

GPU-Based Collision Detection and Resolution for Elastic Polyhedral Meshes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Elias Füricht, BSc

Matrikelnummer 52005515

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Mitwirkung: David Hahn, PhD

Wien, 29. Juli 2026

Elias Füricht

Michael Wimmer



GPU-Based Collision Detection and Resolution for Elastic Polyhedral Meshes

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Elias Füricht, BSc

Registration Number 52005515

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Assistance: David Hahn, PhD

Vienna, July 29, 2026

Elias Füricht

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Elias Füricht, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 29. Juli 2026

Elias Füricht

Danksagung

Zuallererst möchte ich meiner Familie danken, die mich mit ihrem kontinuierlichen Support durch das ganze Studium begleitet hat, ohne den ich mir nicht sicher bin ob ich es überhaupt geschafft hätte. An meine Eltern Josef und Gerda, dafür dass ihr meine Neugier immer ermutigt und unterstützt habt und immer an mich geglaubt habt sogar wenn ich es nicht tat. An meinen Bruder Max, ohne dein sonniges Gemüt wäre meines vielleicht noch dunkler geworden. An meine Oma Maria, dafür dass du mir ein Zuhause zur Verfügung gestellt hast ohne jemals auch nur irgendetwas im Gegenzug dafür zu erwarten.

Außerdem möchte ich meinen Freunden danken, mit denen ich die Herausforderungen des Studiums an der TU Wien angegangen bin. Ich glaube, dass ich es ohne euch nicht geschafft hätte: David, Max, Aline, Nono, Simon, Sabina, Jen und Yeun.

Abschließend möchte ich meinem Betreuer David Hahn meinen Dank aussprechen. Vielen Dank, dass du mich durch diese Arbeit begleitet hast und stets zur Verfügung standest, um meine Fragen zu beantworten und mir wertvollen Input zu geben.

Acknowledgements

First and foremost, I want to thank my family, without your continuous support during the whole duration of my studies I am not sure I would have made it to the finish line. To my parents Gerda and Josef, for endlessly encouraging and supporting my curiosity, and for always believing in me, even when I did not. To my brother Max, without your sunny nature I would have potentially gone to a dark place. To my grandmother Maria, for providing me with a living space without ever asking for anything in return.

I also want to thank my friends with whom I have tackled the courses at TU Wien. I do not think I would have been able to do it without you: David, Max, Aline, Nono, Simon, Sabina, Jen and Yeun.

Lastly I want to express my gratitude to my advisor, David Hahn. Thank you for guiding me through this thesis and for always being there to answer my questions and provide valuable input.

Kurzfassung

Die Kollisionsbehandlung deformierbarer Objekte ist besonders anspruchsvoll, wenn deren Volumen durch allgemeine polyedrische Elemente diskretisiert wird. Im Gegensatz zu herkömmlichen tetraedrischen Meshes weisen diese Elemente eine unregelmäßige Topologie auf, können nicht-planare polygonale Flächen enthalten und sich während der Simulation zu nichtkonvexen Formen verformen. Diese Arbeit präsentiert eine vollständig auf der GPU laufende Pipeline zur Kollisionsbehandlung elastischer polyedrischer Meshes, welche ein bestehendes Simulationsframework auf Basis der Virtual-Elemente-Methode erweitert.

Die entwickelte Pipeline traversiert eine Bounding Volume Hierarchy (BVH), um Kandidatenpaare von Elementen mit sich überschneidenden Axis-Aligned Bounding Boxes (AABBs) zu erzeugen. Topologische Nachbarschaftsinformationen sowie eine Klassifikation aktiver Elemente schließen Tests aus, die zu keinem relevanten Kontakt führen können. Für die verbleibenden Paare rasterisiert ein aus früheren Arbeiten adaptiertes image-based Kollisionserkennungsverfahren mithilfe eines speziell entwickelten CUDA-Compute-Rasterizers eine triangulierte Darstellung der polyedrischen Oberflächen. Anschließend reduziert ein Korrekturverfahren das Überlappungsvolumen kollidierender Elemente und berechnet außerdem die daraus resultierenden Kräfte, die in die nachfolgende Solver-Iteration eingerechnet werden.

Die Evaluierung zeigt, dass das Detektionsverfahren eine hohe Genauigkeit aufweist, das ermittelte paarweise Überlappungsvolumen im Allgemeinen über die Zeit reduziert wird und die Filterung aktiver Elemente und topologischer Nachbarschaften eine große Anzahl unnötiger Kollisionstests vermeidet. Von den drei Stufen der Kollisionsbehandlung trägt die image-based Narrow Phase den größten Laufzeit- und Speicheraufwand und stellt daher das vorrangige Ziel weiterer Optimierungen dar. Insgesamt bildet die Implementierung eine praktische Grundlage für die GPU-basierte Kollisionsbehandlung deformierbarer polyedrischer Netze und zeigt mehrere Ansatzpunkte für zukünftige Arbeiten auf.

Abstract

Collision handling for deformable objects is particularly challenging when their volumes are discretized into general polyhedral elements. Unlike standard tetrahedral meshes, these elements have irregular topology, may contain non-planar polygonal faces, and can become non-convex during simulation. This thesis presents a collision handling pipeline for elastic polyhedral meshes that runs entirely on the GPU and extends an existing Virtual Element Method simulation framework.

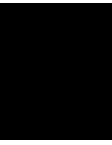
Our pipeline traverses a Bounding Volume Hierarchy (BVH) to generate candidate pairs of intersecting element Axis-Aligned Bounding Boxes (AABBs). Topological neighborhood information and an active-element classification avoid tests that cannot produce relevant contact. For the remaining pairs, an image-based collision detection method, adapted from previous work, rasterizes a triangulated representation of the polyhedral surfaces using a custom CUDA compute rasterizer. A position-based correction approach is then used to reduce interpenetration between colliding elements and also computes resulting forces to be fed into the subsequent solver iteration.

Our evaluation shows that the detection method is accurate, the detected pairwise overlap is generally reduced over time and active-element and neighborhood filtering avoid many unnecessary collision tests. Among the three collision-handling stages, the image-based narrow phase has the most dominant runtime and memory cost and is therefore the primary target for further optimization. Overall, the implementation provides a practical foundation for GPU-based collision handling on deformable polyhedral meshes and identifies several directions for future work.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
1.1 Problem Statement	2
1.2 Contributions	2
2 Related Work	5
2.1 Collision Detection	5
2.2 Broad Phase: Spatial Acceleration	6
2.3 Narrow Phase: Intersection Testing	8
2.4 Collision Resolution	11
2.5 Simulation Framework Background	13
3 Method	17
3.1 Problem Definition	17
3.2 Pipeline Overview	17
3.3 Preprocessing	19
3.4 Collision Candidate Pair Generation	22
3.5 Pair-Local Intersection Detection	22
3.6 Pair-Local Collision Resolution	28
4 Implementation	31
4.1 Framework Integration	31
4.2 Representing Irregular Topology on GPU	32
4.3 Candidate Pair Buffer Construction	33
4.4 Custom CUDA Compute Rasterizer	36
4.5 Collision Response	40
4.6 Contribution Summary	41
5 Evaluation	43
	xv

5.1	Benchmark Scenes	43
5.2	Runtime	44
5.3	Persistent GPU Memory	48
5.4	Kernel Resources	51
5.5	Pairwise Overlap Volume	52
5.6	Narrow-Phase Accuracy	54
5.7	Narrow-Phase Resolution	57
5.8	Confirmed Collisions Ratio	58
5.9	Friction Validation	59
5.10	Approximate Collision Normals	60
5.11	Summary	61
6	Conclusion	63
6.1	Future Work	64
	Overview of Generative AI Tools Used	67
	List of Figures	69
	List of Tables	71
	List of Algorithms	73
	Bibliography	75
	Appendix A: Notes on the Virtual Element Method	81



Introduction

The simulation of deformable objects is a fundamental problem in computer graphics, with applications ranging from physics-based animation in films and games to scientific computing and medical simulation. Realistic behavior in such simulations depends not only on accurately modelling the internal elastic response of a single object, but also on detecting the interactions between multiple bodies as they come into contact. Without these detected collision, simulated objects pass through one another, breaking the illusion of physical plausibility and limiting the range of applications.

A common approach to simulating deformable objects is to discretize their volume into finite elements and to approximate internal elastic forces over this discretization in order to solve the resulting equations of motion. The Finite Element Method (FEM) is one of the most widely adopted techniques for this purpose. However, common volumetric FEM formulations typically use elements such as hexahedra or tetrahedra with known shape functions, but their fixed element topology limits the flexibility of the resulting discretization. The Virtual Element Method (VEM) lifts this restriction by supporting more general polyhedral elements, including meshes with hanging nodes or T-junctions, without requiring the corresponding shape functions to be constructed explicitly, while still providing an approximation of elastic deformation [XPW24]. This added flexibility comes at the price of complexity. Polyhedral elements can be non-convex, may deform such that originally planar faces are no longer planar, and are far less studied than their tetrahedral counterparts in the context of simulation and contact handling.

The simulation framework that forms the foundation of this thesis is a prototype developed by the Research Unit of Computer Graphics at TU Wien. At the beginning of this work, the framework supported the simulation of deformation of polyhedral elements via VEM on the GPU using CUDA [Fü26], but it did not detect collisions between those elements. As a consequence, elements could interpenetrate freely, without the resulting overlaps being resolved in a visually plausible way. Addressing this limitation with a particular focus on a parallel GPU-based implementation, is the main objective of this thesis.

1.1 Problem Statement

Adding collision handling between elements to the existing simulation framework is non-trivial for several reasons. Since the underlying simulation runs on the GPU, the collision handling pipeline should also execute on the GPU. Otherwise, memory transfers between CPU and GPU would dominate the runtime making it orders of magnitude slower than a fully GPU-based pipeline.

The first challenge is the number of possible element pairs. A naive all-pairs collision test scales quadratically with the number of elements and quickly becomes infeasible. A so called broad phase collision detection, using spatial acceleration structures, is therefore employed to prune pairs that cannot intersect based on their bounding volumes. However, because the simulated geometry deforms over time, these data structures must be efficient to build or update on the GPU during the simulation.

The second challenge arises in the narrow phase collision detection, which further examines the candidate pairs produced by the broad phase by determining whether bounding volume overlaps correspond to actual intersections of the underlying elements. In this thesis, this step is complicated by the fact that the considered elements are general polyhedra rather than tetrahedra. Many established geometric narrow phase methods are designed for simpler or more limited representations, and applying them to general polyhedral elements requires additional preprocessing, bookkeeping, or a different algorithmic formulation. All together, this problem becomes more difficult because deformation can turn an initially convex element into a non-convex one. Geometric intersection tests such as the Separating Axis Theorem [GLM96] or the Gilbert-Johnson-Keerthi algorithm [GJK88] are restricted to convex shapes, and extending them to non-convex geometry typically requires convex decomposition and mapping the results back to the original elements, introducing substantial complexity and runtime overhead.

Finally, detected intersections must be resolved in a way that integrates cleanly with the existing solver. The collision resolution should resolve interpenetration and produce visually plausible results without destabilizing the simulation.

Taken together, these requirements define the research question of this thesis: how to design and implement a collision detection and resolution pipeline for deformable, potentially non-convex polyhedral meshes that runs entirely on the GPU and integrates with an existing VEM-based simulation framework.

1.2 Contributions

This thesis investigates a strategy for collision detection and collision resolution on deformable, potentially non-convex polyhedral meshes on GPU. It studies how broad phase acceleration, narrow phase detection, and collision resolution can be combined in the context of a VEM-based simulation, and presents a concrete pipeline that addresses the challenges outlined above. Our implementation extends the existing C++ and

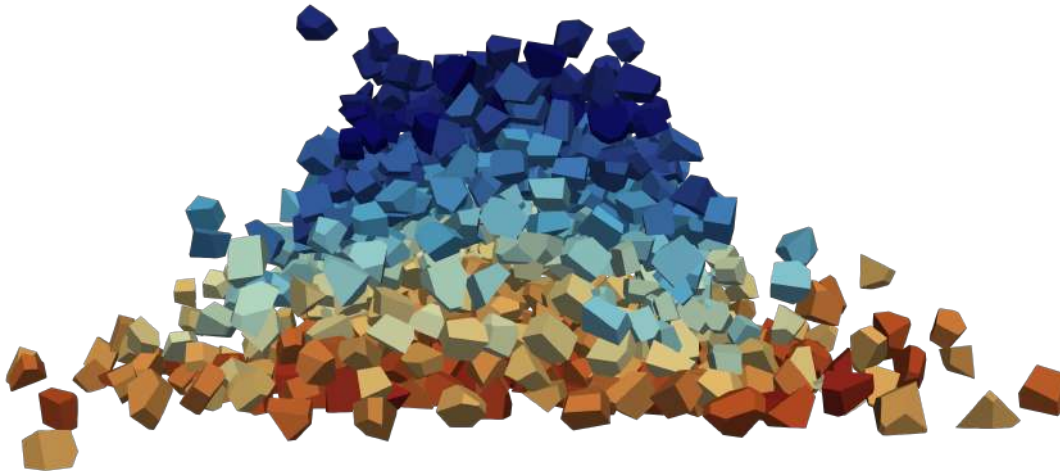


Figure 1.1: Simulation of a stack of five cuboid meshes, subdivided into 500 elements under full fracture.

CUDA VEM framework, which uses GeDiM [BBTV25] and PolyDiM [BBTV26] for polyhedral mesh handling and discretization. For the collision pipeline, we additionally use NVIDIA’s cuBQL [Wal26] for BVH construction and traversal and Thrust [NVI26] for device datastructures. The individual contributions, together with their related sections, are:

- A broad phase, discussed in Sections 3.4 and 4.3, that generates candidate collision pairs by traversing a bounding volume hierarchy over per-element axis-aligned bounding boxes. At the beginning of each time step, we recompute the bounding boxes in parallel and rebuild the hierarchy. During the subsequent solver iterations, we only refit the hierarchy, since updating the existing node bounds is cheaper than rebuilding the entire data structure. To further reduce unnecessary narrow phase invocations, a precomputed active-element classification excludes elements located in the interior of an object, while topological adjacency lists reject pairs of connected neighboring elements from being generated.
- A narrow phase, introduced in Sections 3.5 and 4.4, detecting intersections between deformable, potentially non-convex polyhedral elements. It builds on the image-based formulation of Faure et al. [FBAF08], but replaces the usage of a graphics API for rasterization with a custom CUDA rasterizer. The method triangulates the polygonal faces of each polyhedron, using centroid-fan triangulation, and for every candidate pair produces the overlap volume, its gradient with respect to the vertex positions and collision normals.
- A collision resolution stage, discussed in Sections 3.6 and 4.5, that uses the overlap volume and its gradient produced by the narrow phase to reduce interpenetration

between polyhedral elements. The method computes position-based corrections along the volume-gradient direction and propagates the resulting contact and friction forces into subsequent solver iterations.

- An evaluation of the pipeline, shown in Chapter 5 in terms of detection accuracy, runtime, persistent GPU memory, kernel execution resources, pairwise overlap reduction, and broad phase candidate confirmation ratio. The evaluation also studies the runtime impact of the image-based resolution and validates the behavior of friction and the image-based collision-normal approximation.

Our final implementation enables the resolution of collisions between polyhedral elements while taking advantage of the GPU's parallelism and throughput. More broadly, we demonstrate that image-based collision handling, developed for triangle surface meshes, can be combined with a polyhedral VEM simulation in a computationally efficient way. Figure 1.1 shows a representative simulation in which collisions of a full fractured stack of meshes are handled by our pipeline.

Related Work

Collision handling has been an important part of physics-based simulation since its earliest days and has remained one of its most computationally demanding components. As surveys of the field consistently observe, it is frequently the bottleneck of a simulation, and the requirements of accuracy and speed stand in contrast with one another [TKH⁺05, AEF22, BET14]. Early work focused predominantly on rigid bodies, where substantial parts of the data structures used for acceleration could be precomputed once and reused across the simulation with minimal cost of updating [KHI⁺07]. This assumption becomes less suitable for deformable objects, since their geometry changes over time and therefore also the relationships between vertices, primitives and elements over the course of the simulation. Bounding volumes must be refitted or rebuilt every iteration and spatial acceleration structures must account for the continuously changing geometry [WC21]. Within this evolution, a development particularly relevant to the present work is the migration of increasingly many parts of collision detection onto the GPU. This shift is driven both by the throughput available on modern graphics hardware and by the observation that many collision detection operations, such as constructing acceleration structures and performing bounding volume or primitive-level tests, map well onto the parallel execution model.

With this background in mind, the remainder of this chapter explores work related to collision detection and collision resolution, followed by a brief introduction of the simulation framework underlying this thesis.

2.1 Collision Detection

Collision detection is most commonly decomposed into two stages. The first stage, the broad phase, reduces the inherently quadratic number of pairwise tests through culling element pairs that cannot possibly intersect by performing cheap intersection tests on bounding volumes organized in a spatial acceleration structure [WC21]. The second

stage, the narrow phase, then consumes the generated candidate pairs and determines whether an actual intersection occurs based on the object geometry.

A further distinction between collision detection approaches can be drawn according to whether collisions are detected only at discrete timesteps, referred to as discrete collision detection (DCD), or whether the motion between timesteps is modelled as a continuous trajectory and tested for intersections along its path, referred to as continuous collision detection (CCD). CCD addresses the problem of tunnelling, in which fast-moving objects pass through one another between timesteps without their intersection ever being detected [TMY⁺11]. The narrow phase method adopted in this work is based on image-space rasterization, which evaluates the scene configuration at a single point in time and is therefore inherently discrete [FBAF08]. Accordingly, this work operates in the realm of DCD, and CCD methods are not discussed in detail.

2.2 Broad Phase: Spatial Acceleration

A range of spatial acceleration structures has been proposed for the broad phase, each making different trade-offs between construction cost, query efficiency, memory consumption, and suitability for deforming geometry. The following subsections review three commonly used families of approaches, namely bounding volume hierarchies, spatial partitioning, and sweep-and-prune methods.

2.2.1 Bounding Volume Hierarchies

A bounding volume hierarchy (BVH) wraps the geometric elements of a scene in bounding volumes (BVs) and organizes them in a tree structure, so that intersection queries can be computed by recursively descending only into branches whose BVs overlap [WC21]. Figure 2.1 illustrates this hierarchical organization of the AABBs of a small cube mesh discretized into 20 elements. Common bounding volume types include spheres [Hub95], axis-aligned bounding boxes (AABBs) [vdB98], oriented bounding boxes (OBBs) [GLM96], discrete oriented polytopes (k-DOPs) [KHM⁺98], and convex hulls [EL01], each offering a different trade-off between tightness of fit, construction complexity and computational cost of the overlap test.

