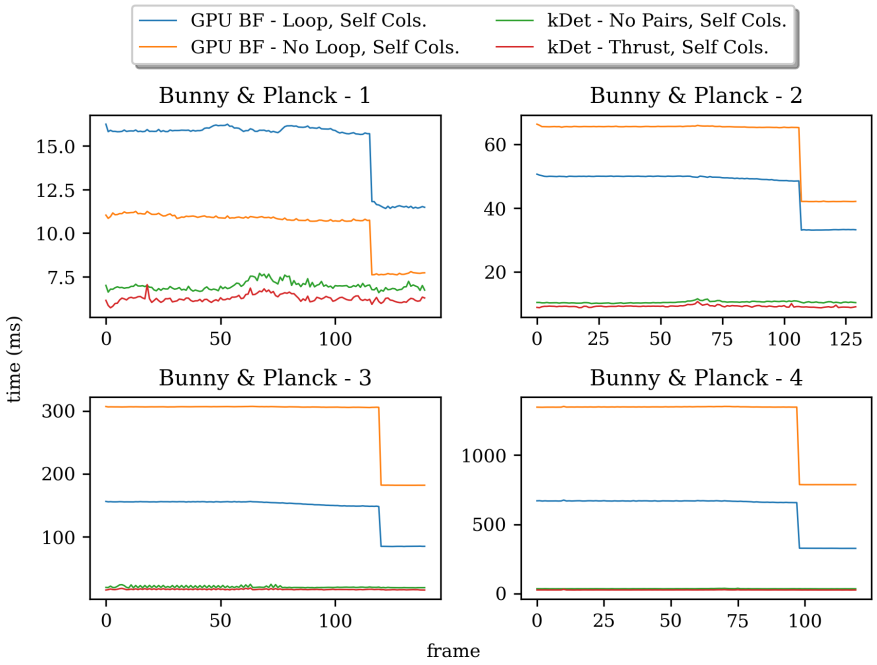
6.3 KDET

6.3.1 *Pass Through Armadillos*

The pass through armadillo scene was created to measure kDet's behavior for different penetration depths. Since calculating the actual penetration would have been rather cumbersome, I instead use an



(a) without self-collisions



(b) with self-collisions

Figure 6.11: Runtime comparison between kDet and GPU brute-force methods: (a) Buddha & Armadillo scenes with self-collisions, and (b) Bunny & Planck scenes with self collisions

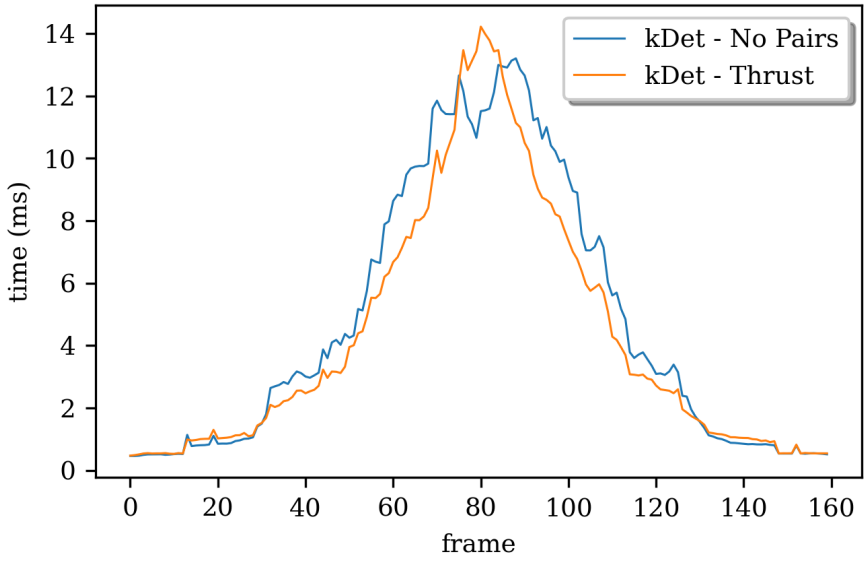
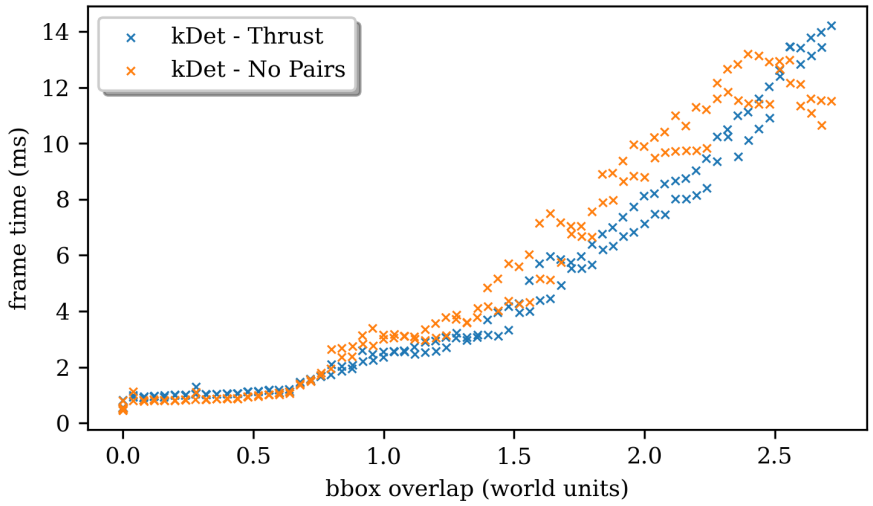


Figure 6.12: Pass-Through-Armadillo frame times with kDet

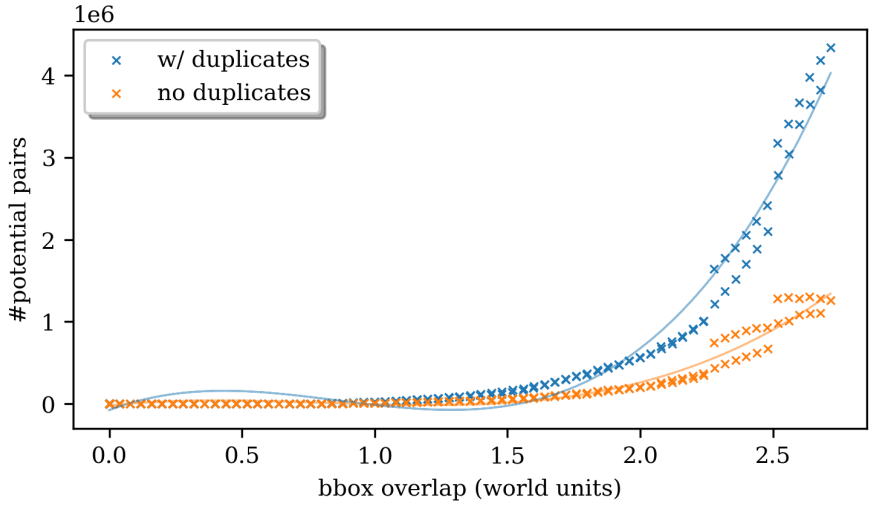
approximation with the bounding box overlap of the two collision objects along the x-axis. Because the objects move only along said axis, this seems like an acceptable estimation. When mentioning to penetration depth in this chapter, I will be referring to this measure of bounding box overlap.

Let us examine first the runtimes per frame with figure 6.12. We see that kDet-Thrust and kDet-No-Pairs show overall similar frame times, with kDet-Thrust having moderately slower runtimes at the peak, but being slightly faster during the rest. What is more, we can roughly see how the runtime for the increasing and decreasing penetration evolves. To put that into better perspective, figure 6.13a plots the runtime against the penetration depth. For one, the former claim seems to be supported by the graph. On the other hand, we can better see how the runtimes develop for the increasing penetration depth; starting at around 1-2 ms and gradually building up to 13-14 ms at the top. Notice that for deep penetrations, the runtime is several times higher than for shallow ones. This hints to us that detecting self-collisions has to be regarded as basically one of the worst case scenarios [5], thus resulting in much higher runtimes. Checking the object against itself can be regarded as practically checking against another object at 100% penetration, where the number of potential collision pairs per tetrahedron and overall become meaningfully larger than it would be for shallow penetrations. Actually, what should be kept in mind, is that the number of potential collision pairs in proportion to the penetration depth increases cubically. This can be

5 The statement does not mean to imply that self-collisions are the definitive worst case scenario for kDet in general, only for the scenes that we look at in this work.

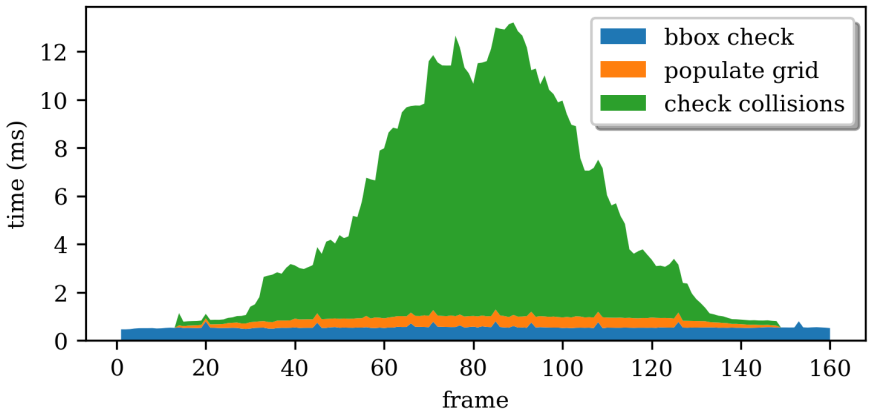


(a) kDet-No-Pairs

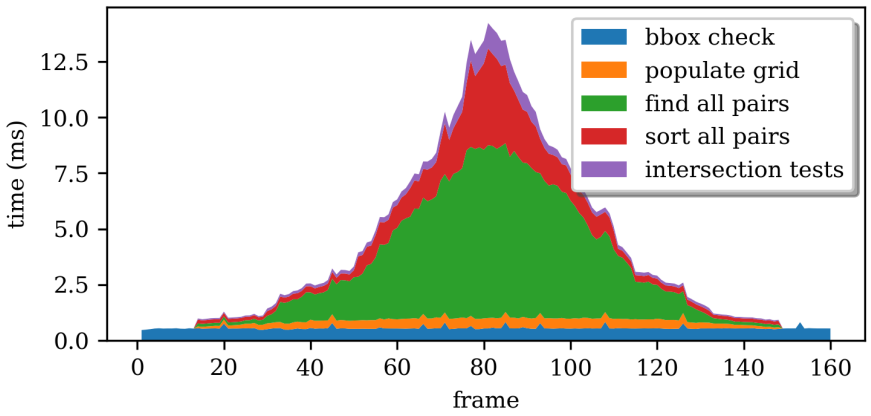


(b) kDet-Thrust

Figure 6.13: (a) Frame time compared to penetration depth approximated by bounding box overlap. (b) Number of potential pairs plotted against penetration depth.