The general issue that acceleration structures cannot be precomputed once and reused for deforming geometry becomes apparent in BVHs as a specific trade-off. Either the tree can be fully rebuilt every frame or iteration, or the existing hierarchy can be refitted to newly enclose the deformed geometry. In a refit, the bounding volumes inside the hierarchy are updated while the tree topology remains unchanged. This preserves intersection-test correctness, since internal nodes still fully enclose their children, but the tree topology, staying the same, may become less efficient as the geometry deforms, leading to larger or more strongly overlapping internal bounding volumes and reduced culling efficiency. A significant body of research has addressed this efficiency problem by adapting BVH build and traversal algorithms to GPU architectures. Lauterbach

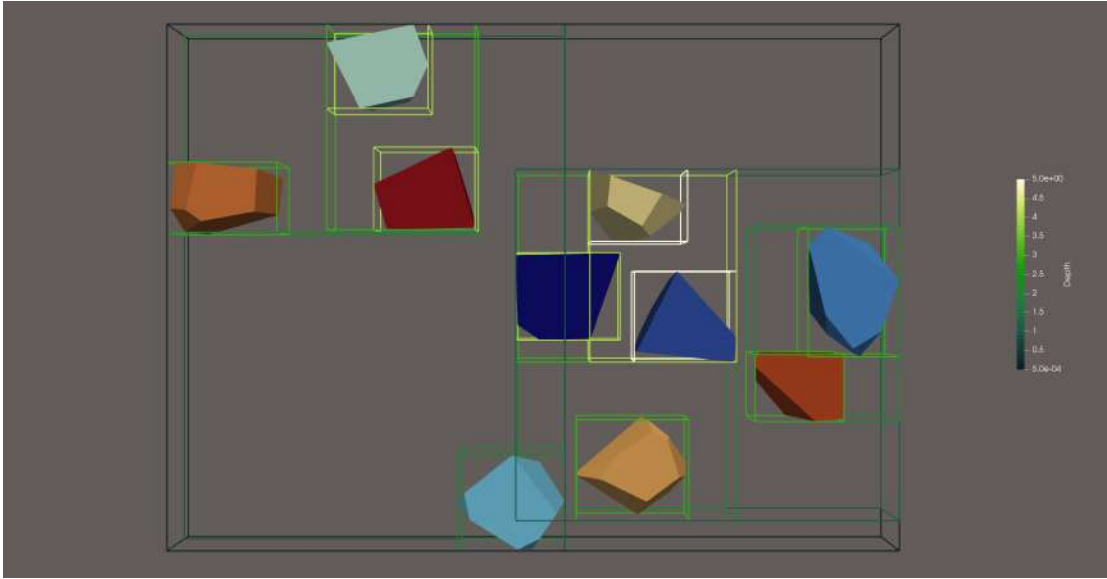


Figure 2.1: A bounding volume hierarchy constructed over 10 AABBs of Voronoi elements. The BVH contains 19 nodes across five hierarchy levels. Dark green represents the root node, with progressively lighter shades indicating increasing depth and white representing the deepest level in the hierarchy.

et al. [LMM10] use Morton codes to generate linear-time BVHs efficiently in parallel, while Wang et al. [WTMT18] apply local restructuring to preserve tree quality under deformation benefiting traversal speed. Despite these improvements, BVHs still incur non-trivial rebuild costs in deformable settings, where the acceleration structure must be updated or regenerated periodically.

2.2.2 Spatial Partitioning

Spatial partitioning subdivides space into cells and tests for collisions only between primitives that share a cell [GI93]. The simplest variant is a uniform grid, where every cell has the same size, and primitives are inserted into all cells they intersect [KGS98]. Hierarchical and adaptive variants such as octrees [WLZ14] or k-d trees [SR19] adapt this idea to better handle scenes with non-uniform primitive sizes, and spatial hashing avoids the memory cost of a dense grid by mapping cell coordinates into a hash table [WDZ17]. Grids are particularly attractive on the GPU since construction is linear in the number of primitives, the work is easily parallelizable and queries do not require traversal of an irregular tree.

The main difficulty is choosing an appropriate cell size, since cells that are too small produce many redundant insertions, while cells that are too large lose their pruning power. A range of approaches address this at the cost of additional bookkeeping, including adaptive and virtual subdivisions [WLZ14] as well as hierarchical grids combined with

hybrid hashing schemes [WDZ17]. A second cost inherent to grid-based schemes is that a primitive overlapping multiple cells generates a candidate pair multiple times, requiring a deduplication step before the narrow phase. Spatial Partitioning therefore offer favorable construction characteristics on the GPU, but at the price of non-trivial bookkeeping and a runtime-memory trade-off [WC21].

2.2.3 Sweep and Prune

The sweep-and-prune (SaP) approach reduces the three-dimensional overlap test to three one-dimensional ones by projecting AABBs onto the coordinate axes and detecting overlaps via the sorted order of interval endpoints [CLMP95]. Two bounding boxes overlap if and only if their projected intervals overlap on all three axes. The expected complexity depends on temporal coherence. When successive frames are similar, the previous sort order is nearly correct, and incremental insertion sort restores it in $O(n + s)$ time, where n is the number of primitives and s the number of swaps required. When coherence is poor, s can grow large enough that fully resorting with quicksort at $O(n \log n)$ becomes preferable [WC21]. The efficiency of sweep-and-prune therefore strongly depends on temporal coherence between frames. This assumption can still hold for mildly deforming objects, but it becomes less reliable when deformation changes the extents of objects significantly. In contrast to rigid-body simulation, where all points of an object undergo the same rigid transformation and its shape remains unchanged, deformable objects may change their shape and therefore their bounding volumes due to individual vertex movement. As a result, the sorted endpoint lists used by SaP may require more updates, reducing the benefit of incremental sorting. The number of swaps required to maintain the sort order grows almost quadratically with the number of objects in dynamic scenes [TBW09]. GPU adaptations and parallel reformulations have been proposed to mitigate these costs [CL16], but SaP remains most effective in settings with high spatial coherence between frames, and its advantages diminish as the geometry dynamically deforms [WC21].

Based on the trade-offs mentioned in the sections above, our work adopts a BVH-based broad phase that rebuilds the hierarchy at the beginning of each timestep and refits it during the subsequent solver iterations. In contrast to spatial partitioning, this choice does not require selecting a spatial resolution or removing duplicate pairs caused by elements spanning multiple cells and hierarchy levels. It also avoids relying on the endpoint ordering required by sweep-and-prune, whose maintenance can become costly under substantial deformation. The resulting broad-phase method and its GPU implementation are described in Sections 3.4 and 4.3, respectively.

2.3 Narrow Phase: Intersection Testing

Once the broad phase has produced a set of candidate pairs, where a collision could occur, the task of the narrow phase is to determine whether the underlying elements, enclosed by the bounding volumes, actually intersect. The methods used for this task

differ in several aspects, namely the accuracy guarantees, how geometry is represented, and the support for non-convexity of elements. The following subsections review three classes of methods prominent in the literature: geometric methods, volumetric methods, and image-space methods.

2.3.1 Geometric Methods

Geometric narrow phase methods operate directly on surface primitives, computing intersections through tests such as vertex-triangle, edge-edge and triangle-triangle intersection. Popular, efficient, and well-researched algorithms for determining intersections between two convex objects are the Separating Axis Theorem (SAT) [GLM96] and the Gilbert-Johnson-Keerthi algorithm [GJK88].

The polyhedral elements considered in this work are not convex in general. Even if the initial mesh is convex, deformation over the course of the simulation can introduce non-convexity. Both SAT and GJK are fundamentally designed for convex geometry [GLM96, GJK88], and therefore require convex decomposition when applied to non-convex meshes. For deformable objects, this decomposition must either be recomputed repeatedly as the geometry changes or risks becoming invalid as the mesh deforms. In addition to the computational cost, intersections detected on the decomposed submeshes must subsequently be mapped back to the original polyhedral elements, introducing additional complexity.

Triangle-triangle intersection tests do not require convexity of the underlying geometry and have been used effectively for deformable cloth and surface meshes, but they primarily compute surface intersections [WDZ17]. Consequently, deriving overlap volumes and their gradients, which are useful for collision resolution, becomes considerably more involved. Geometric methods are therefore effective for convex primitives, but their computational complexity and implementation difficulty increase significantly for deformable, non-convex polyhedral geometry.

2.3.2 Volumetric Methods

Volumetric methods represent collision geometry through spatial fields or sampled volumes rather than through explicit surface primitives. A common formulation is the signed distance field (SDF), which represents a shape by storing the signed Euclidean distance to the closest surface, with the sign distinguishing inside from outside. Collisions can then be detected by sampling the SDF of one object at points of another object. Since the sampled distance directly provides an estimate of penetration depth and the field gradient gives an approximate contact normal, SDFs are attractive for collision response, particularly in cases involving complex surface geometry [WC21].

These advantages come at the cost of constructing and maintaining a volumetric representation, which is particularly problematic for deformable objects. Since the distance field must be recomputed or updated whenever the geometry changes, the method can be expensive in both runtime and memory. Uniform voxel grids are simple to query

but memory intensive, while adaptive variants such as adaptively sampled distance fields [FPRJ00, PF01] reduce memory consumption at the cost of more complex construction and access. GPU-based dynamic SDF construction can reduce the computational runtime cost [LK13], but does not eliminate the underlying problem of updating the representation as the object deforms.

This trade-off is poorly suited to the setting considered in this thesis, where polyhedral elements deform over the duration of the simulation. Maintaining sufficiently accurate volumetric representation for all elements would therefore introduce substantial memory consumption and the discretization resolution would limit the accuracy for thin features and sharp geometric details.

2.3.3 Image-Space Methods

Image-space methods solve the problem of collision detection by projecting the objects into an image-space representation. Triangles, representing the surface of the object, are rasterized into 2D image-space textures from one or more viewing directions and intersections are detected by analyzing per-pixel depth information. Image-space methods typically require little to no preprocessing, particularly when the input geometry is already represented as a watertight triangle mesh. Their independent processing of pixels also maps efficiently onto graphics hardware [WC21].

The line of work most relevant here begins with pair-depth-buffer methods, which stored minimum and maximum depth values per pixel but were limited to convex objects [SF91, BWS98]. Myszkowski et al. [MOK95] extended this idea to concave objects by introducing stencil-buffer counts of fragment entries and exits, while Heidelberger et al. [HTG03, HTG04] later introduced Layered Depth Images (LDIs), which store an ordered stack of depth fragments per pixel to capture multiple surface layers along each viewing ray. Faure et al. [FBAF08] extended the LDI approach to the GPU, eliminating the frame-buffer readback needed for CPU sided overlap computations, which had been the dominant cost in earlier image-space methods. In addition to detecting intersections, the method also computes the intersection volume and its gradient with respect to vertex positions. Research on image-space collision detection was most active in the late 2000s and early 2010s, when programmable graphics hardware became increasingly suitable for general-purpose computation. More recent surveys note that this line of work has developed comparatively slowly in recent years [WC21].

Nevertheless, the properties of image-based methods align well with the requirements of this thesis: the method avoids convex decomposition, supports potentially non-convex closed surfaces, requires little preprocessing, and directly produces volumetric collision information that can be used for collision resolution. Similar to volumetric methods, however, its accuracy is limited by the resolution of the discretization. In the image-space case, the finite raster resolution determines how accurately thin features and small overlap regions can be captured. Based on these trade-offs, this work adopts the image-based overlap volume and volume gradient formulation of Faure et al. [FBAF08] as the basis of

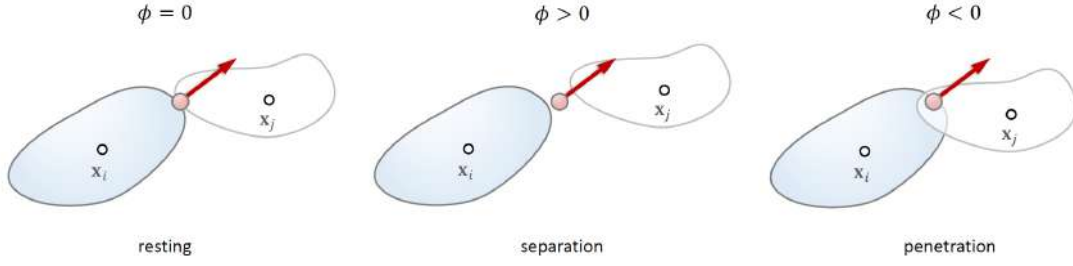


Figure 2.2: The three relationships between two bodies expressed through the value of the gap function ϕ , taken from Andrews et al. [AEF22]

its narrow phase. Its adaptation to the polyhedral elements, considered in this thesis, and the custom CUDA implementation are described in Sections 3.5 and 4.4, respectively.

2.4 Collision Resolution

After the narrow phase has identified intersecting objects, the collision resolution stage computes the response of the simulation. This response aims to remove existing interpenetration, prevent further penetration in subsequent timesteps, or compute contact forces that are fed back into the solver. In contrast to collision detection, which is primarily geometric, collision resolution is often coupled to the underlying time integration and solver formulation. Since this thesis focuses on deformable objects and an iterative solver that already performs position-level projections, the following discussion concentrates on the most relevant formulations for this setting, namely constraint-based methods, penalty-based methods, and position-based methods.

2.4.1 Constraint-Based Methods

Constraint-based methods formulate non-penetration as a unilateral constraint. The relationship between two bodies is commonly represented using a gap function ϕ , where $\phi > 0$ indicates separation, $\phi = 0$ corresponds to contact, and $\phi < 0$ indicates penetration, see Figure 2.2. The normal contact force or impulse is represented by a non-negative Lagrange multiplier λ_n . Ideal non-penetration can therefore be expressed by the complementarity condition

$$\phi \geq 0 \wedge \lambda_n \geq 0 \wedge \phi \lambda_n = 0. \quad (2.1)$$

This states that contact forces may only push objects apart and may be non-zero only when the gap constraint is active [AEF22]. Equivalently, if the bodies are separated, the contact force must be zero and if a contact force is present, the bodies must be exactly in contact.

In practice, many constraint-based solvers determine the contact forces or impulses required to prevent the bodies from moving farther into one another. Since the response

at one contact can influence the others, the unknown contact multipliers are commonly solved together in a linear complementarity problem (LCP). The advantage of constraint-based methods is that they can enforce non-penetration more rigorously than penalty forces. However, this requires solving a coupled problem over all active contacts. For deformable objects, the number of degrees of freedom is large and contact constraints may involve many vertices, making the construction and solution of this contact system expensive [AEF22].

2.4.2 Penalty and Barrier Methods

Penalty-based methods take a different approach by applying repulsive forces when objects interpenetrate. Instead of solving for a constraint multiplier, contact is modeled as a spring-like force that grows with the magnitude of the constraint violation [AEF22]. In the volume-based formulation of Faure et al. [FBAF08], the violation is measured by the overlap volume $\mathcal{V}$. A potential penalty energy can be defined as

$$E^{\text{pen}} = \frac{1}{2}k\mathcal{V}^2, \quad (2.2)$$

where k is a stiffness parameter. The resulting force is obtained by differentiating the potential energy with respect to the vertex coordinates,

$$\mathbf{f}^{\text{pen}} = -\frac{\partial E^{\text{pen}}}{\partial \mathbf{x}} = -k\mathcal{V}\frac{\partial \mathcal{V}}{\partial \mathbf{x}}, \quad (2.3)$$

where $\mathbf{x}$ is the vector of vertex coordinates. Penalty methods are attractive because the generated forces can be added directly to the equations of motion together with elastic and external forces. Their main drawback is the dependence on the penalty stiffness k : if it is too low, visible penetration remains, if it is too high, the system becomes stiff and may require smaller timesteps or more expensive implicit integration [Bar89].

Augmented Lagrangian methods address this trade-off by combining a penalty stiffness with a multiplier that represents the accumulated constraint force. Instead of enforcing the constraint through a large penalty parameter alone, the multiplier is updated according to the remaining constraint violation in each iteration. This allows the constraint force to build up iteratively while avoiding some of the numerical stiffness associated with a large penalty parameter [GDY25].

Barrier methods are closely related, but instead of applying a penalty to existing penetration, they use potentials that grow rapidly the closer the gap function approaches zero [AEF22]. A prominent recent advancement is Incremental Potential Contact (IPC), which incorporates barrier potentials into an optimization-based implicit time integration framework. This can prevent objects from interpenetration, but also requires numerical tuning since the resulting optimization problem becomes increasingly stiff near contact [LFS⁺20].

2.4.3 Position-Based Methods

Position-based methods resolve constraint violations by directly modifying positions. Instead of computing contact forces or impulses, the approach computes positional corrections that move the system toward a valid configuration. This type of formulation is central to position-based dynamics, where constraints are solved on a position-level [MHHR07]. Consider a constraint $C(\mathbf{p}_i) = 0$, where $\mathbf{p}_i$ denotes the position of vertex i . If the current configuration violates this constraint, such that $C(\mathbf{p}_i) \neq 0$, a typical projection step applies a correction $\Delta\mathbf{p}_i$ along the constraint gradient,

$$\Delta\mathbf{p}_i = -\frac{C(\mathbf{p}_i)}{\|\nabla_{\mathbf{p}_i} C(\mathbf{p}_i)\|^2} \nabla_{\mathbf{p}_i} C(\mathbf{p}_i). \quad (2.4)$$

The correction is only applied when the constraint is violated. Position-based methods are attractive for interactive simulations because they are robust, simple to implement, fast, and often produce visually plausible results even when solved only approximately. They are therefore well suited to simulation frameworks that already enforce other constraints at the position level.

Their main limitation is that purely positional corrections do not directly correspond to physically meaningful forces. They may reduce or remove penetration, but the resulting motion can depend on solver iteration count, projection order, and correction step size. For this reason, several extensions move position-level corrections more closely to constraint forces. Extended Position-Based Dynamics (XPBD) introduces Lagrange multipliers, making the constraint response less dependent on timestep and iteration count [MMC16]. In this thesis instead we combine a position-based collision correction with ALM-like force feedback: we convert the applied positional correction into forces and feed them back into subsequent solver iterations, as described in Section 3.6.2.

2.5 Simulation Framework Background

The simulation framework used in this thesis is based on the Virtual Element Method (VEM), a generalization of the Finite Element Method (FEM) that supports polygonal and polyhedral elements without requiring a tetrahedralization of the simulation domain. This is particularly useful for deformable objects represented by general volumetric cells, since the method can operate directly on elements with arbitrary numbers of vertices and faces. In contrast to FEM, the basis functions of a virtual element are not explicitly evaluated inside the element. Instead, the deformation is approximated through projection operators that recover the polynomial part of the deformation from boundary information [WRRH17].

2.5.1 Deformation Gradient

In the formulation underlying this work, each element is described by a linear approximation of its deformation. For an element $\mathcal{E}$, this approximation is represented by the

average deformation gradient $\mathbf{F}_\mathcal{E}$. Let $|\mathcal{E}|$ denote the element measure, i.e. its area in 2D and its volume in 3D. Furthermore, let $\mathbf{x}$ denote a material-space position, $\mathbf{X}$ its world-space position, $\mathbf{n}$ the outward normal on the boundary $\partial\mathcal{E}$, and $\tilde{\mathbf{x}}$ the boundary integration variable on $\partial\mathcal{E}$. Using the divergence theorem, the average deformation gradient can be written as

$$\mathbf{F}_\mathcal{E} = \frac{1}{|\mathcal{E}|} \int_\mathcal{E} \frac{\partial \mathbf{X}}{\partial \mathbf{x}} d\mathbf{x} = \frac{1}{|\mathcal{E}|} \int_{\partial\mathcal{E}} \mathbf{X}(\tilde{\mathbf{x}}) \mathbf{n}^\top d\tilde{\mathbf{x}}. \quad (2.5)$$

For a polyhedral element in three dimensions, this boundary integral can be discretized as a sum over the element faces,

$$\mathbf{F}_\mathcal{E} = \frac{1}{|\mathcal{E}|} \sum_{f \in \mathcal{F}(\mathcal{E})} a_f \bar{\mathbf{X}}_f \mathbf{n}_f^\top, \quad (2.6)$$

where $\mathcal{F}(\mathcal{E})$ is the set of faces of $\mathcal{E}$, a_f is the rest-area of face f , $\mathbf{n}_f$ is its outward rest-normal, and $\bar{\mathbf{X}}_f$ is the average deformed position of the face. To compute these face averages, the rest-space vertices of each face are first mapped to a local two-dimensional coordinate system centered at the rest-space face centroid. Let (x_i, y_i) denote the local coordinates of the i -th vertex of face f . A linear function

$$\hat{\mathbf{X}}_f(x, y) = \mathbf{a}_f + \mathbf{b}_f x + \mathbf{c}_f y, \quad (2.7)$$

with $\mathbf{a}_f, \mathbf{b}_f, \mathbf{c}_f \in \mathbb{R}^3$, is fitted in a least-squares sense such that

$$\hat{\mathbf{X}}_f(x_i, y_i) \approx \mathbf{X}_i \quad (2.8)$$

for all vertices i of face f , where $\mathbf{X}_i$ is the deformed vertex position in three dimensions. Since the local origin coincides with the rest-space face centroid, evaluating this fitted function at the origin gives the constant coefficient $\mathbf{a}_f$. This coefficient is therefore used as the average deformed face position $\bar{\mathbf{X}}_f$. Since these face averages are linear combinations of the element's vertex positions, all geometric coefficients depending on the rest shape can be precomputed. The deformation gradient can therefore be evaluated during simulation by the matrix-vector product

$$\tilde{\mathbf{F}}_\mathcal{E} = \mathbf{G}_\mathcal{E} \mathbf{u}_\mathcal{E}, \quad (2.9)$$

where $\tilde{\mathbf{F}}_\mathcal{E}$ is the vector-unrolled form of $\mathbf{F}_\mathcal{E}$, $\mathbf{u}_\mathcal{E}$ refers to the vector formed by stacking all vertex displacements of element $\mathcal{E}$ and $\mathbf{G}_\mathcal{E}$ is the precomputed deformation-gradient operator [WRRH17].

2.5.2 Material Model

Once the average deformation gradient has been computed, a hyperelastic material model is evaluated in a similar manner to finite elements. For an energy density $\Psi(\mathbf{F}_\mathcal{E})$, the elastic deformation energy $E_\mathcal{E}$ of an element is defined as

$$E_\mathcal{E} = |\mathcal{E}| \Psi(\mathbf{F}_\mathcal{E}). \quad (2.10)$$

Using the deformation-gradient operator, the corresponding elastic force is defined as the negative gradient of this energy:

$$\mathbf{f}_\mathcal{E} = -\nabla_{\mathbf{u}_\mathcal{E}} E_\mathcal{E} = -\left(\frac{\partial E_\mathcal{E}}{\partial \mathbf{u}_\mathcal{E}}\right)^\top = -|\mathcal{E}| \left(\frac{\partial \Psi}{\partial \tilde{\mathbf{F}}_\mathcal{E}} \frac{\partial \tilde{\mathbf{F}}_\mathcal{E}}{\partial \mathbf{u}_\mathcal{E}}\right)^\top = -|\mathcal{E}| \mathbf{G}_\mathcal{E}^\top \frac{\partial \Psi}{\partial \tilde{\mathbf{F}}_\mathcal{E}}. \quad (2.11)$$

Similarly, the tangent stiffness is defined as the negative Jacobian of the forces with respect to the degrees of freedom:

$$\mathbf{K}_\mathcal{E} = -J(\mathbf{f}_\mathcal{E}) = \frac{\partial^2 E_\mathcal{E}}{\partial \mathbf{u}_\mathcal{E}^2} = |\mathcal{E}| \mathbf{G}_\mathcal{E}^\top \frac{\partial^2 \Psi}{\partial \tilde{\mathbf{F}}_\mathcal{E}^2} \mathbf{G}_\mathcal{E}. \quad (2.12)$$

The framework uses an invertible Neo-Hookean material model to compute stress and stiffness terms [SGK18].

2.5.3 Stabilization Term

A consequence of using only the linear average deformation gradient is that not all deformation modes of a general polyhedral element can be captured. An element may deform in a non-linear way that leaves $\mathbf{F}_\mathcal{E}$ unchanged. Such modes would not produce elastic forces and therefore have to be regularized by a stabilization term.

The stabilizer used in this work penalizes the part of the element deformation that cannot be represented by a rigid translation or linear deformation modes, including stretching, shearing, and linearized rotation. Let $\mathbf{T}_\mathcal{E} = (\mathbf{I}_{d \times d} \ \mathbf{I}_{d \times d} \ \dots)$ denote the translation operator with $n_\mathcal{E}$ repeated identity blocks, where $n_\mathcal{E}$ is the number of vertices in element $\mathcal{E}$, and d is either 2 or 3 depending on the dimension of space. The concatenated operator

$$\bar{\mathbf{G}}_\mathcal{E} = (\mathbf{T}_\mathcal{E}^\top \ \mathbf{G}_\mathcal{E}^\top)^\top \quad (2.13)$$

measures both translation and linear deformation modes. Furthermore, let $\mathbf{D}_\mathcal{E}$ be the scaled monomial basis matrix of the element, constructed from rest-space vertex coordinates centered at the element centroid and scaled by an estimate of the element's diameter, defined by the largest distance between any node pair. The projection operator

$$\mathbf{P}_\mathcal{E} = \mathbf{D}_\mathcal{E} \left(\bar{\mathbf{G}}_\mathcal{E} \mathbf{D}_\mathcal{E} \right)^{-1} \bar{\mathbf{G}}_\mathcal{E} \quad (2.14)$$

maps world-space vertex locations to the closest location described purely by rigid translation and linear deformation. The stabilizer $\mathbf{S}_\mathcal{E}$ penalizes the residual, representing non-linear deformation modes that remain after this projection, defined by

$$\mathbf{S}_\mathcal{E} = (\mathbf{I} - \mathbf{P}_\mathcal{E})^\top (\mathbf{I} - \mathbf{P}_\mathcal{E}). \quad (2.15)$$

With stabilization factor α , the stabilization energy, force, and stiffness are

$$E_\mathcal{E}^{\text{stab}} = \frac{1}{2} \alpha \mathbf{u}_\mathcal{E}^\top \mathbf{S}_\mathcal{E} \mathbf{u}_\mathcal{E}, \quad \mathbf{f}_\mathcal{E}^{\text{stab}} = -\alpha \mathbf{S}_\mathcal{E} \mathbf{u}_\mathcal{E}, \quad \mathbf{K}_\mathcal{E}^{\text{stab}} = \alpha \mathbf{S}_\mathcal{E}. \quad (2.16)$$

Thus, rigid translations and linear deformation modes are preserved by the projection, while the remaining non-linear modes are penalized [WRRH17, Yu22].

2.5.4 Parallel Iterative Solver

The time integration is performed using the implicit second-order backward differentiation formula (BDF2). Let $\mathbf{u}$ denote the stacked vector of degrees of freedom, $\mathbf{M}$ the lumped mass matrix, subscripts denoting time steps of size h , and $\mathbf{f}(\mathbf{u})$ the forces from elasticity, stabilization, and external influences. The discrete equation of motion can be written as

$$\mathbf{f}(\mathbf{u}_{t+1}) = \frac{1}{4h^2} \mathbf{M} (9\mathbf{u}_{t+1} - \hat{\mathbf{u}}_t), \quad \hat{\mathbf{u}}_t = 24\mathbf{u}_t - 22\mathbf{u}_{t-1} + 8\mathbf{u}_{t-2} - \mathbf{u}_{t-3}. \quad (2.17)$$

A Newton solve would require assembling and solving a global sparse system. Instead, the simulation follows the structure of position-based and projective dynamics methods, where local updates and constraint projections are iterated directly on positions [MMC16, NOB16].

The mesh is represented in maximal coordinates, where vertices shared by multiple elements are duplicated such that each element $\mathcal{E}$ stores its own local degrees of freedom $\tilde{\mathbf{u}}_{\mathcal{E}}$ to enable parallelism. At the beginning of a time step, local unknowns are initialized from a ballistic prediction based on the BDF2 velocity

$$\tilde{\mathbf{u}}_{\mathcal{E}} = \left(\mathbf{u}_t + \gamma h \frac{3\mathbf{u}_t - 4\mathbf{u}_{t-1} + \mathbf{u}_{t-2}}{2h} \right)_{\mathcal{E}} \quad (2.18)$$

Each solver iteration first updates all elements independently. Using the local elastic and stabilization force $\mathbf{f}_{\mathcal{E}}$ and stiffness $\mathbf{K}_{\mathcal{E}}$, the element-local update is then approximated by

$$\tilde{\mathbf{u}}_{\mathcal{E}} \leftarrow \tilde{\mathbf{u}}_{\mathcal{E}} - \beta \operatorname{diag} \left(\frac{9}{4h^2} \mathbf{M}_{\mathcal{E}} + \mathbf{K}_{\mathcal{E}} \right)^{-1} \left(\frac{1}{4h^2} \mathbf{M}_{\mathcal{E}} (9\tilde{\mathbf{u}}_{\mathcal{E}} - \hat{\mathbf{u}}_{\mathcal{E}}) - \mathbf{f}_{\mathcal{E}} \right), \quad (2.19)$$

where $\beta < 1$ is the step size parameter. Following the Gauss-Seidel approach, only the diagonal of the local system matrix is inverted, which makes the update inexpensive and suitable for parallel execution.

After this element-local relaxation, duplicated vertices that correspond to the same physical mesh vertex generally no longer coincide. Mesh-connectivity is therefore enforced as a position-level constraint, where the positions of all local copies belonging to a physical mesh vertex are averaged and moved to this average position. Dirichlet boundary conditions and simple contact constraints, like floor collisions, are enforced during the same projection stage. The solver repeats this element-local update and the constraint projection step until a fixed iteration count is reached or the position change falls below a set tolerance.

A more detailed mathematical derivation of the underlying simulation framework is provided in Appendix 6.1 [Hah26].

Method

This chapter describes the design of our collision handling method developed in this thesis. Our method extends an existing deformable VEM simulation on polyhedral elements with collision detection and collision resolution. This chapter focuses on the quantities computed by our method and on how the individual stages interact conceptually. We discuss concrete implementation details, including memory layouts, kernel optimizations, parallelism and other implementation-related design choices separately in Chapter 4.

3.1 Problem Definition

At each solver iteration, we consider one or more deformable objects $\mathcal{O} = (\mathcal{U}, \mathcal{F}, \mathcal{K})$, represented by mesh vertices $\mathcal{U}$, polygonal faces $\mathcal{F}$, and polyhedral elements $\mathcal{K}$. Each element $\mathcal{E} \in \mathcal{K}$ is defined by its faces and vertices, whose current positions $\mathbf{u}_{\mathcal{E}}$ determine the space occupied by the element. Mesh topology refers to these incidence relations, where two elements $\mathcal{E}_a$ and $\mathcal{E}_b$ are topologically adjacent if they share a face and therefore all vertices of that face. In the maximal-coordinate representation used by the solver, these shared vertices are duplicated so that each element stores its own local degrees of freedom (DoF); the mesh-connectivity constraint described in Section 2.5.4 couples the corresponding copies. Collision handling is then restricted to the set $\mathcal{K}_{\text{act}}$ of so called *active elements*, containing at least one boundary face, forming the outer layer of the object. Given the current vertex positions, the problem is to identify non-adjacent active element pairs whose spatial regions overlap and compute position corrections that reduce the overlap within the current configuration.

3.2 Pipeline Overview

Our proposed method organizes collision handling as a pipeline within the iterative simulation loop. The pipeline combines and extends existing collision handling concepts,

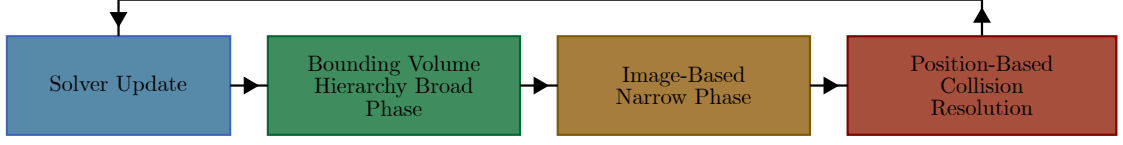


Figure 3.1: A high-level overview of one iteration of the collision handling pipeline. The solver updates the vertex positions, the bounding volume hierarchy broad phase generates candidate pairs, the image-based narrow phase determines element-level intersections, computes overlap volumes, gradients and collision normals. Finally, the collision resolution stage applies corresponding position-based corrections and propagates the resulting forces into the subsequent solver iteration.

discussed in Chapter 2, in a way that fits the polyhedral VEM setting of the simulation framework. We first reduce the set of possible contacts using precomputed topological information and spatial acceleration, explained in Sections 3.3 and 3.4, respectively. We then approximate the overlap of the remaining candidate pairs with an image-based narrow phase following Faure et al. [FBAF08], discussed in Section 3.5. Finally, we convert the resulting overlap information into a position-based correction, derived in Section 3.6.1, and feed resulting forces into the next solver iteration. The solver then continues with the next iteration from the corrected vertex positions. Figure 3.1 shows the high-level flow of the pipeline with its stages. The following paragraphs give a high-level overview of one pipeline iteration, starting from the updated element positions.

3.2.1 Bounding Volume Hierarchy Broad Phase

The solver computes updated vertex positions for all elements, we then use these positions to construct AABBs around the elements. At the beginning of each iteration, we construct or update a BVH, traverse the hierarchy, and generate collision candidate element pairs whose bounding boxes overlap. Since the mesh-connectivity constraint, introduced in Section 2.5.4, already connects neighboring elements of the same mesh, we use topological adjacency information to prevent neighboring candidate pairs from being created and subsequently avoid unnecessary narrow phase invocations. The details concerning the BVH broad phase, shown in Figure 3.1, are explained in the following Section 3.4.

3.2.2 Image-Based Narrow Phase

For each remaining candidate pair, the image-based narrow phase, introduced in Section 3.5, computes whether the two elements actually overlap. We decided on the image-based approach of Faure et al. [FBAF08] because established alternatives such as SAT [GLM96] and GJK [GJK88] apply only to convex primitives. Using them for the potentially non-convex elements considered here would require additional convex decomposition and mapping the resulting collision information back to the original polyhedral cells.

Our proposed approach avoids geometric element-element intersection tests and instead estimates overlap from rasterized surface triangles. For each candidate pair, we perform the following steps:

1. We represent the polygonal element faces by the precomputed centroid-fan triangulation described in Section 3.3.3.
2. We construct a pair-local view volume from the intersection of the element AABBs and orthogonally projects the triangulated surfaces along three perpendicular axes, as illustrated in Figure 3.2.
3. We generate surface fragments and store them in depth-sorted order for each pixel and projection direction.
4. We traverse each ordered fragment list to identify intervals in which the viewing ray lies inside both elements simultaneously.
5. We accumulate these intervals to approximate the overlap volume, its gradients with respect to the element vertex positions, and an approximate collision normal, discussed in Sections 3.5.4 and 3.5.5, respectively.

3.2.3 Position-Based Collision Resolution

The collision resolution stage, see Section 3.6.1, consumes these overlap volumes and gradients. Since the gradient indicates how the overlap volume changes under vertex movement, it provides a direction along which the vertices can be moved to reduce interpenetration. Our method reduces interpenetration through a position-based correction step that directly moves the element vertex positions along the overlap gradient and propagates resulting collision forces into the next solver iteration. This choice integrates nicely with the simulation framework, which already enforces constraints at the position level within the iterative solver loop.

With the overall pipeline established, we now describe its individual stages in detail, beginning with the topology-dependent preprocessing required before collision handling starts.

3.3 Preprocessing

Before the collision pipeline can operate on the element positions, it first precomputes topology-dependent information that does not change during simulation. The preprocessing stage collects this deformation-independent mesh topology data before the simulation starts. In particular, it restricts collision handling to the subset of elements that can participate in collisions located on the boundary of the object, constructs adjacency lists for filtering connected element pairs, and defines the centroid-fan triangulation used by the image-space narrow phase.

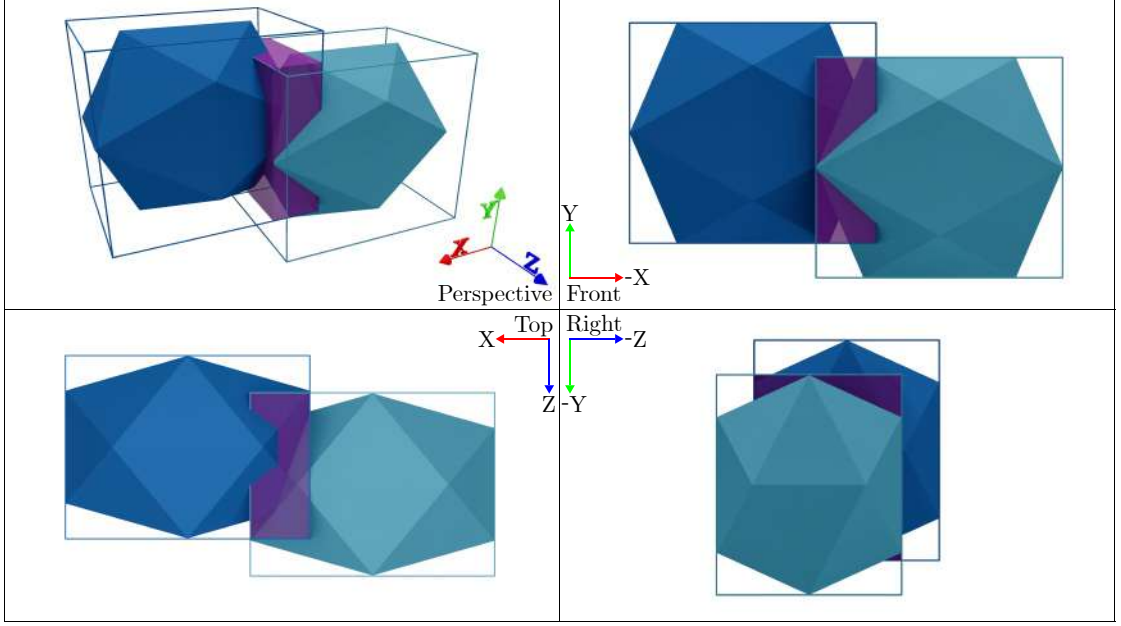


Figure 3.2: View volume and projection directions for one candidate element pair. The transparent purple cuboid marks the intersection of the two element AABBs and defines the pair-local view volume used by the narrow phase. The three orthogonal projection directions rasterize the elements into image space along the coordinate axes, with rasterization restricted to the view volume since overlap can only occur where both bounding boxes intersect.

3.3.1 Collision-Active Elements

Our method restricts collision detection to the active element set $\mathcal{K}_{\text{act}}$ defined in Section 3.1. These elements form the outer layer of the mesh and therefore contain the elements that can first come into contact with other elements of the same mesh or other objects in the simulation. A visualization of collision-active elements of a mesh can be seen in Figure 3.3a. This reduces the number of elements entering the broad phase and lowers the number of potentially colliding element pairs. The approach assumes that internal elements remain enclosed by neighboring elements and cannot participate in external contact before the surrounding boundary elements are passed. If the mesh topology changes, for example due to fracture, we update $\mathcal{K}_{\text{act}}$ accordingly.