(a)



(b)

Figure 6.14: Stack plots for runtimes in Pass-Through-Armadillos Scene with (a) kDet-Thrust and (b) kDet-No-Pairs

easily explained by considering that the intersection volume grows by the same measure in relation to the penetration, and with that also the number of tetrahedra contained within that space. Figure 6.13b depicts this relationship. A trendline for a degree three polynomial was fitted onto the data set.

Lastly, let we will investigate how much each step of the collision pipeline takes per frame for the respective collision method. Looking at figures 6.14b we see this subdivision of the runtime for *kDet - No Pair*. The bounding box test and grid populating take little time overall, around 1 ms, regardless of the penetration depth. Traversing the grid and checking for collisions take most of the time; near full penetration almost ten times as much as the other two steps, meaning between 10-12 ms.

Figure 6.14a shows a similar graph for kDet-Thrust. Populating the grid together with the bounding box test are identical to *kDet - No Pairs*, therefore they take the same amount of time. Moreover, notice that the intersection tests only demand little runtime, even at high penetration; usually less than 1-1.5 ms. What we see instead, is that for increasing penetration depth, traversing the grid to find all potential collision pairs and sorting them afterwards take majority of the runtime, combining together for around 10-12 ms at maximum penetration. These two steps of the collision pipeline are the areas that should be improved upon in future iterations of kDet.

6.3.2 *Remaining Scenes*

After the considerations from the previous chapter, we can now explore all the benchmarking scenes that were created with the *SOFA framework*. Note, that from here on, when referring to "all scenes", I exclude the pass through armadillos scene.

Figure 6.15 pits the runtimes per frame of the two kDet methods against each other in all scenes. In that graph we make out comparable runtimes for kDet-Thrust and kDet-No-Pairs. This means that kDet-Thrust does the computation of finding all unique collision pairs at basically no discernible cost in runtime, when comparing to the old method. For the armadillo scenes, we observe frame times of less than 3 ms; for the buddha & armadillo scene frame times stay beneath 4.5 ms; and for the bunny & planck scene the kDet methods top out at roughly 3.5 ms.

Once we look at frame times with self-collisions, as presented in figure 6.16, we even see that kDet-Thrust runs faster. The runtimes separate further and further with increasing scene resolutions, reaching differences of up to 10-15 ms per frame for resolution 4. Looking at all scenes, kDet-Thrust maxes out at around 30 ms per frame. For lower resolution scenes it even stays comfortably under that margin.

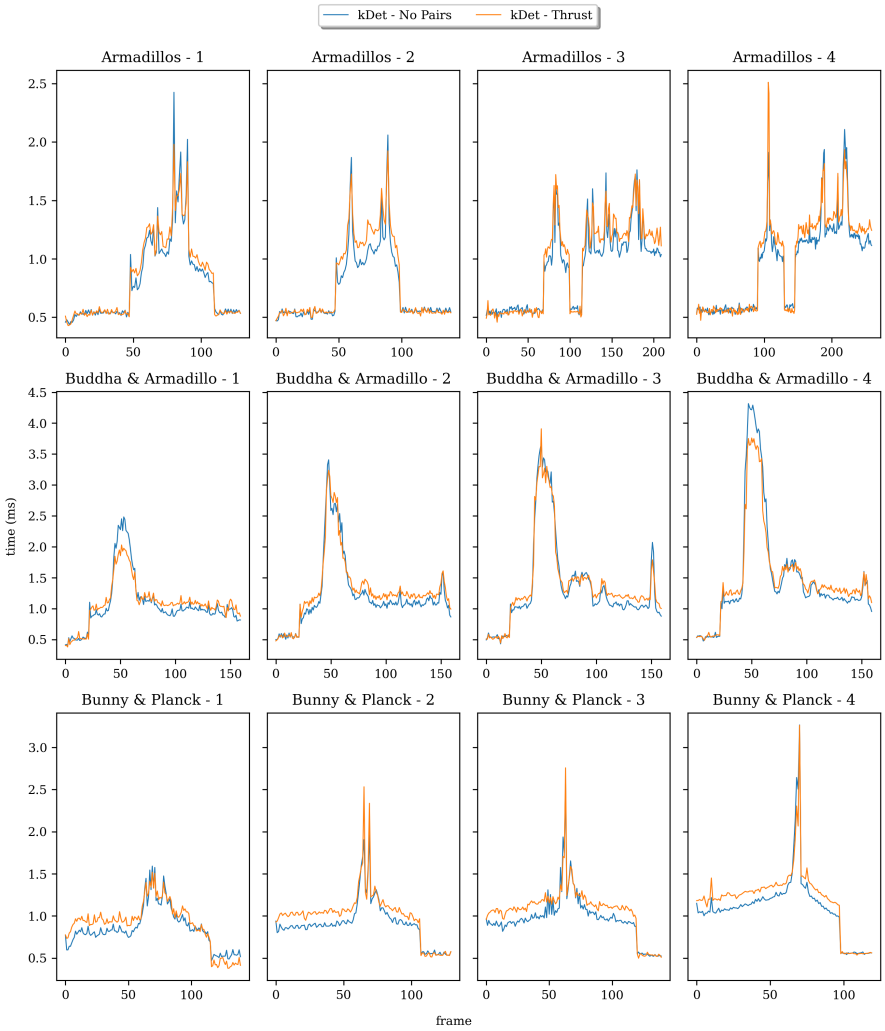


Figure 6.15: Runtimes for kDet methods in the benchmarking scenes with varying resolutions

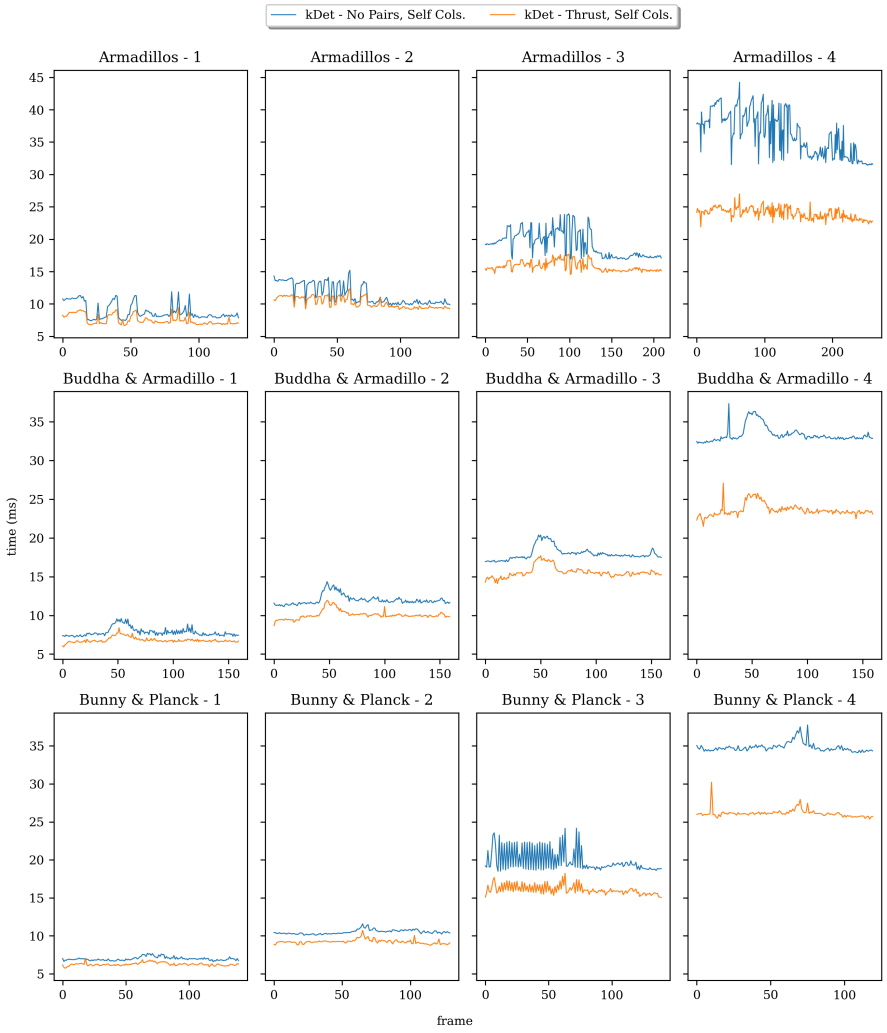


Figure 6.16: Runtimes for kDet methods in the benchmarking scenes with varying resolutions with self-collisions

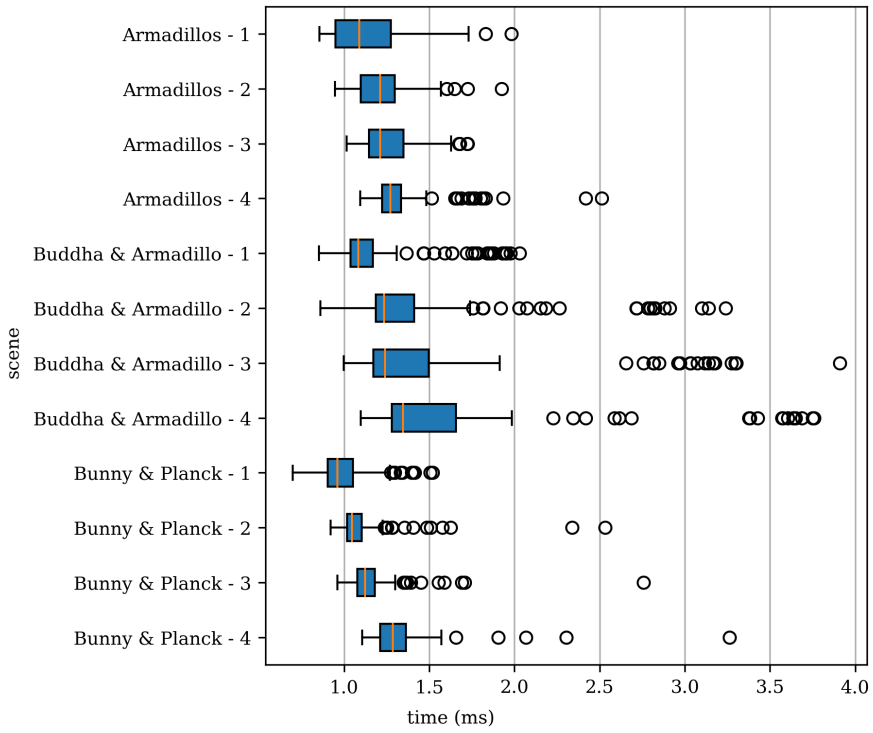


Figure 6.17: Box plot of frame times for kDet-Thrust in the benchmarking scenes with varying resolutions

The old method on the other hand, needs upwards of 45 ms for some of the resolution 4 scenes.