3.3.2 Topological Adjacency

We precompute the topological adjacency information, defined in Section 3.1, to filter connected element pairs during broad phase traversal. Since adjacent elements are already connected by the mesh-connectivity constraint introduced in Section 2.5.4, their overlap does not represent meaningful contact for the simulation and should not be passed to the narrow phase. For this reason, the broad phase discards AABB overlaps between

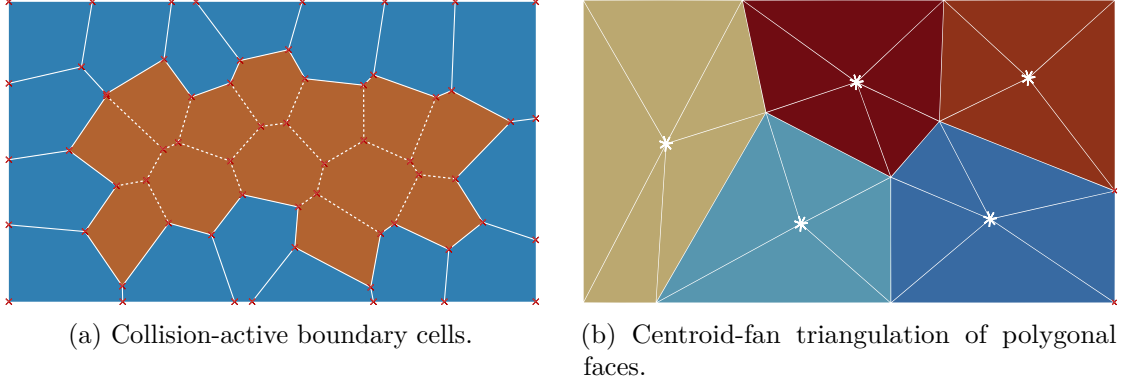


Figure 3.3: Preprocessing for the collision pipeline. On the left (a), boundary cells shown in blue are passed to the collision pipeline, while internal cells shown in orange are excluded to reduce computation in subsequent pipeline stages. On the right (b), each polygonal face is represented by a centroid-fan triangulation, with the face centroid indicated by a white star.

neighboring elements before generating candidate collision pairs.

3.3.3 Centroid-Fan Triangulation

The image-based narrow phase rasterizes surface triangles. Since the simulation mesh consists of polyhedral elements with polygonal faces, we convert each face into triangles using a centroid-fan triangulation.

Let the polygonal face f be defined by vertex indices $i = 0, 1, \dots, n-1$, where n is the number of vertices of f . For each vertex of a face, the preprocessing stage stores one fan triangle entry consisting of the face centroid $\mathbf{c}_f$, the vertex i , and the next vertex $(i+1) \bmod n$. Thus, a face with n vertices produces n triangles. During simulation, we construct these triangles every iteration using the current vertex positions. The face centroid $\mathbf{c}_f$ is computed as

$$\mathbf{c}_f = \frac{1}{n} \sum_{i=0}^{n-1} \mathbf{x}_i, \quad (3.1)$$

where $\mathbf{x}_i$ denotes the current position of the i -th vertex of face f .

In the rest configuration, the polygonal faces are assumed to be planar. Since we are working with potentially non-planar polygonal faces, the face geometry is not uniquely defined. Different triangulation strategies therefore produce different approximations of the deformed face geometry. We opt for a centroid-fan representation since it treats all face vertices symmetrically and avoids storing particular diagonals for each polygonal face. Its additional centroid vertex is not a simulation degree of freedom and can be mapped back to the original face vertices by distributing its contribution equally, as described in Section 3.5.4. An example of a centroid-fan triangulation on polygonal faces can be seen

in Figure 3.3b. With this topology-dependent data precomputed, the remaining stages operate on the current deformed element positions during each solver iteration.

3.4 Collision Candidate Pair Generation

At the beginning of each timestep, we construct a BVH over the current AABBs of all collision-active elements. During the following solver iterations within the same timestep, we update the AABBs from the deformed vertex positions and refit the existing hierarchy instead of rebuilding it from scratch. This is a compromise, since rebuilding the hierarchy every iteration would adapt both the bounding volumes and the tree topology to the current element positions, but it would also increase runtime cost. Reusing and refitting the hierarchy avoids these repeated rebuilds and updates the leaf and internal bounding volumes, but the fixed tree topology may become less effective at culling as the elements move and deform. The parallel AABB construction and the BVH build and refit implementation are described in Sections 4.3.1 and 4.3.2, respectively.

3.4.1 Hierarchy Traversal

Having constructed the BVH for the current state, we then traverse it to find overlapping AABBs. Each overlap between two collision-active elements defines a potential candidate pair for the narrow phase. To avoid duplicate pairs, we only accept one index ordering: for a detected overlap between elements a and b , we discard the pair if $a \geq b$. This prevents generating both (a, b) and (b, a) for the same AABB overlap. We also apply the topological adjacency filtering, introduced in Section 3.3.2, during traversal, where if two overlapping elements share a face, the pair is discarded since neighboring elements are already connected by the mesh-connectivity constraint and should not be treated as collision candidates. The result of this stage is a set of element pairs whose bounding boxes overlap and whose elements are not topological neighbors. The two-pass GPU traversal used to construct the candidate-pair buffer is detailed in Section 4.3.3.

3.5 Pair-Local Intersection Detection

After the broad phase has generated candidate pairs, the narrow phase determines whether the corresponding elements actually intersect. Our method follows the image-based approach by Faure et al. [FBAF08], where instead of testing individual geometric primitives for intersection directly, it rasterizes each triangulated candidate pair into image space from three perpendicular viewing directions and accumulates overlap intervals from the generated fragments. For each candidate pair, the narrow phase computes an approximate overlap volume $\mathcal{V}$, its gradient with respect to the element’s vertex positions $\partial\mathcal{V}/\partial\mathbf{x}$ and an approximate collision normal. These quantities are then passed to the collision resolution stage to handle the detected collision.

3.5.1 Restricted Rasterization Region

The narrow phase processes each candidate pair independently in parallel. For a candidate pair consisting of elements a and b , we first construct a view volume from the intersection of their AABBs, as visualized in Figure 3.2. This restricts rasterization to the region where overlap can occur.

We then orthogonally project each element pair along the three coordinate axes. For a projection direction dim , the remaining two axes, denoted by u and v , define the image plane, while dim defines depth. The extent of the AABB intersection defines the image-space domain for this view. We then discretize this domain into a fixed raster image of $r \times r$ pixels for every candidate pair and projection direction, where r is the selected narrow phase resolution. If h_u and h_v denote the side lengths of the AABB intersection along the image-plane axes, the world-space pixel area used by the later volume and gradient computations, explained in Sections 3.5.3 and 3.5.4 respectively, is

$$A_{\text{pixel},dim} = \frac{h_u h_v}{r^2}. \quad (3.2)$$

Each pixel thus corresponds to an axis-aligned column through the view volume along the depth direction dim . Since the input elements have polygonal faces, we use the centroid-fan triangulation described in the preprocessing stage 3.3.3 to represent the element surfaces during rasterization. For each projection direction, it projects these triangles into the corresponding image plane and clips their projected bounding rectangles to the rasterization region. The concrete implementation of the fixed resolution and its memory implications are discussed in Section 4.4.2. After this image-space domain has been defined, rasterization produces the per-pixel fragment data.

3.5.2 Depth Sorted Fragment Lists

Rasterization generates fragments for the pixels covered by the projected triangles. For each covered pixel, we compute barycentric coordinates of the fragment on the triangle and uses them to interpolate the depth along the current projection direction. Each fragment stores this depth value, the barycentric coordinates, the element to which the fragment belongs, and the triangle index that generated it. For every pixel, we sort these generated fragments by increasing fragment depth. The resulting depth-sorted fragment list (DSFL) represents the intersections of the viewing ray with the two element surfaces, as visualized on the left side of Figure 3.4.

3.5.3 Overlap Volume Computation

By traversing each pixel’s DSFL from first to last entry, our method, following Faure et al. [FBAF08], approximates overlap volumes, gradients and collision normals. During DSFL traversal, we record the depth at which the viewing ray first enters a region where it is inside both elements. When the ray leaves this shared inside region, we multiply the

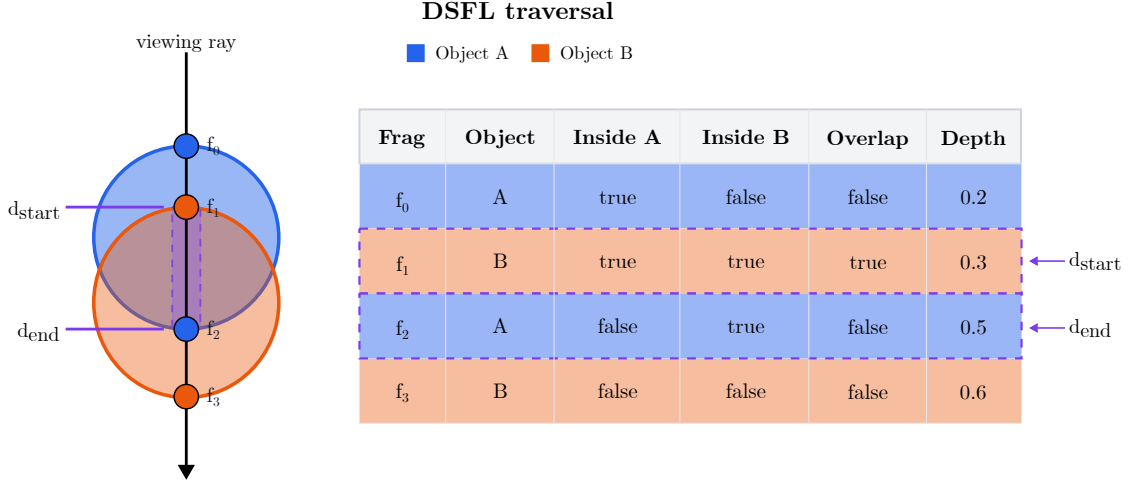


Figure 3.4: Simplified example of a depth sorted fragment list (DSFL) for one pixel ray. The left side shows a cross section of two intersecting elements and the fragments generated along the viewing ray. The table on the right illustrates the traversal of the DSFL, where each fragment toggles the inside state of the element that generated it. An overlap interval is detected when the ray is inside both elements simultaneously; its depth length, multiplied by the pixel area, gives the pixel’s overlap volume contribution.

length of the overlap interval by the pixel area. For one projection direction, the volume contribution of one pixel’s overlap interval is therefore

$$\Delta\mathcal{V} = (d_{\text{end}} - d_{\text{start}})A_{\text{pixel},\text{dim}}. \quad (3.3)$$

where d_{start} and d_{end} are the depths at which the overlap interval starts and ends, and $A_{\text{pixel},\text{dim}}$ is the world-space area represented by one pixel in the image plane along the current viewing direction dim . Figure 3.4 visualizes the process of DSFL traversal. For each projection direction, the pixel contributions are summed to obtain one directional volume estimate. The final pair volume is then computed as the average of the three directional estimates, i.e., their sum scaled by $1/3$. Figure 3.5 visualizes the resulting directional overlap textures for one example candidate pair.

3.5.4 Overlap Gradient Computation

The same traversal logic used for determining overlap volume simultaneously computes the gradient of the overlap volume with respect to the element vertex positions $\partial\mathcal{V}/\partial\mathbf{x}$ for both elements separately. The gradient describes how the measured overlap volume changes when individual vertices are moved. For each overlap interval, only the two fragments that bound the interval effect this change. The signs of their contributions follow directly from the interval length used in the volume computation, following Equation 3.3. If the fragment at the beginning of the interval is moved away from the camera along the current viewing direction, d_{start} increases, the interval becomes shorter,

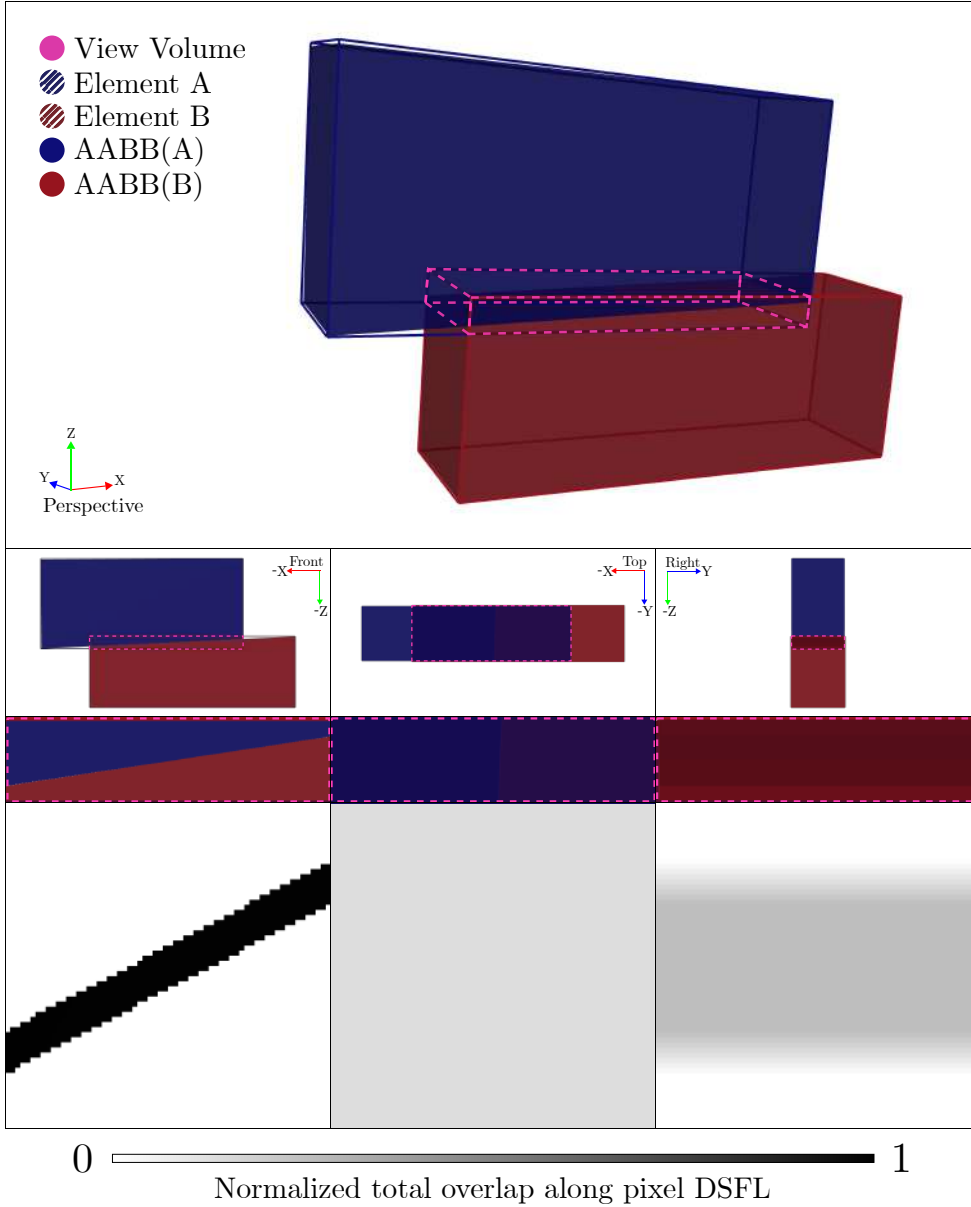


Figure 3.5: Overlap texture visualization for one intersecting element pair. Colored transparent objects denote the elements, wireframe boxes denote their AABBs and the purple dotted box the AABB-intersection view volume. From top to bottom, the rows show the perspective view, the three orthographic projection views, a zoom into the view volume, and the resulting 64×64 overlap textures, representing how much overlap was detected along a pixel's DSFL.

and the overlap volume decreases. This boundary therefore contributes with a negative sign. If the fragment at the end of the interval is moved away from the camera, d_{end} increases, the interval becomes longer, and the overlap volume increases. This boundary therefore contributes with a positive sign. Now to compute a fragment's contribution, its depth along the viewing direction dim is interpolated from the vertices of the triangle that generated it using barycentric coordinates. These barycentric coordinates, stored with the fragment, therefore indicate how much each triangle vertex contributes to the fragment depth.

In summary, consider one boundary fragment f of overlap interval k in pixel p generated by a triangle of element a . For triangle vertex i , with barycentric coordinate $\beta_{f,i}$, the contribution to the derivative of the interval volume $\Delta\mathcal{V}$ with respect to its coordinate along the current projection axis dim is

$$\frac{\partial \Delta\mathcal{V}_{p,k,dim}}{\partial x_{a,i,dim}} = s_f \beta_{f,i} A_{\text{pixel},dim}, \quad (3.4)$$

where $x_{a,i,dim}$ is the coordinate of vertex i along axis dim , and $s_f = -1$ for the interval start and $s_f = 1$ for the interval end. Thus, the projection along the x -axis contributes to the x -components of the gradient, while the projections along the y - and z -axes fill the y - and z -components.

The full gradient with respect to vertex i is formed by collecting the three axis component contributions from all boundary fragments in the list,

$$\frac{\partial \mathcal{V}}{\partial \mathbf{x}_{a,i}} = \sum_{dim \in \{x,y,z\}} \sum_{f \in \mathcal{F}_{a,dim}} s_f \beta_{f,i} A_{\text{pixel},dim}. \quad (3.5)$$

where $\mathcal{F}_{a,dim}$ denotes the set of boundary fragments of element a for projection axis dim . Following Faure et al. [FBAF08], the factor $1/3$ is only applied to the accumulated overlap volume and not the gradient, since the three projections do not provide three complete gradient vectors to be averaged, see [FBAF08] for details. Instead, each projection contributes the derivative component along its viewing axis. The resulting components over all vertices of element a form the stacked gradient $\partial\mathcal{V}/\partial\mathbf{x}_a$ used by the collision resolution stage.

Since the rasterized surface is represented by centroid-fan triangles, one vertex of each triangle is the face centroid, introduced in Section 3.3.3. This centroid is not an independent simulation vertex, but is computed from the vertices of the original polygonal face. In our proposed method, its barycentric contribution is distributed equally to all vertices of the original face, which maps the gradient from the rasterized triangle representation back to the actual element vertices.

3.5.5 Collision Normal Approximation

In addition to the overlap gradient, our method computes an approximate collision normal for each detected pair. This normal is derived from the accumulated overlap gradient

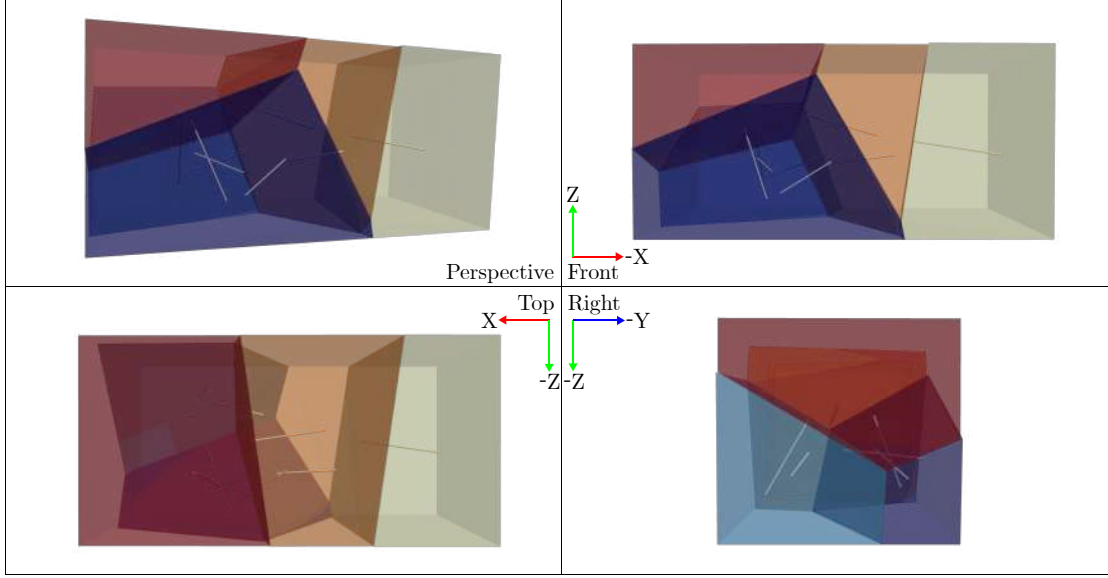


Figure 3.6: Approximate collision normals for eight detected collision pairs on five elements. The white line segments indicate the pair normals estimated from the accumulated overlap volume gradient per element. Visualized normals originate from the center of the view volume used for rasterization.

directions of the two elements. For each boundary fragment that starts or ends an overlap interval, we accumulate the signed pixel-area contribution along the current projection axis separately for element a and element b , without scattering it to the vertices involved. This gives two gradient estimates, denoted by $\mathbf{g}_a$ and $\mathbf{g}_b$.

The pair normal is then computed from their relative direction,

$$\mathbf{n}_a^{\text{coll}} = -\mathbf{n}_b^{\text{coll}} = \frac{\mathbf{g}_b - \mathbf{g}_a}{\|\mathbf{g}_b - \mathbf{g}_a\|}. \quad (3.6)$$

The intuition is that $\mathbf{g}_a$ and $\mathbf{g}_b$ describe directions in which moving an element increases the overlap volume. Since the two elements are colliding, these directions tend to be opposed. Their relative direction therefore provides an estimate of the separating direction of the pair. These computed collision normals are visualized in Figure 3.6 and are used as approximate collision directions.

At this point, the narrow phase has reduced each candidate pair to the quantities required by the collision response stage: an approximate overlap volume, per-element volume gradients and an approximate collision normal. The next section describes how these quantities are converted into position corrections and force feedback for the solver.

3.6 Pair-Local Collision Resolution

Since the volume gradient points in the direction in which moving the vertices increases the overlap volume the most, the negative gradient provides a direction for reducing interpenetration. Based on this observation, our method applies collision response directly to the vertices in a position-based manner. At the same time, the applied position correction is converted into a force-like quantity and fed back into subsequent solver iterations.

3.6.1 Position-Based Correction

A purely force-based collision response would add collision forces to the solver state and let the next solver iteration integrate their effect. In the current pipeline, this would introduce a delay of one iteration between detecting an overlap and moving the intersecting elements apart. To avoid this delay, our method directly manipulates the current vertex positions after the narrow phase has computed the overlap volume and its gradient.

For each collision pair, we split the overlap volume equally between the two elements. Let

$$\mathcal{V}_a = \mathcal{V}_b = \frac{\mathcal{V}}{2} \quad (3.7)$$

denote the overlap volume assigned to each side of the collision pair. The correction is computed independently for both elements. The following derivation considers only element a , while element b is handled analogously.

Our correction method follows the projection ideas used in position-based dynamics [MHHR07]. Element a is responsible for removing its assigned share $\mathcal{V}_a$ of the scalar constraint violation whose target value is zero. Let $\mathbf{x}_a$ contain all vertex coordinates of element a , and let

$$\mathbf{g}_a = \frac{\partial \mathcal{V}}{\partial \mathbf{x}_a} \quad (3.8)$$

denote the gradient of the overlap volume with respect to these coordinates, where each component describes the local change in overlap volume caused by a unit displacement of one vertex coordinate [FBAF08]. Let $\Delta \mathbf{x}_a$ denote a sufficiently small displacement of all vertex coordinates of element a , while element b is considered fixed. Assuming that the volume gradient remains constant during this displacement, each coordinate contributes its partial derivative multiplied by the corresponding displacement. Summing these contributions gives the approximate change in overlap volume:

$$\Delta \mathcal{V} \approx \mathbf{g}_a^T \Delta \mathbf{x}_a. \quad (3.9)$$

The dot product therefore approximates the volume change caused by moving vertices of element a . To remove the overlap volume $\mathcal{V}_a$ assigned to element a , this approximate change must equal $-\mathcal{V}_a$. The linearized constraint is thus

$$\mathcal{V}_a + \mathbf{g}_a^T \Delta \mathbf{x}_a = 0. \quad (3.10)$$

Moving opposite to the volume gradient reduces the overlap, so the correction vector is defined as

$$\Delta \mathbf{x}_a = -\alpha_a \mathbf{g}_a, \quad (3.11)$$

where α_a scales the negative gradient direction. Substituting this correction into the linearized overlap constraint gives

$$\begin{aligned} 0 &= \mathcal{V}_a + \mathbf{g}_a^T \Delta \mathbf{x}_a \\ &= \mathcal{V}_a + \mathbf{g}_a^T (-\alpha_a \mathbf{g}_a) \\ &= \mathcal{V}_a - \alpha_a \mathbf{g}_a^T \mathbf{g}_a \\ &= \mathcal{V}_a - \alpha_a \|\mathbf{g}_a\|^2. \end{aligned} \quad (3.12)$$

Rearranging the last equation gives

$$\alpha_a \|\mathbf{g}_a\|^2 = \mathcal{V}_a, \quad (3.13)$$

and thus

$$\alpha_a = \frac{\mathcal{V}_a}{\|\mathbf{g}_a\|^2}. \quad (3.14)$$

Thus, α_a is the scale factor that solves the linear approximation of the constraint. If the gradient is small, a larger displacement is required to produce the same linear change in overlap volume. If the squared gradient norm is below a certain threshold, no reliable correction direction is available and the implementation applies no correction. The correction is finally applied to the current vertex positions as

$$\mathbf{x}_a^{\text{new}} = \mathbf{x}_a + \Delta \mathbf{x}_a. \quad (3.15)$$

Element b receives an analogous correction that removes its assigned share $\mathcal{V}_b$. Since $\mathcal{V}_a + \mathcal{V}_b = \mathcal{V}$, the two corrections together remove the complete overlap in the linear approximation. However, because this approximation assumes that the volume gradients remain constant during the displacements, the actual overlap volume after applying the corrections may not be exactly zero.

3.6.2 Collision Force Integration

Although the position-based collision response directly changes the vertex positions, our method still feeds the applied collision correction back into the solver as a force quantity. Our method then adds

$$\mathbf{f}^{\text{coll}} = k^{\text{contact}} \Delta \mathbf{x} \quad (3.16)$$

into the collision force buffer, where k^{contact} is a user-defined contact stiffness parameter. These forces accumulate over the iterations of a timestep and get fed back into the next solver iteration as a force acting on the corresponding vertex making the solver aware of the corrections that were applied to reduce the overlap.

If frictional forces are enabled, we additionally compute a tangential force after the position-based projection. For each vertex of element a , we approximate the relative velocity using the average velocity of the opposing element b , obtained from all of its vertices. Since our method does not determine an explicit contact point, this element-average velocity may differ from the velocity at the actual contact location.

For a vertex i , the approximate collision normal $\mathbf{n}_a^{\text{coll}}$ is used to remove the normal component of this relative velocity, leaving only the tangential component $\mathbf{v}_{t,i}$. The friction force is computed as

$$\mathbf{f}_i^{\text{fric}} = -k^{\text{contact}} \mathbf{v}_{t,i}. \quad (3.17)$$

To compute Coulomb friction, its magnitude is clamped by the normal collision force estimated from the position correction, such that:

$$\|\mathbf{f}_i^{\text{fric}}\| \leq \mu \|\mathbf{f}_i^{\text{coll}}\|, \quad (3.18)$$

where μ is the global friction coefficient for the simulation. If the initially computed force $\mathbf{f}_i^{\text{fric}}$ exceeds this bound, it is rescaled to the maximum allowed magnitude,

$$\mathbf{f}_i^{\text{fric}} = \mu \|\mathbf{f}_i^{\text{coll}}\| \frac{\mathbf{f}_i^{\text{fric}}}{\|\mathbf{f}_i^{\text{fric}}\|}. \quad (3.19)$$

The same computation is performed for element b using the average velocity of element a and the opposite normal direction $\mathbf{n}_b^{\text{coll}} = -\mathbf{n}_a^{\text{coll}}$. The resulting friction forces are accumulated in a separate buffer and fed into the next solver iteration.

This concludes the explanation of our collision-handling method. Precomputed active element and adjacency information restricts collision detection to relevant, non-neighboring elements, whose AABBs are organized in a BVH to accelerate candidate pair generation. For each candidate pair, the image-based narrow phase rasterizes the centroid-fan surface triangulation into depth-sorted fragment lists and uses the detected overlap intervals to approximate the overlap volume, its gradients, and an approximate collision normal. The detected volume is split between the two elements and reduced through position-based corrections along their respective gradients. Finally, these corrections are converted into collision forces, while tangential relative motion produces friction forces; both are fed into subsequent solver iterations. The following chapter describes the implementation of our proposed method, with emphasis on its design to achieve parallel execution on GPUs.

Implementation

This chapter describes our implementation of the collision handling method outlined in Chapter 3. While the previous chapter focuses on the conceptual explanation of our method, this chapter discusses how the pipeline integrates into the existing simulation framework and how the data and kernels are designed for GPU execution.

4.1 Framework Integration

Our collision handling pipeline extends the existing GPU-based VEM framework with additional stages during the simulation loop. Within each solver iteration, `LaunchBlockElementLocalSolve()` first updates the vertex positions of each element independently based on its local elastic and stabilization forces, as described in Section 2.5.4. Our collision pipeline then executes before `LaunchConstraintProjection()` enforces the mesh-connectivity, Dirichlet boundary, and floor-contact constraints. All collision stages operate directly on the device-resident maximal-coordinate vertex positions, so the geometry remains on the GPU and we do not transfer geometry data to the CPU during iterations. Host-side topology processing and the corresponding data transfers occur only when the set of collision-active elements changes, such as in the event of fracture. Therefore, host-device memory transfers during simulation remain limited to small scalar values needed to determine the sizes of dynamically allocated, simulation-dependent buffers, such as the collision-pair list or the buffers related to pair overlaps. The GPU kernels update and consume the data contained in these buffers entirely on the GPU.

Our implementation combines BVH-based candidate pair generation using NVIDIA’s BVH library *cuBQL* [Wal26], an image-based narrow phase implemented as a custom CUDA rasterizer, following Faure et al. [FBAF08] method, and position-based collision resolution including force feedback computation. Algorithm 4.1 gives an overview of the stages that one pipeline iteration executes. Except for line 2 and kernel launch related buffer setup, all computations execute entirely on the GPU using CUDA.

Algorithm 4.1: One iteration of the collision handling pipeline.

Input: Solver updated vertex positions $\mathbf{x}$
Output: Corrected vertex positions $\mathbf{x}^{\text{new}}$ and collision-force buffers $\mathbf{f}^{\text{coll}}$, $\mathbf{f}^{\text{fric}}$

- 1 **if** *collision-active element set changed* **then**
- 2 Build active element mapping, adjacency lists, triangulation data and upload to device;
- 3 Construct AABBs;
- 4 **if** *first iteration of time step* **then**
- 5 Build BVH;
- 6 **else**
- 7 Refit BVH;
- 8 $\mathcal{P} \leftarrow$ traverse BVH in parallel over $\mathcal{K}_{\text{act}}$;
- 9 **foreach** *candidate pair p in $\mathcal{P}$ in parallel* **do**
- 10 $(\mathcal{V}, \frac{\partial \mathcal{V}}{\partial \mathbf{x}}, \mathbf{n}^{\text{coll}}) \leftarrow$ execute image-based narrow phase;
- 11 Compute position correction $\Delta \mathbf{x} = -\alpha \frac{\partial \mathcal{V}}{\partial \mathbf{x}}$;
- 12 $(\mathbf{x}^{\text{new}}, \mathbf{f}^{\text{coll}}) \leftarrow$ project positions and compute collision forces using $\Delta \mathbf{x}$;
- 13 $\mathbf{f}^{\text{fric}} \leftarrow$ compute friction force using $\mathbf{n}^{\text{coll}}$;

4.2 Representing Irregular Topology on GPU

Our collision pipeline operates on polyhedral elements with irregular topology. Elements may have different numbers of vertices or faces, and the number of degrees of freedom (DoF) involved in a collision pair depends on the two participating elements. Due to these irregularities, we store variable-length data in flattened arrays and use offset and length arrays to identify the contiguous segments of memory belonging to a specific element, face, or collision pair. Depending on the kernel launch configuration, block or thread indices retrieve the corresponding offsets and lengths.

4.2.1 Global and Active Element Index Spaces

Additionally, we distinguish between two element-index spaces. The global element index refers to the element indices of the original simulation mesh, while the active element index, introduced in Section 3.3.1, refers to the compact set of elements that participate in collision handling. A redirection array maps each active element index to its corresponding global element index. Figure 4.1 illustrates this indirection pattern, from active element indices to global element data and from offset and length buffers to the contiguous memory region of the elements. This redirection avoids duplicating vertex data for collision-active elements.

We assign each candidate pair a pair index p . In the flattened buffer `collisionLists[]`, entries $2p$ and $2p + 1$ store the active indices a and b of the two potentially colliding elements. Quantities with one value per pair are accessed directly through p , for ex-

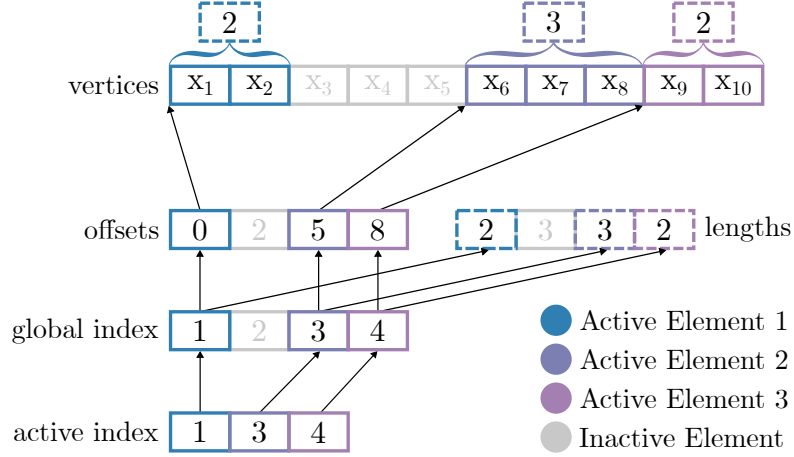


Figure 4.1: Index redirection and offset-based access for irregular element data.

ample `pairVolumes[p]` and `pairNormals[p]`. The number of gradient entries, however, depends on the number of DoFs of elements a and b . Therefore we store their gradients continuously in the flattened buffer `pairVolumeGradient[]`, while `pairGradientOffsets[p]` identifies where the gradient data of pair p begins. The following pipeline stages use this pair-based indexing and offset layout.

4.3 Candidate Pair Buffer Construction

For Candidate pair generation we take the current simulation state and produce the input for the image-based narrow phase. The current vertex positions are stored in the flattened buffer `virtualDOFs[]`, using the maximal-coordinate representation described in Section 2.5.4. Offsets and lengths identify the contiguous memory range of degrees of freedom belonging to each element. The broad phase then writes its output to the pair buffer `collisionLists[]` described above.

Starting from these buffers, candidate pair generation is divided into AABB construction, BVH construction or refitting, and BVH traversal. In the first step we construct one bounding box per active element, forming the primitive set used by the BVH.

4.3.1 Parallel AABB Construction

The AABB construction kernel `BlockReduceConstructAABB()` assigns one block of 32 threads to each active element. The block index maps the active element index to the corresponding global element index and retrieves the element's segment in the maximal-coordinate DoF array `virtualDOFs[]`. We distribute the element vertices among the threads in a block-strided manner: thread t processes vertices t , $t + 32$, $t + 64$, and so forth. Each thread records the minimum and maximum coordinates among its assigned vertices. A block reduction then combines these partial bounds. This reduction

first combines values within each warp and then, if the block contains multiple warps, combines one final result per warp through shared memory into a block result. After the reduction, thread-0 holds the minimum and maximum coordinates of the complete element and writes the final AABB. The same reduction pattern is used in later kernels and Algorithm 4.2 summarizes this general reduction pattern.

Algorithm 4.2: block reduction of thread values.

Input: One partial result r_t per thread**Output:** One combined result r_b for the complete block

```
1 foreach thread in parallel do
2   combine the partial results  $r_t$  within each warp using shuffle instructions;
3   if the thread is the first thread of its warp then
4     store the warp result  $r_w$  in shared memory;
5 synchronize threads;
6 the first warp combines the stored warp results  $r_w$  using shuffle instructions;
7 thread 0: write the combined block result as  $r_b$ ;
```

4.3.2 BVH Construction and Refitting

After the AABB construction kernel has computed the active element bounds, our implementation either builds a new BVH or refits the existing one, depending on the current iteration. For BVH construction and traversal, we use NVIDIA’s BVH Build-and-Query Library *cuBQL* [Wal26]. During the first solver iteration of a time step, `LaunchBroadPhaseBVHBuild()` calls `cuBQL::gpuBuilder()` on the active element AABB array. The default *cuBQL* build configuration uses an adaptive spatial-median strategy, recursively splitting the AABBs according to their centers along the axis with the widest spatial extent. For subsequent solver iterations within the same time step, `LaunchBroadPhaseBVHLeafRefit()` launches the same parallel AABB construction kernel but then calls `cuBQL::gpuBuilder_impl::refit()` instead. The refit updates the bounding boxes of the leaf and internal nodes without changing the BVH topology, avoiding a full rebuild of the hierarchy. If fracture or another topology update changes the set of active elements, we rebuild the active element mapping, adjacency data, and triangulation data on the host side and upload it to the device before the next BVH construction or refitting step. Once the hierarchy contains the current active element bounds, we traverse it to generate the candidate pairs.

4.3.3 BVH Traversal

In our main traversal path we construct the candidate pair array in two passes, as shown in Algorithm 4.3. Each active element queries the BVH with its own AABB using `cuBQL::fixedBoxQuery::forEachPrim()`. We only accept overlaps with larger active index, avoiding duplicate entries, and rejecting adjacent elements, by scanning

through the precomputed adjacency lists. The first kernel, `BVHTraversalCount()`, performs this traversal only to count the accepted collision partners of each active element. An exclusive prefix sum over these counts converts them into output offsets and determines the required size of the flattened pair buffer `collisionLists[]`. The second kernel, `BVHTraversalFill()`, repeats the same traversal and uses the offset assigned to each active element to write its accepted pairs into the predetermined section of the buffer. Each pair occupies two consecutive entries containing the two active element indices. This two-pass approach enables exact buffer allocation and avoids atomically appending pairs making the broad phase result deterministic.

Both kernels are launched with one block per active element and one thread per block. This assigns each BVH query to an independently scheduled block. Since different queries may traverse different numbers of BVH nodes, grouping many queries into the same block would introduce thread divergence and tail effects, where threads that finish short traversals cannot receive new work until the longest traversal in the block has completed. The single thread block configuration avoids this block-level dependencies at the cost of launching many small blocks.

Algorithm 4.3: Two-pass BVH traversal.

Input: BVH, active element AABBs, active-index adjacency lists

Output: Candidate pair array $\mathcal{P}$

```

1 foreach active element  $a$  in parallel do
2    $c[a] \leftarrow 0$ ;
3   foreach BVH overlap  $b$  reported for  $AABB(a)$  do
4     if  $b > a$  and  $b$  is not adjacent to  $a$  then
5        $c[a] \leftarrow c[a] + 1$ ;
6  $k \leftarrow \text{exclusivePrefixSum}(c)$ ;
7 allocate  $\mathcal{P}$  with  $k[\text{last}] * 2$  entries;
8 foreach active element  $a$  in parallel do
9    $p \leftarrow k[a]$ ;
10  foreach BVH overlap  $b$  reported for  $AABB(a)$  do
11    if  $b > a$  and  $b$  is not adjacent to  $a$  then
12       $\mathcal{P}[2p] \leftarrow a$ ;
13       $\mathcal{P}[2p + 1] \leftarrow b$ ;
14       $p \leftarrow p + 1$ ;

```

In the broad phase we only identify pairs with overlapping AABBs. Since such overlaps do not imply actual intersection of the underlying polyhedral elements, we further evaluate the resulting candidate pairs in the narrow phase.

4.4 Custom CUDA Compute Rasterizer

We implement the narrow phase, following our method introduced in Section 3.5, as a custom CUDA compute rasterizer that processes each entry of the candidate-pair buffer independently.