Also notice, that for the benchmarks without self-collisions, frame times for the varying scene resolutions do not seem to differ by much. Whereas with self-collisions, we see a clear development of the frame times. To further investigate these frame times, I created a box plot with the frame times of each scene benchmarked with kDet-Thrust and self-collisions, as presented in figure 6.17. Again, we see a clear progression in the median frame time (orange line within each box) for the increasing resolutions. Starting at around 6-7 ms for resolution 1 scenes; then going to 9-10 ms for resolution 2 scenes; 15-16 ms with resolution 3; and finally between 23-26 ms for scenes of resolution 4. Figure 6.18 depicts the same for kDet-No-Pairs with self-collisions. We see a similar development, though the median frame times for the scenes are higher: 7-8 ms for resolution 1; 10-12 ms for resolution 2; 17-19 ms for resolution 3; and 33-36 ms for resolution 4 scenes.

Previously, we were also interested in the difference in runtime between collision detection with and without self-collisions. Figure 6.19 plots the frame times for kDet-Thrust with and without self collision. self-collisions have a noticeable impact on the runtime, further increasing with the scene resolution. In scenes with resolution 4 we

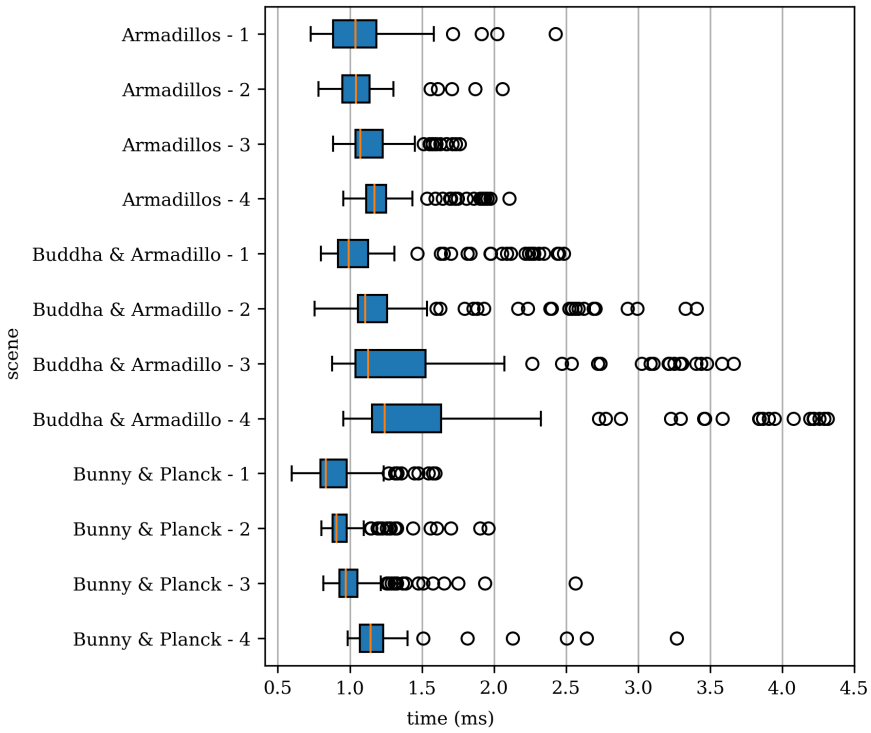


Figure 6.18: Box plot of frame times for kDet-No-Pairs in the benchmarking scenes with varying resolutions

see almost 20-25 ms differences in runtime. Keep in mind that kDet-Thrust with self-collisions topped out at around 30 ms in those scenes, and generally stayed around 23-26 ms. When we look at the runtime allocations for those scenes with self-collisions, as shown in figure 6.20, we see that almost all of the runtime is occupied by the self-collisions. I made comparable observations for kDet-No-Pairs.

Next, to roughly gauge the overall differences in runtime, I summed the frame times over all scenes for each kDet method, with and without self-collisions respectively. The results can be viewed in figure 6.21. Without self-collisions, kDet-Thrust needed about 2,100 ms for collision detection on all scenes, whereas kDet-No-Pairs only needed about 2,000 ms. Consequently, without self-collisions kDet-No-Pairs is roughly 5.3% faster than my method.

Yet with self-collisions, kDet-No-Pairs was the slower method, needing around 37 seconds total, while kDet-Thrust required only about 29 seconds, which is about 23.1% faster.

Comparing kDet-Thrust with and without self-collisions, then collision detection with self-collisions requires almost 14 times as much computation time as without.

Lastly, we evaluate how usable collision detection with kDet is for real time applications. With self-collisions, kDet-Thrust runs with

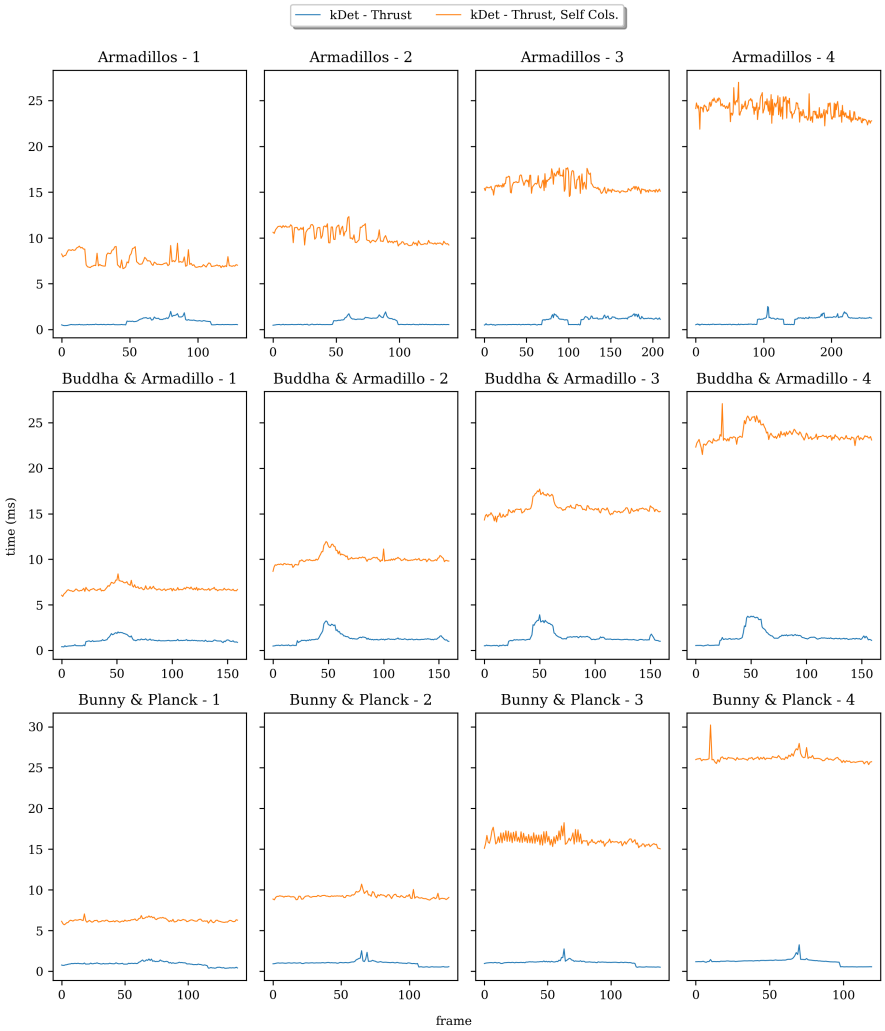


Figure 6.19: Runtimes for kDet-Thrust with and without self-collsions in the benchmarking scenes with varying resolutions

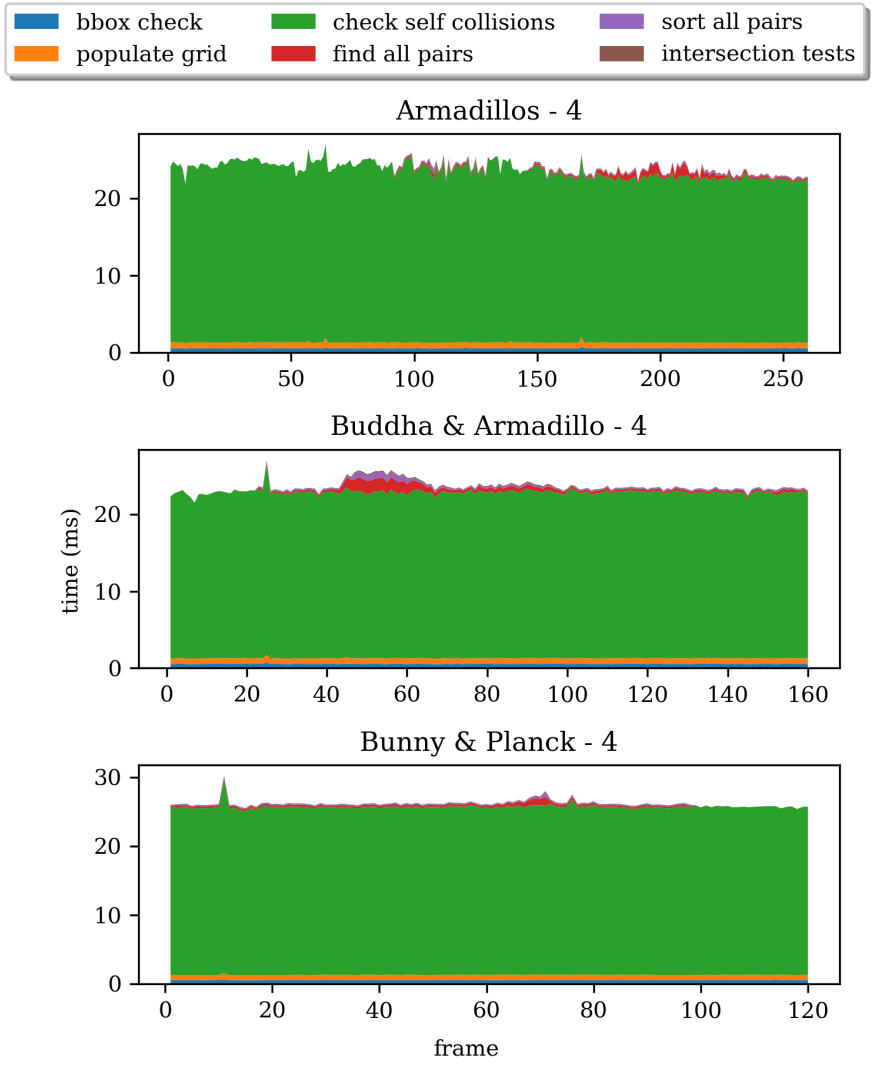


Figure 6.20: Stack plot of runtimes for kDet-Thrust with self-collisionsin the benchmarking scenes with the highest resolutions

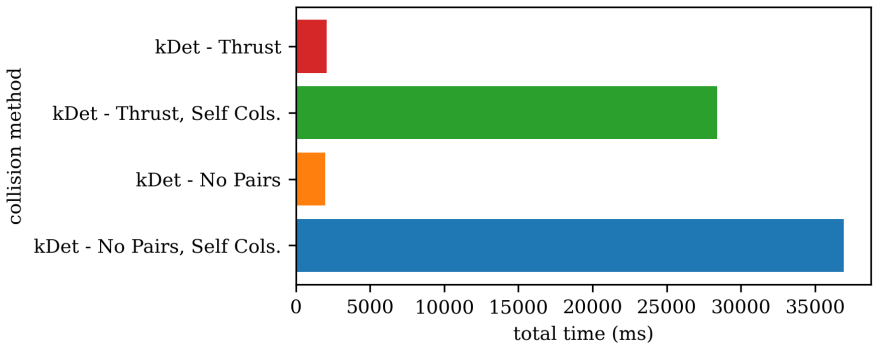


Figure 6.21: Total runtimes of kDet-methods over all scenes

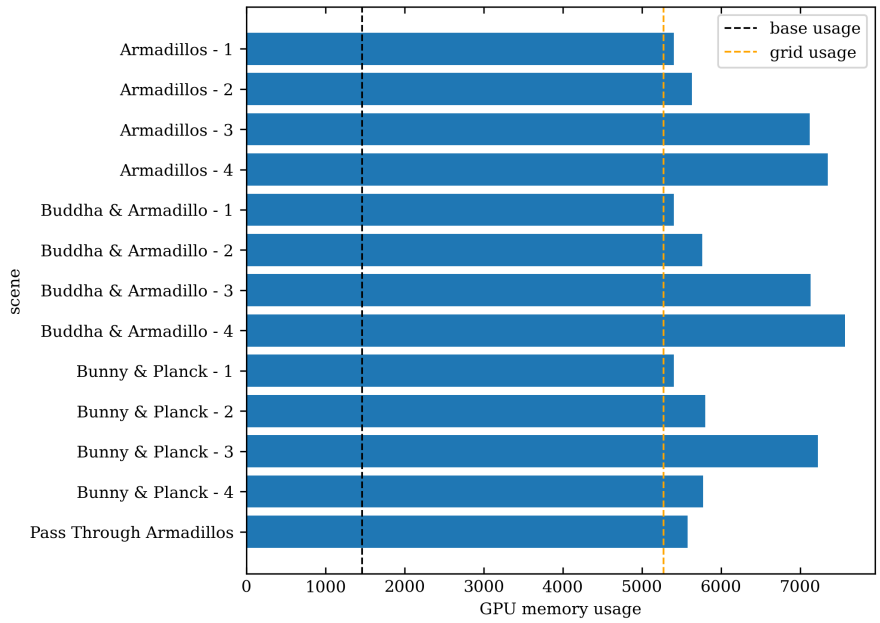


Figure 6.22: kDet Memory Usage: Baseline usage is delimited by the black line, the memory usage of the grid with the orange-yellow line. Additional Usage is to be attributed to the vector that stores the potential pairs.

over 38 fps on average for our benchmarking scenes of resolution 4, where the median frame time was about 23-26 ms. These scenes have between 170,000 - 190,000 tetrahedra each. For real time applications that could be a little too slow however, since the collision detection would not leave much room for other computations (e.g. physics and rendering), which would result in unusable frame rates. It is probably more realistic to target a maximum of 100,000 tetrahedra per scene, like with resolutions 3, which had a median frame time of 15-16 ms. If we are targeting a minimum frame rate of 30 fps, that leaves approximately the same amount of time per frame for different computations.

Having said that, without self-collisions, the fps count for both kDet methods goes into the low three digit range, leaving plenty room for other computations with very interactive frame rates.

6.3.3 *Memory Usage*

Before finishing up this chapter, we should also look at memory usage. Figure 6.22 demonstrates how the new version of kDet has rather high memory demands. The reason being, that in addition to the base memory usage (black vertical line) and the memory allocation for the spatial grid (yellow vertical line), we also need to allocate memory for the `thrust` device vector. Scenes of resolution 1 need around 5.5 GB in total scenes of resolution 2 require around 5.8 GB; while scenes of

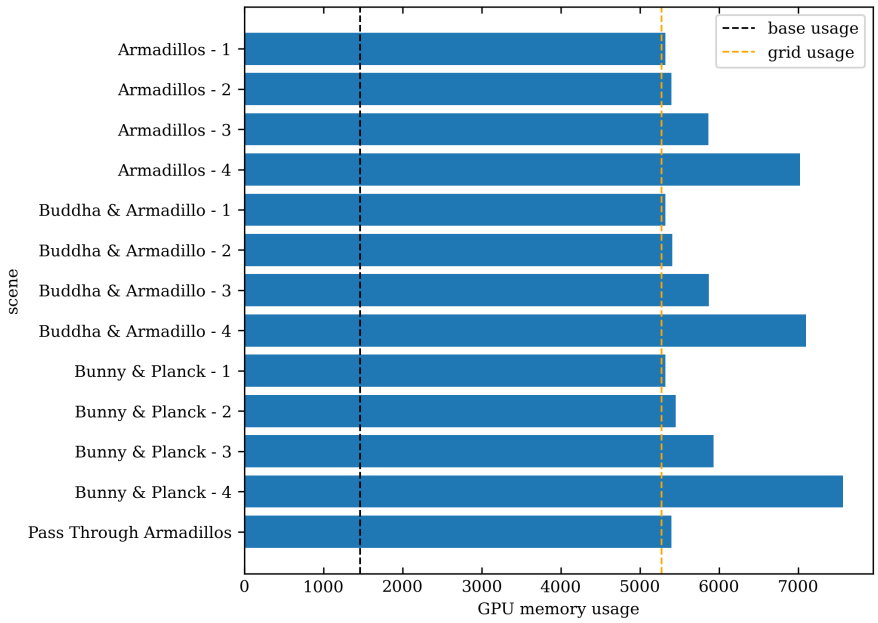


Figure 6.23: kDet Memory Usage with Fix Applied

resolution 3 and 4 need upwards of 7 GB. In comparison, the original kDet only requires the spatial grid in addition to the base usage, which is always around 5.2 GB.

However, we notice that *Bunny & Planck - 4* has suspiciously low memory usage. Also, if we remember that the vector size (and therefore also the memory demands) should grow by some square law. The memory usage for the different model resolutions does not reflect this characteristic. Instead there is only a huge jump between resolutions 2 and 3. After some investigation, I could trace back the issue to memory rollover during the calculation of the vector size. This was due to the usage of different integer types (signed and unsigned) within the same calculation. I implemented a fix and figure 6.23 presents the memory usage of the different scenes with said fix. We can observe that the allocated memory for the different resolutions now better fits our prior expectations. For scenes of resolution 1, the total memory usage ends up being around 5.3 GB; scenes of resolution 2 require around 5.5 GB; around 6.0 GB for scenes of resolution 3; and finally 7.0 - 7.5 G.B of memory usage for resolution 4 scenes.

The version of the software included with this work still uses the old implementation without the fix, because in my short testing the fixed version would crash at times.

7 FUTURE WORK

During the span of this research numerous interesting ideas came up in addition to the work already presented. Unfortunately, I lacked the time to (properly) implement and test these ideas, yet found them to be worthy of discussion. Hence, the decision to dedicate a whole chapter to them instead.

Note that the presented ideas might not be compatible with each other due to the respective limitations posed by each. Furthermore we will also discuss some considerations and possible downsides for every proposal.

7.1 HASH MAPS

Hash maps (or hash tables) could be used to replace the `thrust` GPU vector which stores the potential pairs. The intention is to avoid inserting duplicate pairs with the hashing, since with a deterministic hash function we can always evaluate whether pairs have already been inserted into the hash map. This in turn would save the time for sorting and removing said duplicates, which took considerable runtime as previously demonstrated.

I implemented a tentative version for this approach and ran a few benchmarks, which we will consequently discuss.

7.1.1 *Provisional Implementation*

On a general level, the hash map implementation only differs to the standard one in that when searching for the potential collision pairs they are not inserted into a list but instead a hash map. As previously explained, this also makes the step of sorting the pairs obsolete. The resulting collision pipeline is presented in figure 7.1, while algorithm 7.1 presents the updated procedure to find all potential collision pairs with a hash map.

In the following, we will discuss how the hash map is used in that function. In my short research about the topic of hash maps on the GPU, I was unable to find a suitable and easily usable implementation for this use case., which is why I had to implement one on my own. For this experiment I chose a closed hashing approach and combined it with a variation of the prime number method presented in chapter 4.2.2. The reasoning being that with the available code and my knowledge at that time this was the fastest way to put together a

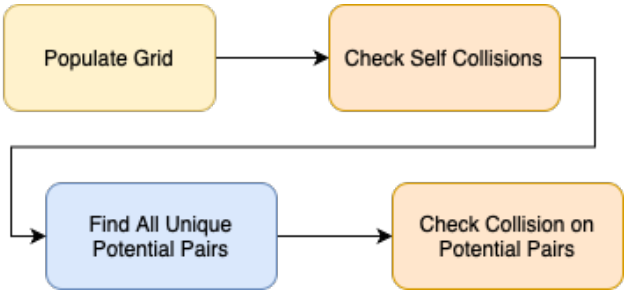


Figure 7.1: Collision Pipeline - kDet with Hash Map

Algorithm 7.1 Find All Potential Pairs - Hash Map

Input: objects A, B with tetrahedral meshes
Result: writes all potential collision pairs (t_A, t_B) to a list, with $t_A \in A$, $t_B \in B$ and t_A being smaller than t_B

```
1:  for all t_A ∈ A do in parallel
2:      determine size s of t_A
3:      determine grid layer l of t_A based on s
4:      find set C of all grid cells intersected by t_A on all layers l_i ≥ l
5:      for all cells c ∈ C do
6:          calculate hash key k based on grid position g of c
7:          while k does not point to bucket b in hash table
                with same hashed grid position g do
8:              increment hash misses m by 1
9:              recalculate hash k with g and m
10:         end while
11:         for all tetrahedra t_B ∈ B hashed into bucket b do
12:             if bbox of t_A and t_B overlap then
13:                 try inserting pair (t_A, t_B) into list of potential
                    collision pairs via hash map (see algorithm 7.2)
14:             end if
15:         end for
16:         if bucket b has reference to overflow bucket b_o then
17:             update b to b_o
18:             go-to line 10
19:         end if
20:     end for
21: end for
```

sample build for me. However, other possible approaches will also be discussed later in the results.