4.4.1 Pre-launch Buffer Preparation

Before launching the rasterizer kernel, we allocate one output entry per candidate pair for overlap volumes, approximate normals, and gradient offsets. Only the pair gradient buffer `pairVolumeGradient[]` has variable size, since it depends on the two elements involved. For each pair, our implementation sums the DoF counts of both elements, an exclusive scan then stores the offsets into the corresponding buffer `pairGradientOffsets[]`, and the total sum over all pairs determines the capacity.

If a thread’s fixed-size depth-sorted fragment list (DSFL), which holds the rasterized fragments, overflows during rasterization, our implementation records the maximum number of fragments generated by any pixel of a pair in a separate buffer `perPairMaxFrag[]`. We then rerun the pair with a preallocated global DSFL buffer sized to the recorded maximum. After these pair-dependent buffers have been prepared, our rasterizer kernel can start to process each candidate pair independently in parallel.

4.4.2 Rasterizer Kernel Setup

We launch the custom rasterizer kernel `NarrowPhaseKernel()` with one block per candidate pair and a thread count of 128. The block index retrieves the corresponding entry in the pair buffer and consequently the two active elements, their DoF ranges, and the pair-dependent buffer write regions. The raster image has a fixed, compile-time resolution, and each thread handles a fixed number of pixels, computed from the image resolution and the thread count as shown in the Listing 4.1. This value determines the size of the DSFLs and fragment counters that the thread allocates. This pixel ownership is one of the main design decisions that enables our rasterization stage to run efficiently in parallel, with each thread handling its assigned pixels and storing the corresponding fragment lists locally. When the required storage is small enough, the compiler can keep these lists in low-latency registers. However, increasing the resolution or the DSFL depth increases the amount of thread-local storage required by the kernel. Depending on the compiler’s register allocation and the target GPU architecture, parts of this memory may spill to local memory, which (contrary to what the name suggests) resides in global memory and therefore leads to higher latency memory accesses.

Listing 4.1: Example memory allocation for one thread, slightly altered from original source code for readability.

```
constexpr int threadsPerBlock = 128;
constexpr int imgResolution = 32;
constexpr int DSFLdepth = 16; // stores 16 fragments per pixel
```

```
constexpr int totalPixels = imgResolution * imgResolution; // 1024
constexpr int pixelsPerThread = totalPixels / threadsPerBlock; // 8
NPFFragment DSFL[pixelsPerThread][DSFLdepth]; // DSFL[8][16]
int fragmentCount[pixelsPerThread]; // fragmentCount[8]
```

Our kernel allocates data used by all threads of a block in shared memory to reduce latency. This includes the view volume, the accumulated pair volume, memory used for block reduction, aggregate gradient sums for the approximate normal, the overflow flag, and an array `shTriBatch[]` for storing preprojected triangles. The first thread in each block (Thread-0) initializes the shared variables and computes the view volume as the intersection of both element AABBs.

4.4.3 Axis Loop

After setting up and initializing all needed variables, the most complicated section of our implementation starts. Our rasterizer projects each candidate pair's triangles from all three coordinate directions, x , y and z sequentially. Algorithm 4.4 summarizes the program flow of this axis loop, and the following subsections describe its main stages in more detail.

Algorithm 4.4: Rasterizer axis loop for one candidate-pair block.

```
1 foreach projection axis  $d \in \{x, y, z\}$  do
2   compute image-plane axes and pixel size for axis  $d$ ;
3   reset per-pixel fragment counters;
4   foreach triangle batch of candidate pair do
5     foreach triangle from current thread's block-strided subset do
6       construct preprojected triangle and store in shared memory;
7     __syncthreads();
8     foreach preprojected triangle in batch do
9       foreach pixel owned by current thread do
10        test triangle against pixel and insert generated fragments into
11        thread's DSFL;
12      __syncthreads();
13    foreach pixel owned by current thread do
14      traverse DSFL, accumulate thread's overlap contribution and compute
15      overlap gradients;
16    block reduce thread's overlap contributions into pair volume;
17    __syncthreads();
```

4.4.4 Triangle Batching

The first nested loop of the axis loop, shown in Lines 4–6 of Algorithm 4.4, processes triangles in fixed-size batches of 256 entries. This batching is the mechanism that makes the irregular topology efficiently manageable on the GPU. Since elements may contain different numbers of triangles, storing all preprojected triangles of a pair would require a pair-dependent buffer size. CUDA does support dynamic memory allocation inside kernels, but it is generally considered slow for performance-critical tasks. An alternative would be to preallocate a sufficiently large buffer in global memory, but the rasterization loop repeatedly accesses the stored triangle data while testing pixels, which would introduce many high-latency memory accesses in the innermost loop of the kernel in Line 10 of Algorithm 4.4. By streaming triangles through the fixed-size shared-memory array `shTriBatch[256]`, we reduce memory latency and still support pairs of arbitrary topological size by processing as many batches as required.

During the cooperative load phase of each batch, shown in Line 6 of Algorithm 4.4, the block’s threads construct the centroid-fan triangles using the precomputed triangulation indices, as discussed in Section 3.3.3, of both elements. We then project these triangles onto the current image plane and store the data required during rasterization inside `shTriBatch[]` using a shared-memory representation `SharedMemTri` shown in the Listing 4.2.

Listing 4.2: Shared-memory representation of one preprojected triangle.

```
struct SharedMemTri
{
    int pixMinX, pixMinY, // clipped bounding rectangle min
    int pixMaxX, pixMaxY; // clipped bounding rectangle max
    float w0Start, w1Start; // barycentrics at first pixel center
    float w0dCol, w0dRow; // barycentric increments
    float w1dCol, w1dRow; // barycentric increments
    float v0d, v1d, v2d; // vertex depths along current axis
    unsigned int packed; // object and triangle idx
};
```

For every triangle, we first compute its clipped pixel bounding rectangle. We then compute the barycentric coordinates at the center of the first pixel in this rectangle, located at its minimum-coordinate corner (`pixMinX, pixMinY`). A half-pixel offset from the corner places the sample at the pixel center. We store these coordinates as `w0Start` and `w1Start`. Since barycentric coordinates vary linearly across a projected triangle, moving by one pixel along a row or column changes them by a constant amount. We precompute these changes as the row and column increments used for incremental barycentric evaluation [Pin88]. We also store the vertex depths along the current depth axis. To reduce memory, we pack the object index and triangle index into one integer, which later identifies the original triangle during gradient computation. Performing these computations once per triangle and axis avoids repeatedly clipping the triangle and computing its barycentric coordinates from scratch for every pixel.

4.4.5 Per-Pixel Fragment Generation

After we construct the full shared triangle batch, the nested loops shown in Lines 8–10 of Algorithm 4.4 generate the pixel fragments. Every thread iterates over all triangles of the batch and evaluates them against its assigned subset of pixels. We immediately skip empty entries, which represent degenerate or fully clipped triangles. For all remaining triangles, we use the stored pixel bounding rectangle as a rejection test before computing the barycentric coordinates from the precomputed start values `w0Start` and `w1Start` and the row and column increments `w0dCol`, `w0dRow`, `w1dCol`, and `w1dRow`. If the resulting barycentric coordinates place the pixel center inside the triangle, we interpolate the fragment depth from these coordinates and the stored vertex depths `v0d`, `v1d`, and `v2d`. We store the generated fragment’s depth, two barycentric coordinates, and packed triangle identifier in the `NPFragment` struct defined in Listing 4.3. We then insert the `NPFragment` into the pixel’s DSFL in depth order using insertion sort, avoiding a separate sorting pass afterwards. We increment the pixel’s fragment counter for every generated fragment; this counter later also determines the range traversed in the DSFL. If the fixed-size DSFL capacity is exceeded, we record the required capacity in the corresponding pair-indexed entry of `perPairMaxFrag`s[] for an overflow rerun.

Listing 4.3: Per-pixel fragment representation.

```
struct alignas(16) NPFragment
{
    float depth;           // depth along the current projection axis
    float bary0;           // barycentric weight for v0
    float bary1;           // barycentric weight for v1
    unsigned int packed;   // object and global triangle id
};
```

4.4.6 DSFL Traversal and Overlap Accumulation

After our rasterizer has processed all triangle batches of the current projection direction for all pixels, the final stage of the axis loop, shown in Lines 12–15 of Algorithm 4.4, executes the traversal introduced in Section 3.5.4 on the DSFLs. The work distribution follows the same pixel ownership logic as the rasterization stage, where each thread traverses the DSFLs of its assigned pixels and accumulates their contribution in a scalar `threadVol`.

The traversal skips the pixel’s DSFL if the fragment counter is zero and otherwise processes only the recorded fragments up to the counter’s value, which eliminates the need to clear the occupied memory after every pass. During traversal, following Section 3.5.3, we store the start depth of an overlap interval inside the local variable `overlapStart`. Once the interval closes, we compute `intervalDepth` as the difference between the current fragment’s depth and `overlapStart`. We then multiply this resulting interval depth value by the current pixel’s area `pixelArea` and add it to `threadVol`.

4.4.7 Pair-Local Gradient Scatter and Normal Aggregation

In the same traversal described above, we pass the two fragments that define an overlap interval to `ScatterGradContribution()` with the sign determined by whether the fragment starts or ends the interval, as described in Section 3.5.4. Inside `ScatterGradContribution()`, we use the unpacked information `globalTriIdx` and `objectId` from the current `NPFragment` to map the triangle back to the element's original polygonal face vertex component and retrieve the index into the pair gradient buffer. We distribute the centroid's barycentric weighted gradient contribution evenly over all DoFs of the original polygonal face, while the two remaining barycentric weighted gradient contributions are added to the corresponding gradient buffer location belonging to the original element's DoF. For this our implementation uses `atomicAdd()` since several pixels, and therefore threads, of the same pair may contribute to the same DoF simultaneously, leading to race conditions. At the same time, we add the unscattered contribution for the current projection axis to the shared arrays `shPairGradSumA` and `shPairGradSumB`, which we later use to approximate the collision normal.

4.4.8 Overlap Volume Block Reduction

Once we have traversed all DSFLs of the current axis, we reduce all threads' `threadVol` values into the shared block variable `shBlockVol` using block reduction, as shown in Lines 14–15 of Algorithm 4.4.

After our rasterizer has processed all three projection directions, `thread-0` averages the accumulated block volume and stores it as the pair's overlap volume. We also compute the approximate pair normal from the difference between `shPairGradSumB` and `shPairGradSumA` and store it for later force computations. The pair volume, pair gradients, and approximate normal form the narrow phase output for the candidate pair and provide the information required for the following collision response stage.

4.5 Collision Response

After our narrow phase has processed all candidate pairs, the final stage of our collision handling pipeline consumes the buffers produced by the narrow phase. we apply position-level corrections to reduce interpenetration and fill the corresponding collision and friction force buffers that the next solver iteration consumes.

4.5.1 Position-based Correction

We launch the collision resolution kernel `PBCRProjectPairPositions()` with 128 threads per block, and keep the same one-block-per-pair execution model as with our narrow phase kernel. Each block maps the active pair indices back to element indices, as shown in Figure 4.1, retrieves the DoF offsets and DoF counts of both elements, and uses `pairGradientOffsets[]` to access the current pair's range in the gradient buffer `pairVolumeGradient[]`.

We compute the squared gradient norm for both elements separately using the block-reduction pattern introduced in Section 4.3.1. Thread zero stores the two values `gradNormSqA` and `gradNormSqB` into a global buffer for later friction force computation and computes the step factor per element, following Section 3.6.1. We store the resulting step factors in shared memory, so all threads can access them in the subsequent projection loop. we then traverse the DoF ranges again, multiply each corresponding gradient entry by the element’s step factor, and apply the positional correction by atomically adding the resulting value to the DoF in the position buffer `virtualDOFs`.

4.5.2 Force Accumulation

The same correction loop also stores the force feedback for the collision response. We scale the correction by the contact stiffness, as explained in Section 3.6.2, and accumulate it per DoF in the global buffer `collisionForces[]`. For Both the position update and the force update we use `atomicAdd()` to prevent race conditions since multiple collision pairs may reference the same element and therefore the same DoF entries.

If friction is enabled, we launch `PBCRCollisionFriction()` immediately after the position correction kernel, using the same launch configuration. As described in Section 3.6.2, the kernel computes an average velocity for each element from the current DoF positions stored in `virtualDOFs[]` and the previous iteration positions stored in `virtualDOFs_t0[]`. We combine the per-thread velocity sums using block-reduction and store it in shared memory. Our kernel then uses the approximate pair normal to compute the tangential friction contribution. We accumulate the resulting values in a separate buffer `collisionFrictionForces[]`.

We clear both the collision force and collision friction force buffers before the first solver iteration of every timestep. During the subsequent element-local solve, the solver adds `collisionForces[]` and `collisionFrictionForces[]` to the element force vector together with other external forces.

4.6 Contribution Summary

Overall, we implement our complete collision pipeline on the GPU and adapt it to the irregular topology of polyhedral meshes. We use flattened buffers, offsets, and the active-to-global index mapping to process elements and collision pairs of different sizes without duplicating any data. In the broad phase, the two-pass traversal produces a compact and deterministic candidate-pair buffer. The central implementation contribution is our custom parallel CUDA compute rasterizer for the image-based narrow phase. We map each candidate pair to one block and distribute its pixels among the block’s threads, enabling parallelism across both pairs and pixels. Fixed-size shared-memory batches let us stream arbitrary numbers of triangles through the rasterizer without allocating pair-dependent triangle storage in high-latency global memory. Rather than allocating complete raster textures in global memory for every candidate pair, each thread maintains

DSFLs for its assigned pixels and traverses them to compute the overlap volume, gradients, and approximate normal. Since the DSFLs belong to individual threads, only resident blocks require these temporary resources, and the GPU reuses them as blocks finish and get exchanged for new ones. The total number of candidate pairs is therefore not limited by the number of raster textures in VRAM, enabling the processing of large amounts of collisions. Together, these design decisions make the image-based method efficiently applicable to irregular polyhedral meshes while limiting its runtime memory requirements. We pass the resulting pair quantities to the position-correction and force kernels and use atomic updates since several collision pairs may share the same DoFs. During solver iterations, we keep the geometry and main computations on the GPU and limit host-device transfers. Next, the following chapter evaluates detection accuracy, resulting runtime, memory requirements, collision behavior, and normal approximations.

Evaluation

This chapter focuses on the evaluation of the proposed GPU collision handling pipeline. The benchmark set consists of five scenes with different contact patterns and densities and is used to evaluate runtime, memory consumption, overlap volume, and the relation between broad phase candidate pairs and confirmed narrow phase collisions. Additional tests use dedicated scenes instead of the full benchmark set to evaluate kernel execution resources, the impact of the narrow phase texture resolution on runtime and throughput, and narrow phase detection accuracy against a tetrahedral SAT reference. Friction is validated with a two-object inclined-plane-equivalent setup, and collision normal accuracy is measured in a cube-on-cube contact scene. Unless stated otherwise, all measurements use a fixed narrow phase texture resolution of 8×8 pixels, since it is a good trade-off between detection accuracy and speed.

All evaluations and measurements are performed on an Intel Core i9-13900HX with 32 threads, 32 GB of DDR5 memory at 5600 MT/s, and an NVIDIA GeForce RTX 4070 Laptop GPU with 8 GB GDDR6 VRAM on Windows 11 with CUDA version 13.1.

5.1 Benchmark Scenes

We select a set of benchmark scenes to cover three characteristic contact scenarios: activation through full fracture, gradually increasing contact between elements, and dense contact caused by confinement inside a bucket. At full fracture, we simultaneously remove the mesh-connectivity constraints between all elements, making all elements of the scene participate in the collision handling pipeline. Table 5.1 summarizes the scene setup and material parameters. Figures 5.1 and 5.2 show representative screenshots of these scenes where the first image depicts the starting configuration at timestep zero, images in the center show the simulation during its runtime and the rightmost image shows the final configuration at the end of the simulation. Specific scenes dedicated to evaluating certain aspects of the pipeline are introduced in their respective sections.

Table 5.1: Scene setup and material parameters used for the evaluation scenes. *Release* specifies when we remove the initial Dirichlet boundary condition, as illustrated for the single cube in Figure 5.1 sticking to the wall at the beginning of the simulation. *Fracture* specifies when we remove all mesh-connectivity constraints and activate the complete scene for collision handling; *none* indicates that no full fracture event occurs over the entire runtime of the simulation. *Bucket* indicates whether barriers confine the objects.

Scene	Setup	Objects	Elem./object	Total elem.	Release (s)	Fracture (s)	Bucket	λ	μ	α	β
Single cube	one cube	1	1 000	1 000	1.0	2.0	no	250.0	200.0	0.5	0.8
Stacked cubes	vertical stack	20	1 000	20 000	0.0	2.0	no	1 000.0	800.0	0.5	0.8
Pyramid	19-layer pyramid	19	100	1 900	0.0	none	no	1 000.0	800.0	0.5	0.8
Pyramid bucket	pyramid with barriers	19	100	1 900	0.0	2.0	yes	1 000.0	800.0	0.5	0.8
Grid bucket	$10 \times 10 \times 20$ cube grid	2 000	2	4 000	0.0	0.0	yes	1 000.0	800.0	0.5	0.8

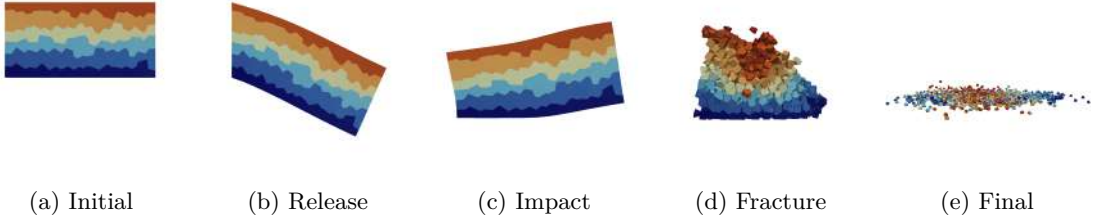


Figure 5.1: Snapshots of the single-cube test scene.

The material parameters are fixed across all scenes except the single cube, so that differences mainly reflect scene layout and collision work. The Lamé parameters λ and μ define the elastic behavior, α scales the stabilization term, and β controls the step size of the element-local solver update, which were introduced in Section 2.5. Since these parameters influence deformation, they also affect the resulting contacts.

5.1.1 Qualitative Limitations

The benchmark simulations also reveal a qualitative limitation of the current collision resolution. Although the method generally resolves collisions, the pyramid-bucket and grid-bucket scenes do not reach an equilibrium. In these confined configurations with dense contact, elements near the bottom remain in motion even though they would intuitively be expected to come to rest. The confinement makes this behavior particularly visible, but does not necessarily restrict it to such configurations. We therefore identify a more robust, complex collision response that reaches equilibrium as an important direction for future work, discussed in Section 6.1.

5.2 Runtime

The runtime measurements separate collision handling into broad phase, narrow phase, and collision resolution. Values are reported per timestep and Table 5.2 gives average and peak timings.