On the GPU, insertions into the hash map need to be done as atomic operations to avoid race conditions. For that I used CUDA's `atomicCAS()` (case and switch), which is implemented for various integer types. If we assign each tetrahedron a prime number, it would be possible to use the products of said primes (which would be some type of integer), as keys for the hash function and also as entries in the hash map to check if a pair has already been inserted. This works, once again, due to the prime factorization theorem: If we assigned each tetrahedron a distinct prime, the product of any two different tetrahedron primes will also be distinct (this by extension means that tetrahedra from different collision objects also cannot have the same prime number assigned to them). In addition, compared to the previous prime numbers approach we do not need any expensive division or modulo operations, only multiplications and equals comparisons.

Once the prime product has been successfully inserted into the hash map, the ids of the two tetrahedra in question are inserted into a separate list that stores the potential pairs. The rationale for this second list is to save a) the space of maintaining a second array with the size of the hash map to insert the potential pairs into (at the same hashed address) and b) the time afterwards to sweep through the array and gather all hashed pairs. Based on all that, algorithm 7.2 shows the procedure for inserting a potential pair into the hash map.

Algorithm 7.2 Insert Pair into Hash Map

Input: two tetrahedra t_A, t_B
Output: determines whether to write pair (t_A, t_B) into the list of potential collision pairs

Require: all $t_i \in \mathrm{A} \cup \mathrm{B}$ are assigned a unique prime p_i

```
1: calculate prime-product: q ← p_tA × p_tB
2: misses m ← 0
3: while True do
4:     generate hash-key k for hash map with q and m as input
5:     if hash map at position k is empty then
6:         atomic: write q into hash map at k
7:         write (t_A, t_B) to list of potential pairs return
8:     else if entry at position k is equal to q then return
9:     else                          ▷ entry at position k is not equal to q
10:        m ← m + 1
11:        continue
12:    end if
13: end while
```

In preprocessing we assign each tetrahedron a different prime number. We can simply pick the n-th largest prime for the n-th tetrahedron, but technically any primes would work, so long they are distinct.

On a little side note, because the algorithm will not multiply more than two primes the chance of facing the event of an integer overflow is much smaller and could realistically only happen if n becomes very large (roughly in the order of 10^8).

We begin by calculating the product of the primes of the two tetrahedra. This value is then used for our hash function and we check if the product of the primes is already inserted into the table at the hashed position. From here, there are three possibilities:

1. The cell at the hashed position is empty, thus we can write the prime product into it. We proceed to write the id-pair of the two tetrahedra into the list of potential pairs and the function exits.

2. The cell at the hashed position already contains the prime product. In this case the function exits, since the pair will already be in the list of potential collision pairs.

3. The cell at the hashed position contains another prime product. Here we increase the number of hash misses and pass it to the hash function together with prime product and repeat.

Given that the table is large enough to hold all potential pairs, the function either ends with inserting the prime product into the hash map and writing the id-pair to the list of potential pairs, or finding said product in the table and exiting. Doing so, we should end up with the same list of potential pairs as the standard implementation.

7.1.2 *Results*

The benchmarks were run on the same machine that was used for the evaluation, with a GPU block size of 128, SAT as the intersection test, and DJB2 hashing. The hash map used the same hashing algorithm as the grid.

Figure 7.2 shows the frame time comparison between our current standard method with the `thrust` device vector and the presented hash map for various scenes.

It should be noted, that because of an issue where the GPU memory would run full we could not benchmark the *Armadillos* and *Buddha & Armadillo* scenes on the highest resolution. This has mostly to do with the fact, that we stored both the `thrust` vector as well as the hash map which increased memory demands. For the same reason, the benchmarks were run without self-collisions. In any case, we observe

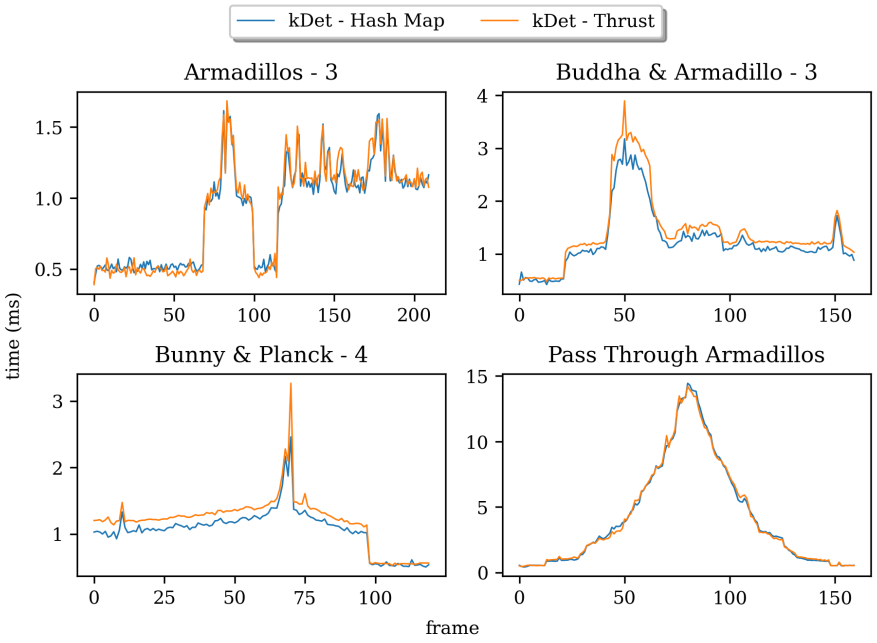


Figure 7.2: Frame times between kDet-HashMap and kDet-Thrust in various scenes.

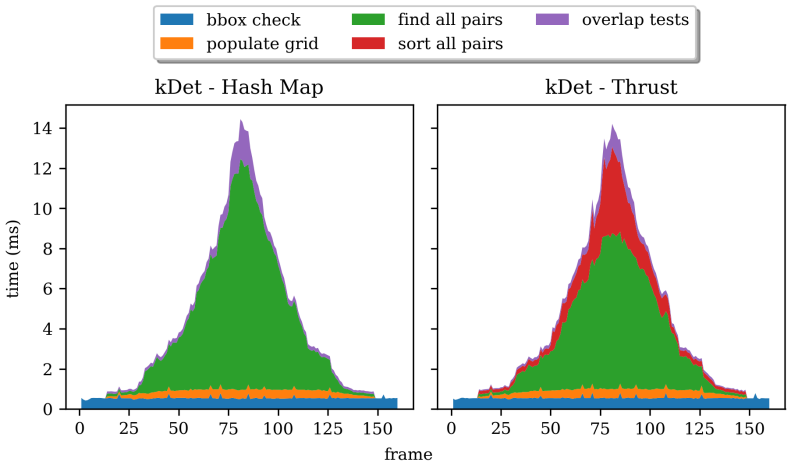


Figure 7.3: Stack plot of the frame times between kDet-HashMap and kDet-Thrust in the Pass-Through-Armadillos Scene.

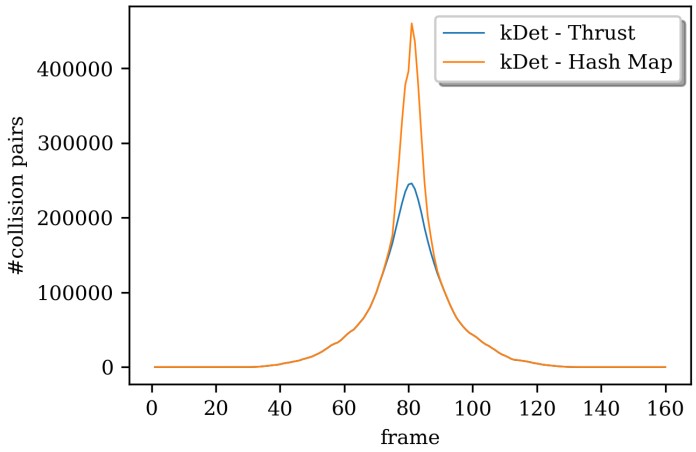


Figure 7.4: Plot of collision pairs per frame in the Pass-Through-Armadillos scene. Notice the difference between kDet-Thrust and kDet-HashMap.

that both methods display similar runtimes, with the hash map being slightly faster in the *Bunny & Planck* and *Buddha & Armadillo* scenes.

With figure 7.3 we can compare how much each step of the collision pipeline took for the `Pass Through Armadillos` scene. We observe that finding all pairs with the hash map took a similar amount of time as finding and sorting all pairs with the `thrust` vector. The other steps are identical between the methods, so no difference is to be expected there.

From that, we would conclude that our hash map implementation offers only marginal improvements in runtime compared to the standard method. The time to insert the pairs into the hash table seems to compensate for the time that was saved by skipping the sorting. Perhaps the great number of atomic operations causes a lot of stalling of the threads. Or maybe the closed hashing approach is not well suited for the GPU due to larger numbers of random memory accesses. Also, in my short testing I did not explore how the size of the hash map or other hashing methods would affect the hash collision rate and runtime.

Having said that, all of the benchmarking data for the presented hash map method regrettably has to be taken with a grain of salt. While plotting the collision pairs per frame for the `Pass Through Armadillos` scene, as shown in figure 7.4, I noticed a strange anomaly: kDet-HashMap would find additional collision pairs comparred to kDet-Thrust.

Unfortunately, this was another point that was noticed only a few weeks before thesis submission, which is why I did not have the time to fix it. Since I cannot estimate how much this bug affects the actual runtime, the presented figures can at best only be taken as a rough

measure. I estimate that fixing the bug would improve runtimes only slightly, although this remains speculation on my behalf.

On another note, even though the implementation with the prime numbers is "clever", in hindsight it might have been faster to just insert the pairs into the hash map directly and use them for comparison instead of the *prime product*. Afterwards, when all potential pairs are hashed they can be gathered by sweeping through the table once, before launching the intersection tests. At the time of implementation, I presumed the list sweep might turn out rather slow, which is why I stuck with the prime numbers. Whether this assumption is correct remains to be tested.

In addition, storing the pairs directly in the hash map would save the space of the second list as well as the time to insert the pairs into that list. This might be especially important on the GPU, since random memory accesses are rather expensive.

Another possible avenue is to use a bucket hashing approach, similar to what kDet uses for populating and traversing the grid.

Last but not least, the size of the hash map remains a concern for GPU memory usage, considering the need for the hash table to be large enough to minimize the number of hash collisions and store all potential pairs at the same time. Picking an appropriate size for the hash map, and whether it or the vector occupies more space is its own topic of analysis.

Nonetheless, I think that the hash map approach bears a lot of potential and could show better results with further research and experimentation.

7.2 SELF AND INTER OBJECT COLLISIONS IN THE SAME PASS

The current implementation of kDet checks self and inter object collisions in different stages of the algorithm. Figure 7.5 once again shows the current collision pipeline. Furthermore, we remember that self-collisions are comprised of the same three steps as inter object collisions, i.e. finding all (potential) collision pairs, sorting them & removing duplicates, and finally the intersection tests for each potential pair.

Checking these two types of collisions separately is convenient for the case where it is necessary to have the ability to toggle either on or off. But if the use-case requires that both should always be checked anyway, it might be possible to save the overhead of launching the GPU kernel twice for each of the mentioned steps. We could achieve this by handling both self and inter object collisions in the same pass.

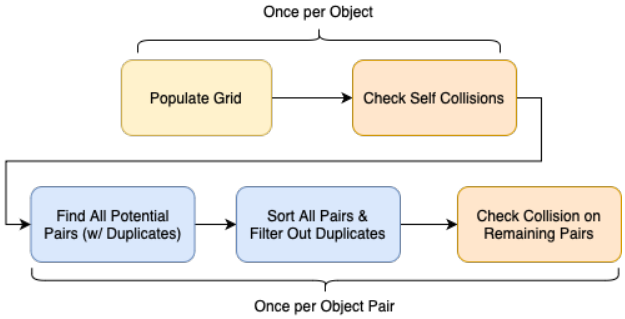


Figure 7.5: Current collision pipeline of kDet with annotations about how often a single pipeline step is executed during CD.

This is possible due to the nature of the kDet algorithm which allows us to ignore the object association of tetrahedra and just check against nearby tetrahedra to find all self and inter object collision pairs in one go.

Upon further thought, this approach could even be extended to the case where one has multiple objects in a scene. If we have another look at figure 7.5, we note that the last three steps are executed for each pair of objects. Thus, instead of checking for collisions between each pair of objects in a separate kernel, all collisions between the objects could also be checked in one pass.

The easiest way to go about these measures would be to store the tetrahedra of all objects in a single large list. Each collision object could then either just store the list of tetrahedron indices belonging to it, or - if we inserted the tetrahedra for each object sequentially - a pair of indices which would delimit the section of tetrahedra belonging to the object in said list. For example, we could have a list of 100,000 entries and an object with the index pair 20,000 and 30,000, telling us all the tetrahedra in that index range belong to it. This kind of object association for the tetrahedra could be useful when collision responses, like forces and deformations, should be applied to the object in question.

Finally, it should trivially follow that we can run the collision pipeline for kDet with just the large list of tetrahedra. Alternatively, it would be possible to assign an *object id* to each tetrahedron, so that we can associate each tetrahedron to its object during runtime. This would, for example, allow us again to toggle self (or inter object) collisions on or off, by simply adding a check for whether the object id of two tetrahedra matches or not.

The hope is that combining all the different passes into one would reduce the runtime of the algorithm, but still a few concerns remain. First of all, it might be possible that this method results in more global random memory accesses, which could slow down the algorithm

considerably. The same could be true for sorting the larger lists of potential pairs.

Moreover, while we are on that topic, we would also need to consider the growing memory usage of the vector which stores the potential pairs. Even currently the algorithm boasts almost 7.5 GB GPU memory usage for our highest resolution scenes. This is while reusing the lists for the different steps, meaning self collision and inter object collisions will clear the list between their execution and reuse it to save memory. If all steps are done in a single pass, the list cannot be reused but instead has to be large enough to hold all the potential self collision pairs as well as inter object collision pairs at once. This would necessitate a substantially larger list, further increasing memory demands.

As mentioned in the beginning, we also lose the ability to easily toggle kernels for certain steps on or off.

And lastly, the bounding box pre-check, which determined whether we would launch the collision check between objects at all and which would especially be relevant for scenes with multiple objects, would become obsolete or at the very least less useful. With the current implementation, we can just skip a kernel launch if we do not detect any bounding box overlap between two objects. In the one pass method, we would still launch the kernel since we at the very least check for self-collisions. On the other hand, it is also possible to argue that this is irrelevant because kDet only checks nearby simplices for collisions and thus if the bounding boxes of the objects do not overlap, no considerable time would be lost on checking collisions between those objects anyway.

7.3 SIZE OF POTENTIAL PAIRS VECTOR

As was mentioned earlier, the size of the vector which would store the potential pairs was picked rather arbitrarily and just so that the implementation would work and not crash. To reinstate, the size of the vector was calculated with:

$$size_{vector} = c\ (\#tetrahedra_A \times \#tetrahedra_B), \text{ where } c = 1/10.$$

This estimate was based on my assumption that the vector size should be somewhat proportional to the maximum possible combinations of potential pairs.

Nonetheless, speculations remain whether there exists a better heuristic to determine a suitable size for that vector. The two important criteria for such a heuristic would be to minimize the size of that vector to save memory, while staying large enough to avoid overflows during runtime. Importantly, the heuristic would need to

take into account the number of potential pairs before sorting and removing duplicate pairs.

Previously we saw that the number of potential pairs scales roughly cubically in proportion to the penetration depth. Perhaps that could be taken into account as well, i.e. if one can estimate the maximum expected penetration depth for an application, it should be possible to determine a scaling factor for the vector size. Though, when checking for self-collisions we theoretically have to assume the worst case scenario of 100% overlap anyway, so this consideration would only be useful for applications where self-collisions are not checked.

In the end, perhaps the most valuable heuristic could be derived from one of the major assumptions for the kDet algorithm itself, being the fact that we expect each simplex to be k-free, i.e. to have at most k "larger" neighbors. With proof 3.1 in chapter 3.1 we derived the upper bound of intersections for k-free sets of size n, which was nk multiplied with some constant depending on the objects included in the sets. This additional constant is irrelevant to us and can instead be replaced with some factor that accounts for implementation details like duplicate pairs. This would give us a heuristic of $size_{vector} \sim c * n * k$, where n is the total number of tetrahedra in the scene, c is some constant, and k is the maximum number of "larger" neighbors per tetrahedron. It might be possible to estimate or precompute c and k for a scene, though this is just a guess on my behalf.

8

CONCLUSIONS

In this work, I aimed to present how the existing kDet algorithm could be applied and extended to tetrahedral meshes. For that, first the geometric predicate at the foundation of the algorithm was proven for tetrahedra and polyhedra of any kind. Afterwards, the upper bound on the expected number of neighbors per polyhedra was shown, based on sphere coverings for the specific type of polyhedra.

We moved on to explore the short-comings of the old implementation of the algorithm, namely the inability to find all collisions pairs. Several solutions were discussed and one was successfully implemented. Due to a lack of available CD benchmarks with tetrahedral meshes, I created three different benchmark scenes in four different model resolutions, each consisting of two deformable and colliding objects. These scenes were made publically available for other researchers, together with additional material and the benchmarking data obtained in this work.

The implementation was tested extensively with different parameters, being the GPU block sizes, hashing methods for the grid, and intersection tests. Overall, the parameters seemed to have little effect on the runtimes of kDet. We only noticed an anomaly where our rudimentary imolementation of GJK would find little to no collision pairs for scenes of resolution 3 or higher. Runtimes without self-collision detection maxed out at around 4-5 ms and were comparable to the old version of kDet, meaning we obtained the list of unique collision pairs per frame at no additional cost in runtime. With self-collision detection, this dynamic flipped, as the new version of kDet ran faster, having a maximum runtime of around 30 ms per frame in scenes with higher resolutions. The old implementation ran almost 10-15 ms slower in these cases. Next, we looked at how self-collision detection exemplifies a worst case runtime for kDet and how frame times with it are basically one order of magnitude slower. We also looked at, how runtimes develop for different scene resolutions, and how they scale with increasing penetration depth in the Pass-Through-Armadillos scene. For the latter, the observation was that kDet scales cubically with the penetration depth. This was rationalized with the claim that the intersection volume and the number of tetrahedra contained within it grow by the same rate, thus keeping kDet's linear runtime. Furthermore, we can say that currently the steps to find and sort all potential collision pairs are the limiting factor of the algorithm. This ran against my assumption that the intersection test would be the major bottleneck. Memory demands of the algorithm were also ad-

dressed, because the algorithm in its present implementation needed upwards of 7.5 GB of VRAM.

Concluding the discussion of the results, I recommended that for real-time applications that intend to use kDet with self collision detection, maybe the number of tetrahedra per scene should not exceed around 100,000, to leave enough time per frame for other computations. Although, if self-collisions need not be considered, then even higher scene resolutions should be possible while comfortably maintaining interactive frame rates.

Lastly, we discussed several possible high-level optimizations and adjustments that could be incorporated in the future for better performance. One interesting idea was to check self-collisions and inter-object collisions between possibly multiple object in the same traversal of the grid. Most noteworthy however, was the concept of using a hash map instead of lists to store potential collision pairs before the intersection tests. Considering the complexity of programming on the GPU and the impact of small implementation details, I believe there are also numerous low-level optimizations that could have a very noticeable effect.

To sum up, I hope with these results to have offered a collision detection which offers sufficient capabilities for various applications and a good point of comparison for other GPU collision methods.

A

APPENDIX

A.1 SOURCE CODE

The software is written in C++ and can be built with CMake to generate the project files. The source code is located in `/kdet-tetrahedra`. The project depends mainly on the CUDA SDK. The following libraries are also required, but should already be included together with the source code:

- Eigen
- GLFW
- OpenGL
- GLEW
- Threads

A.1.1 *Asset Folder Location & CUDA SDK Samples*

When using Windows and compiling with Visual Studio, copy the *assets* folder from the root directory of the project to:

- `buildfolder/src/apps` for running the executable with Visual Studio
- `buildfolder/bin/[DEBUG | RELEASE | etc.]` or the same location as the executable, when running it by itself

The build process might also need the CUDA SDK Samples [1]. Download them and add the path of the directory to `src/kdet_tetrahedron/CMakeList.txt`:

```
[...]
find_path(CUDA_SDK_INCLUDE_DIR
  [...]
  PATHS
    "C:/ProgramData/NVIDIA Corporation/CUDA Samples/${
        CUDA_VERSION}/common"
    # PATH TO GPU TOOLKIT SAMPLES
    <add the path to CUDA samples here>
    "$ENV{HOME}/NVIDIA_CUDA-${CUDA_VERSION}_Samples/common")
endif()
```

1 CUDA SDK Samples - GitHub: `https://github.com/NVIDIA/cuda-samples` (last accessed: July 25, 2023)

A.1.2 *Demo Application*

The demo_tet.exe offers a visual user interface that can be interacted with. Several scenes can be loaded and collision detection with various methods can be run.

For animated scenes, benchmarks can be performed. In that case, the scene will be animated and benchmarked with the selected parameters. Once finished, the recorded results for each frame are averaged over the number of runs and written to an output file, which will be placed under `assets/animations/<scene>/benchmark_data` (consider where assets were placed for the build process; see Chapter A.1.1). The file name will reflect the parameters used for the benchmark. See Chapter A.1.4 for details on the benchmarking files.