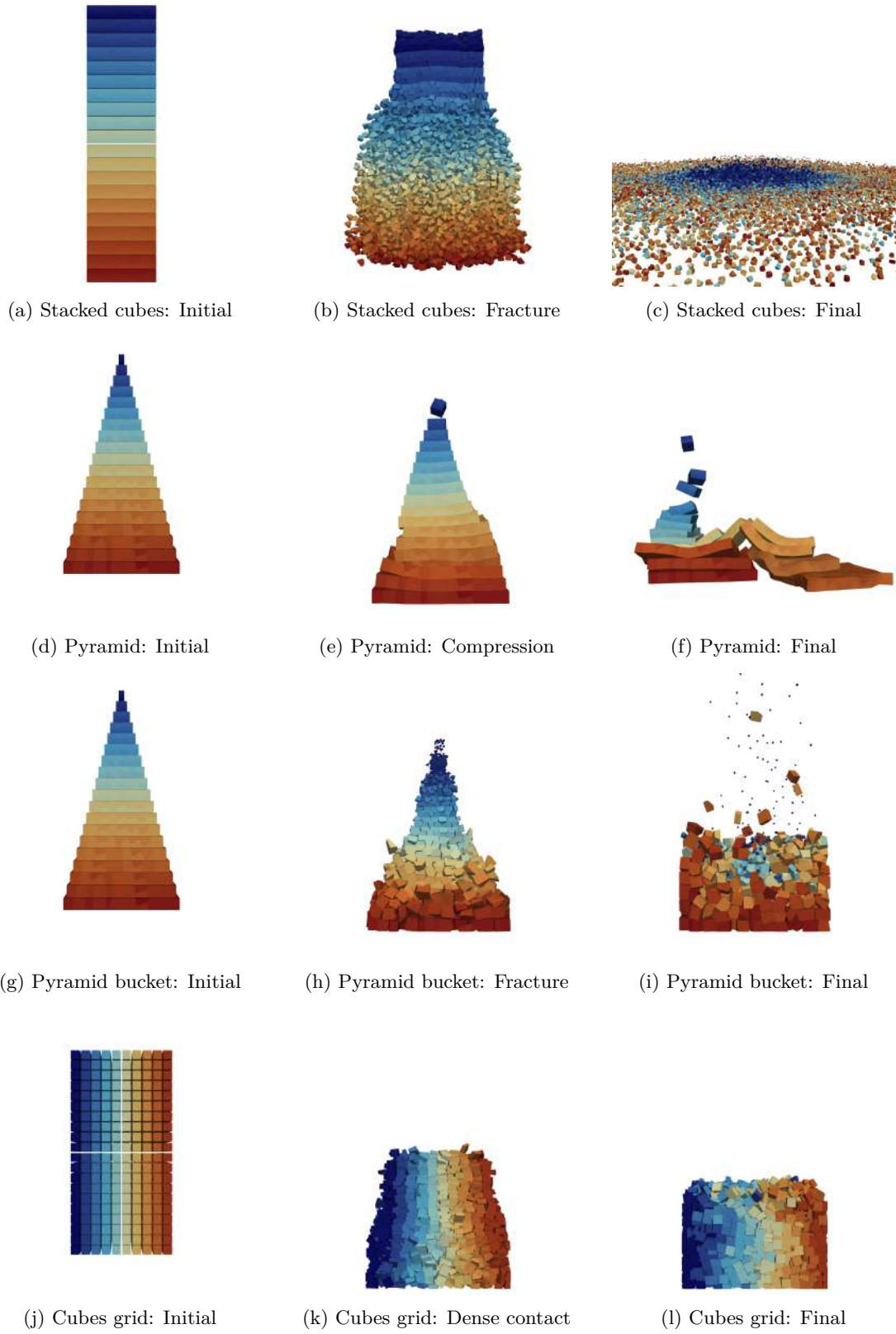


Figure 5.2: Snapshots of the stacked-cubes, pyramid, pyramid-bucket, and grid-bucket test scenes.

5. EVALUATION

Table 5.2: Overview of the evaluation scenes and their collision pipeline runtime characteristics. Runtime values are measured per timestep with a narrow phase texture resolution of 8×8 . The stage columns report the average runtime in milliseconds and its share of the average total runtime.

Scene	Elements	Vertices	Time per step avg. (ms/%)			Total avg. (ms)
			BP	NP	CR	
Single cube	1 000	21 760	6.4/22.9%	19.8/71.0%	1.7/6.1%	27.9
Stacked cubes	20 000	412 520	21.6/6.4%	307.8/91.0%	8.9/2.6%	338.3
Pyramid	1 900	29 590	6.8/15.6%	34.6/79.6%	2.1/4.8%	43.5
Pyramid bucket	1 900	29 590	7.0/12.0%	48.9/83.6%	2.6/4.4%	58.5
Grid bucket	4 000	32 000	8.3/8.2%	86.8/86.0%	5.9/5.8%	101.0

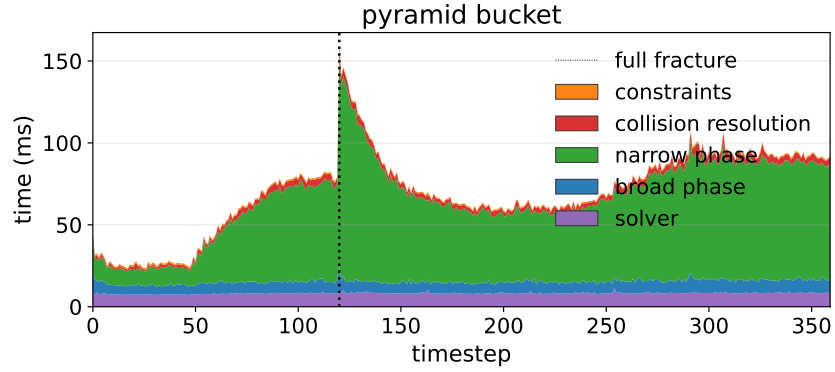
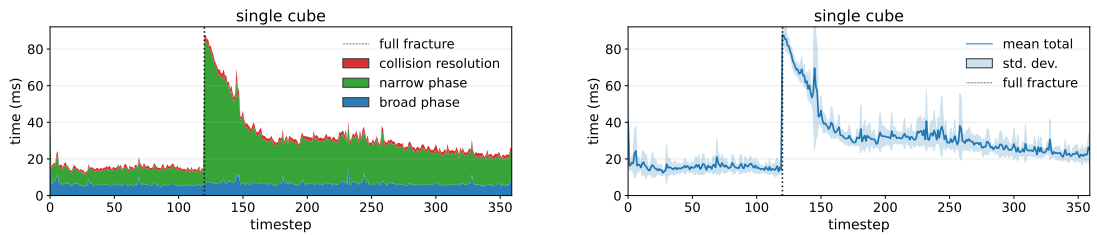


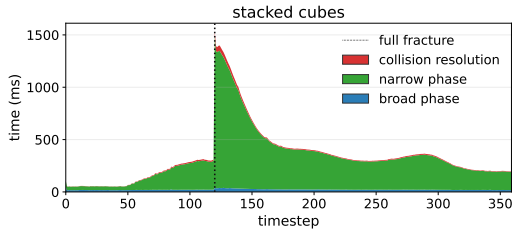
Figure 5.3: Full GPU runtime breakdown for the pyramid-bucket scene, including solver, collision stages, and constraint projection.



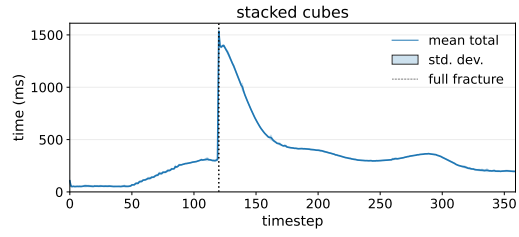
(a) Stage breakdown

(b) Total runtime

Figure 5.4: Runtime measurements for the single cube scene.

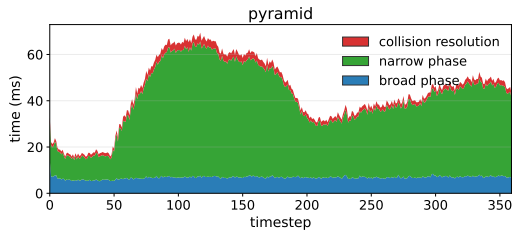


(a) Stage breakdown

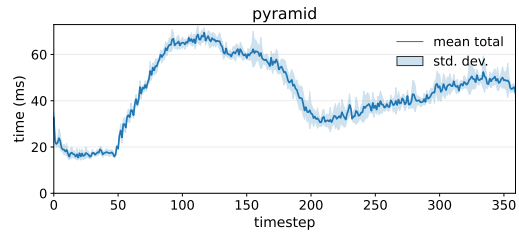


(b) Total runtime

Figure 5.5: Runtime measurements for the stacked cubes test scene.

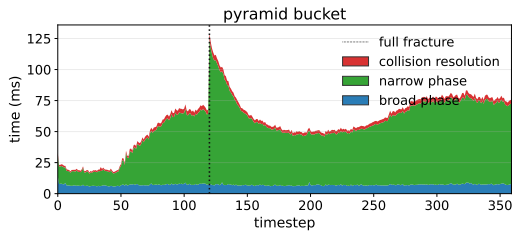


(a) Stage breakdown

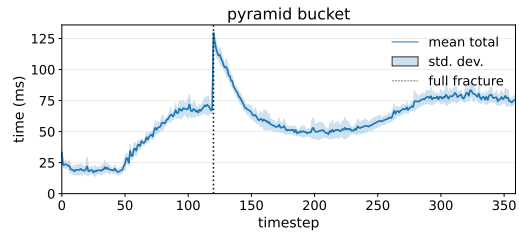


(b) Total runtime

Figure 5.6: Runtime measurements for the 19 pyramids test scene.

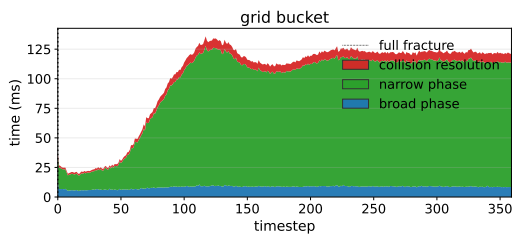


(a) Stage breakdown

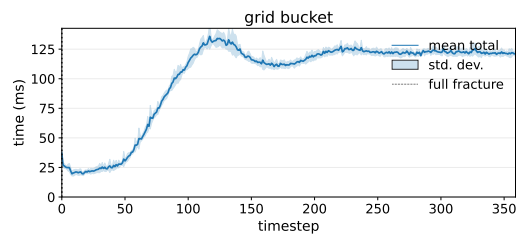


(b) Total runtime

Figure 5.7: Runtime measurements for the bucketed 19 pyramids test scene.



(a) Stage breakdown



(b) Total runtime

Figure 5.8: Runtime measurements for the grid bucket test scene.

5.2.1 Results

Figure 5.3 first shows a full breakdown for the pyramid-bucket scene, including solver and constraint projection. The solver does contribute noticeably to the runtime of a timestep, but within the full pipeline the narrow phase is still the dominant stage. Figures 5.4–5.8 therefore focus on the thesis-relevant aspects: broad phase, narrow phase, and collision resolution.

The narrow phase dominates in all collision pipeline plots. The strongest spike occurs both in the stacked-cubes scene, as well as the single cube scene, after full fracture, where newly active elements strongly increase the number of candidate pairs. The pyramid bucket scene shows the same effect at smaller scale, since the objects were already compressed at full fracture. In contrast, the pyramid scene has no full fracture event during the measured interval and reflects the compression and expansion of the objects. The grid bucket starts full fractured and its runtime grows more gradually as the cubes compact inside the bucket.

Table 5.2 shows that the narrow phase accounts for 71.0–91.0% of the average collision-pipeline runtime across all scenes. The broad phase contributes 6.4–22.9%, while collision resolution remains the smallest stage at 2.6–6.1%. Across the benchmark scenes, the narrow phase is approximately $3\times$ to $14\times$ more expensive than the broad phase, with the largest ratio occurring in the stacked cubes scene. Although the absolute runtimes differs significantly, the ordering of the stages remains unchanged. The single-cube scene has the lowest narrow phase share and the highest broad phase share because its narrow phase invocations are comparatively small due to fast scattering of the elements after full fracture. In contrast, the stacked-cubes scene spends 91.0% of its collision runtime in the narrow phase.

5.2.2 Discussion

The broad phase is therefore not the limiting stage in the pipeline. Its cost increases with the number of active elements but remains small compared to the narrow phase. The main scalability aspect is the number of candidate pairs that we must rasterize and evaluate. The results also show the effect of active-element and neighborhood filtering: as long as only a subset of elements is active and we reject neighboring pairs, we avoid many unnecessary narrow phase launches. At full fracture, all elements become active and we disable neighborhood filtering. This explains the large narrow phase spikes in the full fracture scenes. Image-based intersection detection is therefore the primary target for future optimization.

5.3 Persistent GPU Memory

This section measures persistent GPU buffers; that is, device buffers explicitly allocated by the simulation code and kept across kernel launches. Buffers are assigned to a stage if they are filled or precomputed for that stage and exclusively consumed by it. For the

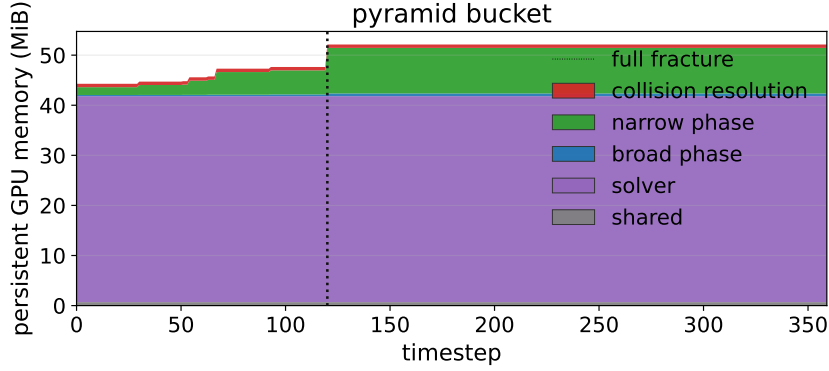


Figure 5.9: Persistent GPU memory breakdown for the pyramid-bucket scene, including solver and collision pipeline buffers.

buffers assigned to the solver, this includes, among others, the matrix operators, buffers holding positions from previous timesteps, and data used for constraint projection. Broad phase buffers contain the AABBs of the elements, the BVH, adjacency lists and the buffers related to the collision candidate pairs. Narrow Phase buffers contain triangulation data and pair-dependent output buffers. Buffers counted towards the collision resolution contain data related to the collision and friction forces. Finally buffers required by multiple stages, such as element-DOF indexing and maximal coordinates, are counted as shared.

5.3.1 Results

Figure 5.9 gives a breakdown including solver buffers for proportion. Figure 5.10 then excludes the solver and focuses on the collision pipeline. The narrow phase buffers dominate in all scenes, while broad phase memory stays small and collision resolution memory is nearly constant relative to the other buffers.

The collision-count-dependent buffers are resized to the currently needed number of entries in every iteration. However, the measured memory does not decrease once a peak allocation has been reached. The curves therefore follow the peak allocated capacity rather than the current collision count. This is most visible after full fracture in the single-cube, stacked-cubes, and pyramid-bucket scenes. The pyramid and grid bucket increase more smoothly as the contact density grows.

5.3.2 Discussion

The persistent-memory results mirror the runtime measurements: the narrow phase dominates because it owns the largest candidate-pair-dependent buffers. The peak-capacity behavior is also a practical finding, where a short interval with many candidate pairs can determine the reserved buffer capacity for the remainder of the simulation, even

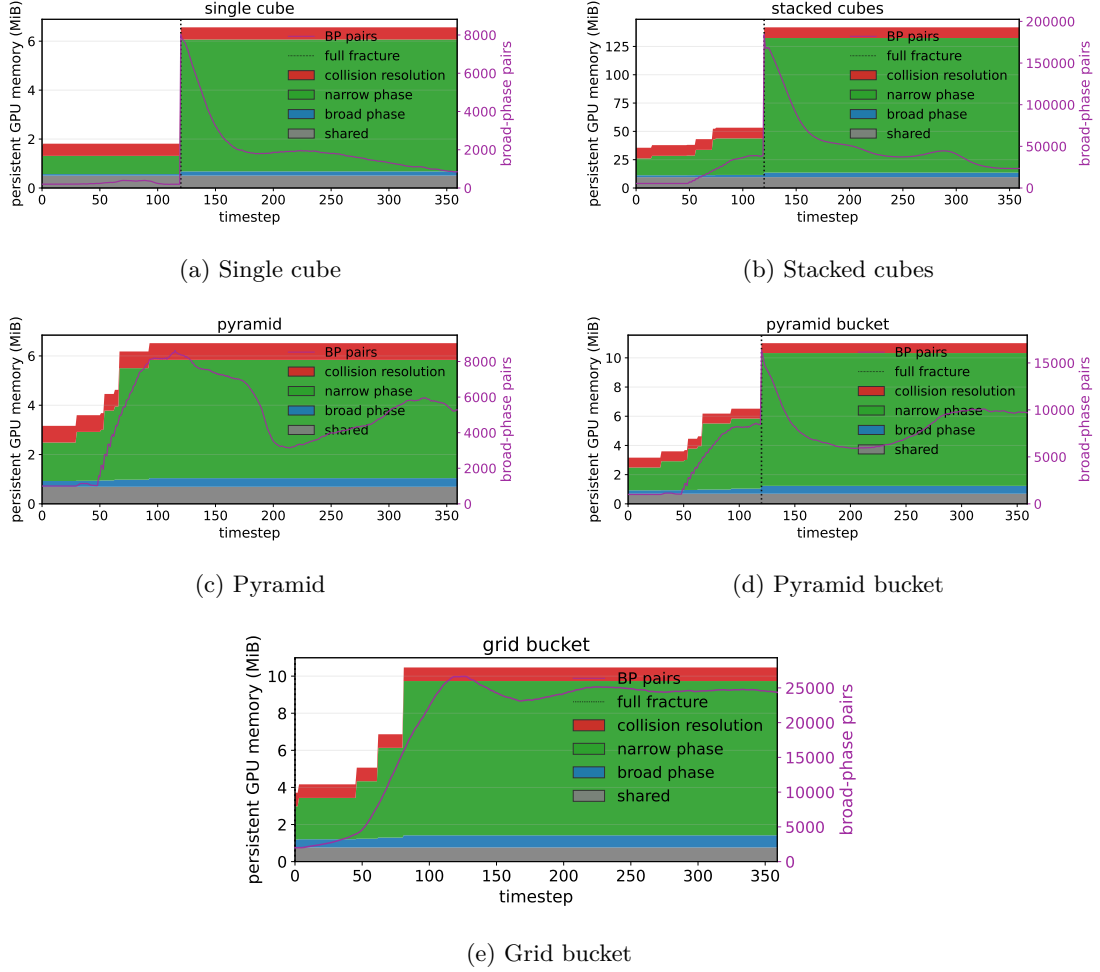


Figure 5.10: Persistent GPU memory used by the collision pipeline, excluding solver buffers. The narrow phase buffers dominate the persistent memory and grow with the number of active collision candidates. The purple line shows the broad phase candidate pair count.

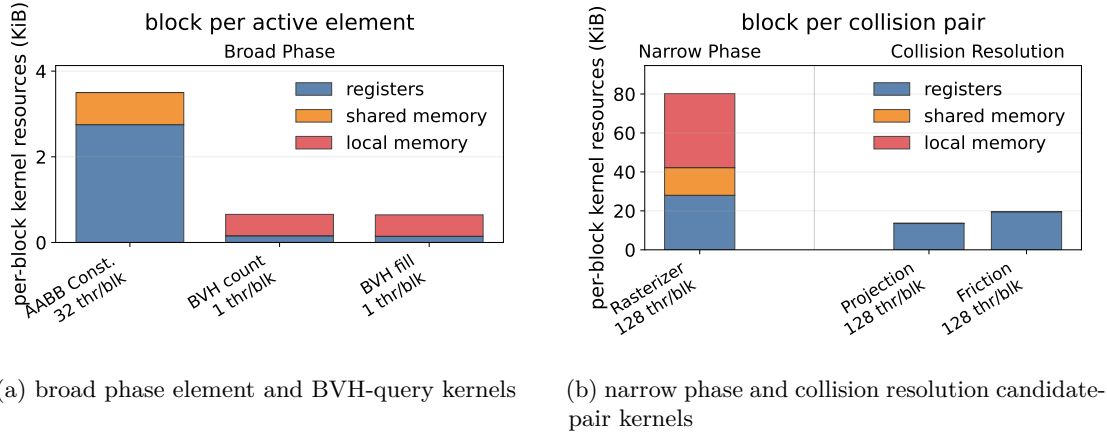


Figure 5.11: Per-block execution resources for collision kernels, separated by launch configuration. Bars show individual kernels with register, shared-memory, and local-memory contributions stacked. Values are shown in KiB and describe one scheduled CUDA block, not the accumulated footprint of all blocks in a launch. The x-axis labels include the number of threads per block.

if later timesteps use fewer entries. This avoids repeated allocations, but the memory should be seen as reserved capacity, not as actively used data in every timestep.