Please use the command-line benchmarking tool if the raw output files for each run are needed, compared to only the averaged results.

Controls, options and other information regarding the software are documented in the project's README.

A.1.2.1 *Notable Bugs*

- The hashing method for kDet cannot be changed before having executed the collision check for a scene at least once. Doing otherwise will crash the program.
- When changing scenes and using kDet several times, the GPU memory will most likely run full and crash the program. In such a case, just restart the demo application.

A.1.3 *Command-Line Benchmarking Tool*

Benchmarks without rendering can be executed from the command line with benchmarking_tet.exe (note, that during the benchmarking an empty window will pop-up).

It can be used as follows:

```
$ ./benchmark_tet.exe [scene] [options]
```

The scene option has to be placed first, but other parameters can be passed in arbitrary order. Additionally, all parameters are optional since default values are provided. The option flags are listed in table A.1 and can also be referred to via the project's README.

Example usage:

```
$ ./benchmark_tet.exe buddha_and_armadillo -r 2 -n 5 -c
    gpu_no_loop -o sat-mtv -d ./../../exports/benchmark_data
```

IMPORTANT: The program does not have proper error handling, so please just stick to the provided values. If values are faulty, the

flag	option	possible values	default value
	scene	pass_through_armadillos buddha_and_armadillo bunny_and_planck armadillos	←
-r	model resolution	1 - 4	1
-c	collision method	kdet kdet_no_pairs gpu gpu_no_loop	kdet
-o	simplex intersection test	sat sat_no_edges sat_mtv gjk	sat
-g	gpu block size	16 32 64 128 256	128
-n	number of runs	1 - 20	10
-s	self collisions	yes or y no or n	no
-h	hashing algorithm (if applicable)	djb2 fnv-1a morton simple	djb2
-d	destination folder for benchmark data	any string value	`exports/benchmark_data`, usually turns out to `buildfolder/bin/[DEBUG | RELEASE | etc.] /exports/benchmark_data` (see Chapter A.1.1)

Table A.1: Option flags and default values for CMD benchmarking tool

program has undefined behavior. Always check the command line and file output to make sure.

A.1.4 *Benchmarking Files*

Files produced by the benchmarking routines, be it by the demo application or the benchmarking cmd tool, are in *.csv* format.

Each file has a header, which contains some meta data about the run. It might look something like this:

```
scene name, Buddha & Armadillo - 2
algorithm, gpu-no-loop
cuda block size, 128
overlap test, sat
self collisions, no
hashing algorithm, none
#frames, 160
total runs for benchmark, 5
#tetsA, 8647
#tetsB, 5912
#verticesA, 28584
#verticesB, 20052
```

Which is then followed by the actual data about the run. This usually includes various timers (in ms) and info about the # collision pairs etc.:

```
frame, host time, device time, bbox time, collision time, col
    tets A to B, col tets B to A, collision pairs
1, 1.39198, 0.491174, 0.491174, 0, 0, 0, 0
2, 1.34758, 0.505254, 0.505254, 0, 0, 0, 0
3, 1.37228, 0.537926, 0.537926, 0, 0, 0, 0
4, 1.44264, 0.631341, 0.631341, 0, 0, 0, 0
5, 1.32118, 0.51639, 0.51639, 0, 0, 0, 0
[...]
```

A.2 BENCHMARKING SCENES & DATA

All the obtained benchmarking data, *SOFA* scene-configurations, exported animations & models, as well as short video recordings of the animations can be found under `kdet-benchmarks`.

BIBLIOGRAPHY

[BLD13] J. Baert, A. Lagae, and Ph. Dutré. "Out-of-core Construction Of Sparse Voxel Octrees." In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, pp. 27–32. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492048. URL: http://doi.acm.org/10.1145/2492045.2492048.

[Ber99] G. Van den Bergen. "A Fast and Robust GJK Implementation for Collision Detection of Convex Objects." In: *J. Graph. Tools* 4.2 (Mar. 1999), pp. 7–25. DOI: 10.1080/10867651.1999.10487502. URL: https://doi.org/10.1080/10867651.1999.10487502.

[Ber97] G. van den Bergen. "Efficient Collision Detection of Complex Deformable Models using AABB Trees." In: *Journal of Graphics Tools* 2.4 (1997), pp. 1–13. DOI: 10.1080/10867651.1997.10487480. URL: https://doi.org/10.1080/10867651.1997.10487480.

[BSA17] M. Bern, J. R. Shewchuk, and N. Amenta. "TRIANGULATIONS AND MESH GENERATION." In: ed. by C. Toth, J. OŔourke, and J. Goodman. 3rd ed. Boca Raton, Florida: CRC Press, 2017. Chap. 29, pp. 763–785. ISBN: 9781315119601. URL: https://doi.org/10.1201/9781315119601.

[BF23] Blender-Foundation. *Blender: Homepage*. Last accessed: 23.07.23. 2023. URL: https://www.blender.org.

[CL16] G. Capannini and T. Larsson. "Adaptive Collision Culling for Large-Scale Simulations by a Parallel Sweep and Prune Algorithm." In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by E. Gobbetti and W. Bethel. The Eurographics Association, 2016. ISBN: 978-3-03868-006-2. DOI: 10.2312/pgv.20161177.

[CL18] G. Capannini and T. Larsson. "Adaptive Collision Culling for Massive Simulations by a Parallel and Context-Aware Sweep and Prune Algorithm." In: *IEEE Transactions on Visualization and Computer Graphics* 24.7 (2018), pp. 2064–2077. DOI: 10.1109/TVCG.2017.2709313.

[Cig+08] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia. "MeshLab: an Open-Source Mesh Processing Tool." In: *Eurographics Italian Chapter Conference*. Ed. by V. Scarano, R. D. Chiara, and U. Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[CL96] B. Curless and M. Levoy. "A Volumetric Method for Building Complex Models from Range Images." In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1996, New Orleans, LA, USA, August 4-9, 1996*. Ed. by J. Fujii. ACM, 1996, pp. 303–312. DOI: 10.1145/237170.237269. URL: https://doi.org/10.1145/237170.237269.

[Dev23] N. Developer. *CUDA Thrust API Documentation*. Last accessed: 23.07.23. 2023. URL: https://docs.nvidia.com/cuda/thrust/index.html.

[DK83] D. P. Dobkin and D. G. Kirkpatrick. "Fast detection of polyhedral intersection." In: *Theoretical Computer Science* 27.3 (1983). Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982, pp. 241–253. DOI: https://doi.org/10.1016/0304-3975(82)90120-7. URL: https://www.sciencedirect.com/science/article/pii/0304397582901207.

[EL07] M. Eitz and G. Lixu. "Hierarchical Spatial Hashing for Real-time Collision Detection." In: *IEEE International Conference on Shape Modeling and Applications 2007 (SMI '07)*. 2007, pp. 61–70. DOI: 10.1109/SMI.2007.18.

[Eri04] C. Ericson. *Real-Time Collision Detection*. USA: CRC Press, Inc., 2004, p. 288. ISBN: 1558607323.

[Fau+12] F. Faure et al. "SOFA: A Multi-Model Framework for Interactive Physical Simulation." In: *Soft Tissue Biomechanical Modeling for Computer Assisted Surgery*. Ed. by Y. Payan. Vol. 11. Studies in Mechanobiology, Tissue Engineering and Biomaterials. Springer, June 2012, pp. 283–321. DOI: 10.1007/8415_2012_125. URL: https://inria.hal.science/hal-00681539.

[FNV91] G. Fowler, L. C. Noll, and K.-P. Vo. *FNV-1a alternate algorithm*. Last accessed: 23.07.23. 1991. URL: http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-1a.

[GPR02] F. Ganovelli, F. Ponchio, and C. Rocchini. "Fast Tetrahedron-Tetrahedron Overlap Algorithm." In: *Journal of Graphics Tools* 7.2 (2002), pp. 17–25. DOI: 10.1080/10867651.2002.10487557. URL: https://doi.org/10.1080/10867651.2002.10487557.

[GASF94] A. Garcia-Alonso, N. Serrano, and J. Flaquer. "Solving the collision detection problem." In: *IEEE Computer Graphics and Applications* 14.3 (1994), pp. 36–43. DOI: 10.1109/38.279041.

[GJK88] E. Gilbert, D. Johnson, and S. Keerthi. "A fast procedure for computing the distance between objects in three-dimensional space." In: *Robotics and Automation, IEEE Journal of* 4 (May 1988), pp. 193–203. DOI: 10.1109/56.2083.

[Got96] S. Gottschalk. *Separating axis theorem*. Technical Report TR96-024. Department of Computer Science, UNC Chapel Hill, 1996.

[GLM96] S. Gottschalk, M. Lin, and D. Manocha. "OBBTree: A Hierarchical Structure for Rapid Interference Detection." In: *Computer Graphics* 30 (Oct. 1996). DOI: 10.1145/237170.237244.

[HSS95] R. Hardin, N. Sloane, and W. Smith. *Spherical Coverings*. Last accessed: 23.07.23. 1995. URL: http://neilsloane.com/coverings/index.html.

[He+15] L. He, R. Ortiz, A. Enquobahrie, and D. Manocha. "Interactive continuous collision detection for topology changing models using dynamic clustering." In: *Proceedings of the 19th symposium on interactive 3D graphics and games*. i3D '15. San Francisco, California: Association for Computing Machinery, 2015, pp. 47–54. ISBN: 9781450333924. DOI: 10.1145/2699276.2699286. URL: https://doi.org/10.1145/2699276.2699286.

[HKM95] M. Held, J. Klosowski, and J. Mitchell. "Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs." In: *Proc. 7th Canad. Conf. Comput. Geom. (CCCG'95)*. 1995, pp. 205–210.

[KL96] V. Krishnamurthy and M. Levoy. "Fitting Smooth Surfaces to Dense Polygon Meshes." In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1996, New Orleans, LA, USA,*

August 4-9, 1996. Ed. by J. Fujii. ACM, 1996, pp. 313–324. DOI: 10.1145/237170.237270. URL: https://doi.org/10.1145/237170.237270.

[LMM10] C. Lauterbach, Q. Mo, and D. Manocha. "gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries." In: *Computer Graphics Forum* (2010). DOI: 10.1111/j.1467-8659.2009.01611.x.

[MZ15] D. Mainzer and G. Zachmann. "Collision Detection Based on Fuzzy Scene Subdivision." In: *GPU Computing and Applications*. Ed. by Y. Cai and S. See. Singapore: Springer Singapore, 2015, pp. 135–150. ISBN: 978-981-287-134-3. DOI: 10.1007/978-981-287-134-3_9. URL: https://doi.org/10.1007/978-981-287-134-3_9.

[MAC04] D. Marchal, F. Aubert, and C. Chaillou. "Collision between Deformable Objects Using Fast-Marching on Tetrahedral Models." In: *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '04. Grenoble, France: Eurographics Association, 2004, pp. 121–129. ISBN: 3905673142. DOI: 10.1145/1028523.1028540. URL: https://doi.org/10.1145/1028523.1028540.

[Mig10] S. Migdalskiy. "SAT in Narrow Phase and Contact-Manifold Generation." In: *Game Physics Pearls*. Ed. by G. van den Bergen and D. Gregorius. Taylor & Francis, 2010, pp. 63–98. ISBN: 9781568814742. URL: https://books.google.de/books?id=8vIpAQAAMAAJ.

[MJ23] N. Mirzayousef Jadid. *kDet Tetrahedron Animation Benchmarks*. Last accessed: 23.07.23. 2023. URL: https://gitlab.informatik.uni-bremen.de/cgvr_public/kdet-benchmarks.

[Mö97] T. Möller. "A Fast Triangle-Triangle Intersection Test." In: *Journal of Graphics Tools* 2.2 (1997), pp. 25–30. DOI: 10.1080/10867651.1997.10487472. URL: https://doi.org/10.1080/10867651.1997.10487472.

[MPB17] M. Montanari, N. Petrinic, and E. Barbieri. "Improving the GJK Algorithm for Faster and More Reliable Distance Queries Between Convex Objects." In: *ACM Trans. Graph.* 36.3 (June 2017). DOI: 10.1145/3083724. URL: https://doi.org/10.1145/3083724.

[Mon+22] L.-R. Montaut, Q. L. Lidec, J. Sivic, and J. Carpentier. "Collision Detection Accelerated: An Optimization Perspective." In: *ArXiv* abs/2205.09663 (2022). URL: https://doi.org/10.48550/arXiv.2205.09663.

[MP78] D. Muller and F. Preparata. "Finding the intersection of two convex polyhedra." In: *Theoretical Computer Science* 7.2 (1978), pp. 217–236. DOI: https://doi.org/10.1016/0304-3975(78)90051-8. URL: https://www.sciencedirect.com/science/article/pii/0304397578900518.

[NVI23] NVIDIA. *NVIDIA Developer: PhysX*. Last accessed: 23.07.23. 2023. URL: https://developer.nvidia.com/physx-sdk.

[PKS10] S. Pabst, A. Koch, and W. Straßer. "Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces." In: *Computer Graphics Forum* 29.5 (2010), pp. 1605–1612. DOI: https://doi.org/10.1111/j.1467-8659.2010.01769.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01769.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01769.x.

[RRM07] V. Roussev, G. G. Richard, and L. Marziale. "Multi-resolution similarity hashing." In: *Digital Investigation* 4 (2007), pp. 105–113. DOI: https://doi.org/10.1016/j.diin.2007.06.011. URL: https://www.sciencedirect.com/science/article/pii/S1742287607000473.

[Rus+] S. Rusinkiewicz, D. DeCarlo, A. Finkelstein, and A. Santella. *Suggestive Contour Gallery*. Last accessed: 23.07.23. URL: https://gfx.cs.princeton.edu/proj/sugcon/models/.

[Set96] J. A. Sethian. "A fast marching level set method for monotonically advancing fronts." In: *Proceedings of the National Academy of Sciences* 93.4 (1996), pp. 1591–1595. DOI: 10.1073/pnas.93.4.1591. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.93.4.1591. URL: https://www.pnas.org/doi/abs/10.1073/pnas.93.4.1591.

[Si15] H. Si. "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator." In: *ACM Trans. Math. Softw.* 41.2 (2015). DOI: 10.1145/2629697. URL: https://doi.org/10.1145/2629697.

[Sid23] SideFx. *SideFx: Houdini*. Last accessed: 23.07.23. 2023. URL: https://www.sidefx.com/products/houdini/.

[Sne08] G. Snethen. "XenoCollide: Complex Collision Made Simple." In: *Game Programming Gems 7*. Ed. by S. Jacobs. Charles River Media, 2008, pp. 165–178.

[SF21] Sofa-Framework. *SOFA Training Session 2020: Introduction to SOFA*. Last accessed: 23.07.23. 2021. URL: https://youtu.be/KHTAgD1oG8Y.

[Sof06] Sofa. *SOFA Github Repository*. Last accessed: 23.07.23. 2006. URL: https://github.com/sofa-framework/sofa/tree/master/examples.

[Tan+09] M. Tang, S. Curtis, S.-E. Yoon, and D. Manocha. "ICCD: Interactive Continuous Collision Detection between Deformable Models Using Connectivity-Based Culling." In: *IEEE Transactions on Visualization and Computer Graphics* 15 (2009), pp. 544–557. DOI: http://doi.ieeecomputersociety.org/10.1109/TVCG.2009.12.

[Tan+14] M. Tang, R. Tong, Z. Wang, and D. Manocha. "Fast and Exact Continuous Collision Detection with Bernstein Sign Classification." In: 33.6 (Nov. 2014). DOI: 10.1145/2661229.2661237. URL: https://doi.org/10.1145/2661229.2661237.

[Tan+18a] M. Tang, t. wang, Z. Liu, R. Tong, and D. Manocha. "I-Cloth: Incremental Collision Handling for GPU-Based Interactive Cloth Simulation." In: *ACM Trans. Graph.* 37.6 (Dec. 2018). DOI: 10.1145/3272127.3275005. URL: https://doi.org/10.1145/3272127.3275005.

[Tan+18b] M. Tang, Z. Liu, R. Tong, and D. Manocha. "PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs." In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.1 (July 2018). DOI: 10.1145/3203188. URL: https://doi.org/10.1145/3203188.

[TCF13] V. Tereshchenko, S. Chevokin, and A. Fisunenko. "Algorithm for Finding the Domain Intersection of a Set of Polytopes." In: *Procedia Computer Science* 18 (2013). 2013 International Conference on Computational Science, pp. 459–464. DOI: https://doi.org/10.1016/j.procs.2013.05.209. URL: https://www.sciencedirect.com/science/article/pii/S1877050913003529.

[Tes+03] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross. "Optimized Spatial Hashing for Collision Detection of Deformable Objects." In: *VMV'03: Proceedings of the Vision, Modeling, Visualization* 3 (Dec. 2003).

[Tes+05] M. Teschner et al. "Collision Detection for Deformable Objects." In: *Computer Graphics Forum* 24.1 (2005), pp. 61–81. DOI: https://doi.org/10.1111/j.1467-8659.2005.00829.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2005.00829.x.

[THS19] Y. Tian, Y. Hu, and X. Shen. "A multi-GPU finite element computation and hybrid collision handling process framework for brain deformation simulation." In: *Computer Animation and Virtual Worlds* 30.1 (2019). e1846 cav.1846, e1846. DOI: https://doi.org/10.1002/cav.1846. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cav.1846.

[TL94] G. Turk and M. Levoy. "Zippered polygon meshes from range images." In: *Proceedings of the 21th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994, Orlando, FL, USA, July 24-29, 1994*. Ed. by D. Schweitzer, A. S. Glassner, and M. Keeler. ACM, 1994, pp. 311–318. DOI: 10.1145/192161.192241. URL: https://doi.org/10.1145/192161.192241.

[VDB01] G. Van Den Bergen. "Proximity queries and penetration depth computation on 3d game objects." In: *Game developers conference*. Vol. 170. 2001, p. 209.

[VDB03] G. Van Den Bergen. *Collision Detection in Interactive 3D Environments*. Oct. 2003. ISBN: 9780429176364. DOI: 10.1201/9781482297997.

[VG05] J.-L. Verger-Gaugry. "Covering a Ball with Smaller Equal Balls in $\mathbb{R}^n$." In: *Discrete & Computational Geometry* 33.1 (2005), pp. 143–155. DOI: 10.1007/s00454-004-2916-2. URL: https://doi.org/10.1007/s00454-004-2916-2.

[VT94] P. Volino and N. M. Thalmann. "Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity." In: *Computer Graphics Forum* 13.3 (1994), pp. 155–166. DOI: https://doi.org/10.1111/1467-8659.1330155. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1330155.

[Wal13] I. Wald. *The Utah 3D Animation Repository*. Last accessed: 23.07.23. 2013. URL: http://www.sci.utah.edu/~wald/animrep/.

[WC21] M. Wang and J. Cao. "A review of collision detection for deformable objects." In: *Computer Animation and Virtual Worlds* 32.5 (2021), e1987. DOI: https://doi.org/10.1002/cav.1987. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cav.1987.

[Wan+17] T. Wang, Z. Liu, M. Tang, R. Tong, and D. Manocha. "Efficient and Reliable Self-Collision Culling Using Unprojected Normal Cones." In: *Computer Graphics Forum* 36.8 (2017), pp. 487–498. DOI: https://doi.org/10.1111/cgf.13095. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13095.

[Wan+18] T. Wang, M. Tang, Z. Wang, and R. Tong. "Accurate self-collision detection using enhanced dual-cone method." In: *Computers & Graphics* 73 (2018), pp. 70–79. DOI: https://doi.org/10.1016/j.cag.2018.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0097849318300475.

[Wel13] R. Weller. "A Brief Overview of Collision Detection." In: *New Geometric Data Structures for Collision Detection and Haptics*. Heidelberg: Springer International Publishing, 2013, pp. 9–46. DOI: 10.1007/978-3-319-01020-5_2. URL: https://doi.org/10.1007/978-3-319-01020-5_2.

[WDZ17] R. Weller, N. Debowski, and G. Zachmann. "kDet: Parallel Constant Time Collision Detection for Polygonal Objects." In: *Computer Graphics Forum* 36.2 (2017), pp. 131–141. DOI: https://doi.org/10.1111/cgf.13113. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13113.

[Win20] WinterDev. *GJK: Collision detection algorithm in 2D/3D*. Last accessed: 23.07.23. 2020-08-29. URL: https://blog.winter.dev/2020/gjk-algorithm/.

[WLZ14] T. H. Wong, G. Leach, and F. Zambetta. "An adaptive octree grid for GPU-based collision detection of deformable objects." In: *The Visual Computer* 30.6 (2014), pp. 729–738. DOI: 10.1007/s00371-014-0954-1. URL: https://doi.org/10.1007/s00371-014-0954-1.

[Wyn12] E. Wynn. *Answer to: Covering a Unit Ball with Balls Half the Radius*. Last accessed: 23.07.23. Aug. 5, 2012. URL: https://mathoverflow.net/a/103981.

[Ye+16] X. Ye, J. Zhang, P. Li, T. Wang, and S. Guo. "A fast and stable vascular deformation scheme for interventional surgery training system." In: *BioMedical Engineering OnLine* 15.1 (2016), p. 35. DOI: 10.1186/s12938-016-0148-3. URL: https://doi.org/10.1186/s12938-016-0148-3.

[dan10] danfis. *The Utah 3D Animation Repository*. Last accessed: 23.07.23. 2010. URL: https://github.com/danfis/libccd.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`