5.4 Kernel Resources

Persistent buffers do not describe the resources needed by an executing kernel. For this we analyze each kernel launch individually, where kernel attributes are queried through CUDA using `cudaFuncGetAttributes()` and combined with the launch configuration to obtain a per-block footprint for registers, shared memory, and local memory. Registers and shared memory are on-chip resources while local memory is private per-thread storage that may be backed by global device memory. These values therefore complement the persistent-memory measurements and cannot simply be added to them like other global allocations.

5.4.1 Results

Figure 5.11 separates kernels by launch configuration. Broad phase kernels are launched over active elements, see Section 4.3, with 32 and one thread(s) per block respectively, while narrow phase and collision resolution kernels are launched per candidate pair, see Sections 4.4.2 and 4.5, with 128 threads per block.

The plotted values are normalized to one block rather than accumulated over the full grid of blocks. For instance, the main narrow phase bar, the rasterizer, can be read as the resource footprint of one concurrently scheduled candidate-pair block, not as the total memory required for all candidate pairs in the simulation. The broad phase section,

shown in Subfigure 5.11a, should be read together with the thread counts in the x-axis labels since they differ. AABB construction uses a 32-thread cooperative block, whereas the traversal kernels use one-thread blocks, for reasons mentioned in Section 4.3.3.

5.4.2 Discussion

The traversal kernels show comparatively high local memory usage because the cuBQL traversal implementation requires per-thread traversal information. In the pair-level plot, shown in Subfigure 5.11b, the rasterizer has the largest per-block footprint due to its per-pixel fragment lists and triangle batches, while collision resolution mostly uses registers because its 128-thread blocks hold several intermediate data. These per-block resources determine how many blocks can reside on a streaming multiprocessor simultaneously and affect occupancy and therefore runtime. However, they do not accumulate over every block in a launch, since the GPU only schedules additional blocks as resources come available. Provided that at least one block can reside on a streaming multiprocessor, the kernel resources therefore do not directly limit the total number of collision pairs that can be processed.

Consequently, when we consider these results together with the persistent-memory measurements from Section 5.3, the persistent allocations represent the practical limit on scene size. The largest evaluated scene, the stacked cubes scene, contains 20 000 elements and reaches approximately 850 MiB of persistent GPU memory at its recorded peak, including solver memory. If its element-to-memory ratio remained the same, a linear extrapolation to the theoretical 8 GB of VRAM on the tested machine would correspond to a rough estimate of 190,000 elements. For scenes with contact behavior similar to the benchmark scenes, the solver buffers would reach the memory limit first because they account for approximately 700 MiB of the measured 850 MiB peak.

5.5 Pairwise Overlap Volume

The image-based narrow phase estimates an overlap volume for every broad phase candidate pair. For this evaluation, the pairwise values are accumulated, normalized by the total rest-state volume of the object(s), recorded per solver iteration, and averaged per timestep. The result is not a unique geometric intersection volume of the full scene, since regions involved in several pairwise contacts can contribute multiple times. Here, the metric is mainly used to observe how the detected pairwise overlap evolves and whether the collision pipeline reduces it over time.

5.5.1 Results

Figure 5.12 shows the normalized pairwise overlap volume over time. Across the scenes, the curves generally decrease after contact, indicating that the collision response reduces the detected overlap instead of allowing it to grow. Full fracture events create visible jumps because additional active elements introduce new contacts. The pyramid and

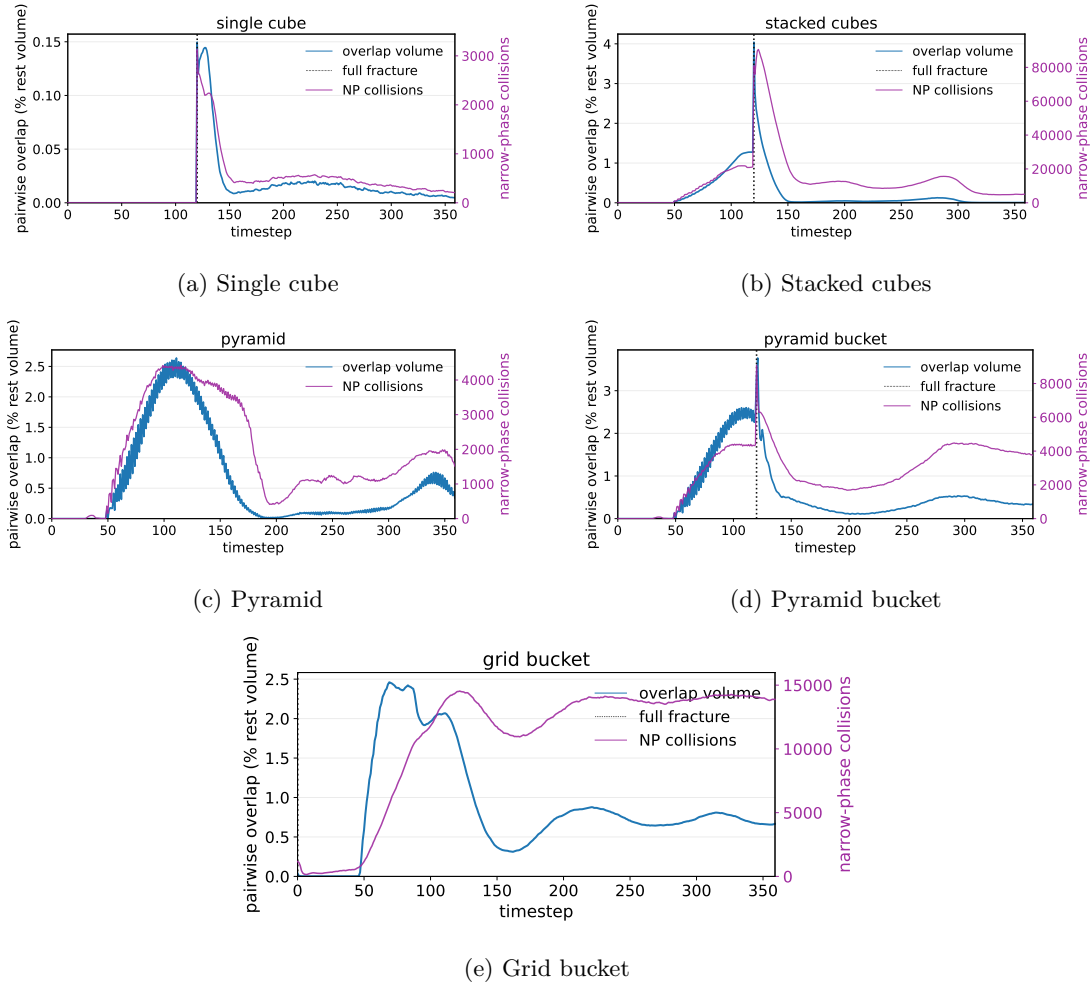


Figure 5.12: Detected pairwise overlap volume over time, normalized by the total rest-state volume of the simulated elements.

pyramid-bucket scenes also show repeated spikes, which correspond to changes in the contact configuration during compression, such as deeper penetration or additional contact pairs.

5.5.2 Discussion

The pairwise overlap volume helps judge the behavior of the collision pipeline. The expected outcome is not that the overlap remains zero at every timestep, since compression and newly created contacts introduces fresh overlap, but that these increases are followed by a reduction. The rise in overlap volume should therefore be read as changes in the scene state rather than as failures of the method: more contact pairs or stronger compression can all increase the summed pairwise overlap before the collision response

reduces it again.

The spiky behavior in the pyramid curves remains a limitation of the current analysis. Notably, a similar repeated pattern appears in the collision normal deviation analyzed in Section 5.10, which could be related to the same image-based approximation effects. However, this remains a hypothesis rather than a confirmed cause.

5.6 Narrow-Phase Accuracy

We compare the image-based narrow phase detection accuracy against a tetrahedron-tetrahedron intersection reference using the Separating-Axis-Theorem (SAT) [GLM96]. We use tetrahedra because SAT provides an independent and well-defined reference for their intersection. Our image-based method does not use a tetrahedron-specific path, but processes tetrahedral elements in the same way as general polyhedral elements. The test is therefore representative of how our narrow phase detects collisions while providing a reference against which we can classify its results.

We first use PolyDiM’s tetrahedral mesh generator to discretize a unit cube into 1,000 tetrahedra. The discretized cube serves as a pool from which we can obtain non-degenerate tetrahedra with different shapes. For each test case, we randomly select two tetrahedra from the template mesh, rotate them about their centroids, and translate them such that their AABBs overlap. We arrange 10,000 such pairs with sufficient distance between them to prevent unintended cross-pair contacts, as shown in Figure 5.13. Consequently, every intended pair reaches the narrow phase, and the broad phase produces exactly the expected 10,000 candidate pairs.

We evaluate the geometry over a range of raster resolutions from 1×1 to 256×256 . SAT tests the tetrahedral face-normal and edge-cross-edge axes and classifies only strictly positive intersection as a collision. The image-based method follows the criterion used by the simulation and confirms a collision if its approximated overlap volume is greater than zero.

5.6.1 Results

A true positive (TP) is detected by both methods, while a true negative (TN) is rejected by both. A false positive (FP) is detected only by the image-based method, and a false negative (FN) is a SAT intersection missed by the image-based method. Based on these counts, the metrics shown below are

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}, \quad \text{precision} = \frac{TP}{TP + FP}, \quad (5.1)$$

$$\text{recall} = \frac{TP}{TP + FN}, \quad F_1 = 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \quad (5.2)$$

Accuracy measures the fraction of all correctly classified pairs. Precision measures how many image-based detections are SAT intersections, while recall measures how many

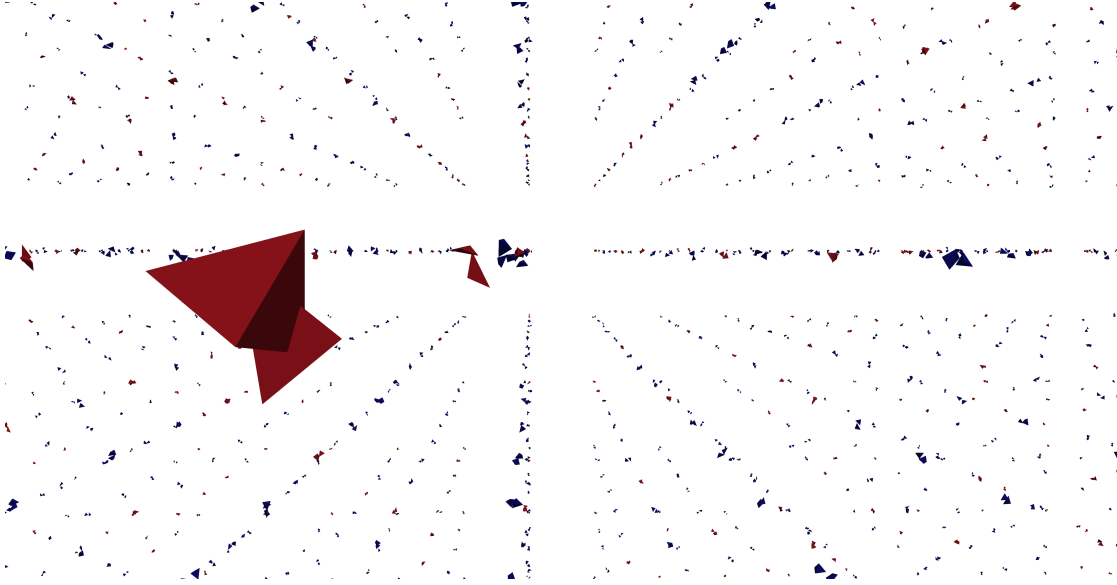


Figure 5.13: Accuracy test scene containing 10,000 spatially separated tetrahedron pairs. A pair in red indicates a SAT Intersection, blue pairs do not intersect.

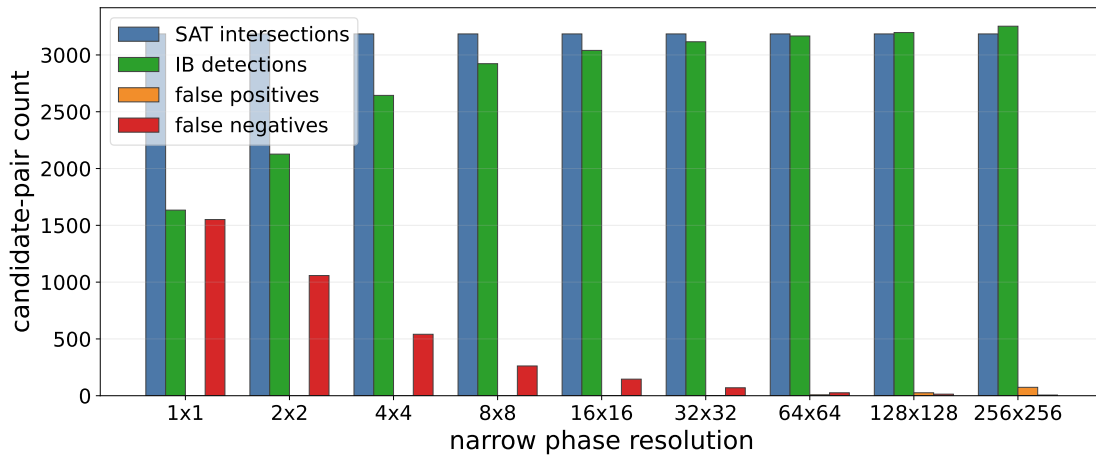


Figure 5.14: SAT intersections, image-based detections, false positives, and false negatives over 10,000 AABB-overlapping tetrahedron pairs for each raster resolution.

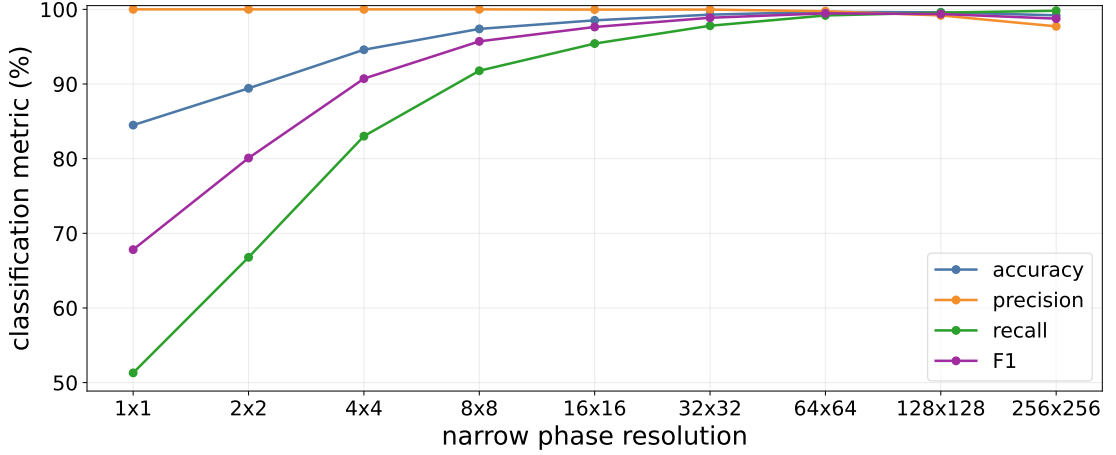


Figure 5.15: Accuracy, precision, recall, and F1 score of the image-based narrow phase relative to the tetrahedral SAT reference for each raster resolution.

SAT intersections are correctly found by the image-based method. The F_1 score captures the balance between precision and recall.

Figures 5.14 and 5.15 compare the detections and classification metrics. SAT identifies 3,185 intersecting and 6,815 separated pairs. Increasing the resolution consistently reduces the number of false negatives, from 1,551 at 1×1 to six at 256×256 . No false positives occur up to 8×8 . Their number remains low at intermediate resolutions, with one at 16×16 and 32×32 and eight at 64×64 , before increasing to 26 at 128×128 and 74 at 256×256 .

The 64×64 configuration provides the highest overall accuracy of 99.66%, with 26 false negatives and eight false positives. The 256×256 configuration reaches the highest recall of 99.81%, but its 74 false positives reduce the overall accuracy to 99.20%.

5.6.2 Discussion

The decreasing false-negative count shows that a finer raster resolution detects more actual intersections. At low resolutions, the low number of pixels can miss small overlap regions, increasing the resolution therefore improves recall. In this test set, 64×64 gives the best balance between false negatives and false positives. Higher resolutions reduce the remaining false negatives further, but no longer improve the overall classification accuracy since some false positives are introduced.

A false positive in this comparison means that the SAT reference finds a separating axis while the image-based method returns a positive overlap volume. One possible explanation is that, in rare cases, the rasterizer does not record the entry and exit fragments along a pixel ray correctly. A regular ray through a closed element should produce one entry and one exit fragment, but a pixel center lying directly on a projected

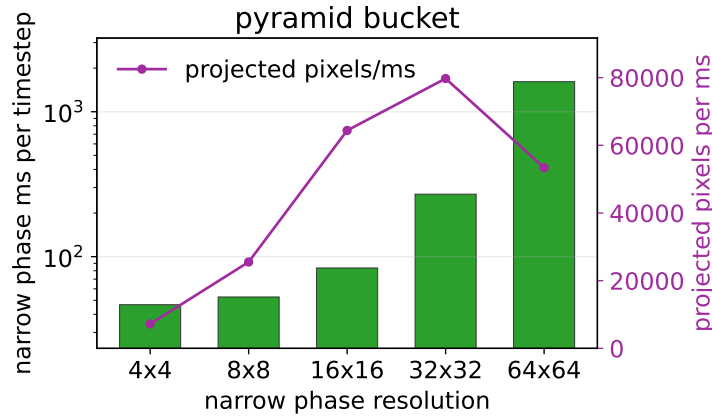


Figure 5.16: Influence of the image-based narrow phase resolution on runtime for the pyramid-bucket scene. Bars show the average narrow phase runtime per timestep, while the line shows the estimated projected raster pixels processed per millisecond.

triangle edge can cause one fragment to be generated as well as an adjacent triangle to produce an additional one. The resulting erroneous fragment can leave the inside state of one element wrongfully active when the ray enters the second element, causing our method to accumulate a small nonexistent overlap interval. The increasing number of false positives at high resolutions is consistent with more pixels potentially encountering such rare cases, but this does not by itself confirm this explanation.

5.7 Narrow-Phase Resolution

To analyze the influence of the image resolution on narrow phase runtime, we repeat the pyramid-bucket scene with resolutions from 4×4 to 64×64 while keeping the remaining scene setup unchanged. The plotted throughput contains the number of narrow phase pairs times the three projection directions times the resolution-dependent number of pixels per projection.

5.7.1 Results

Figure 5.16 shows the average narrow phase runtime per timestep. Runtime increases with resolution, as expected from the quadratic growth in pixel count. The projected-pixel throughput increases up to 32×32 , where the narrow phase processes the largest estimated raster workload per millisecond. It decreases again at 64×64 , while the absolute runtime increases substantially.

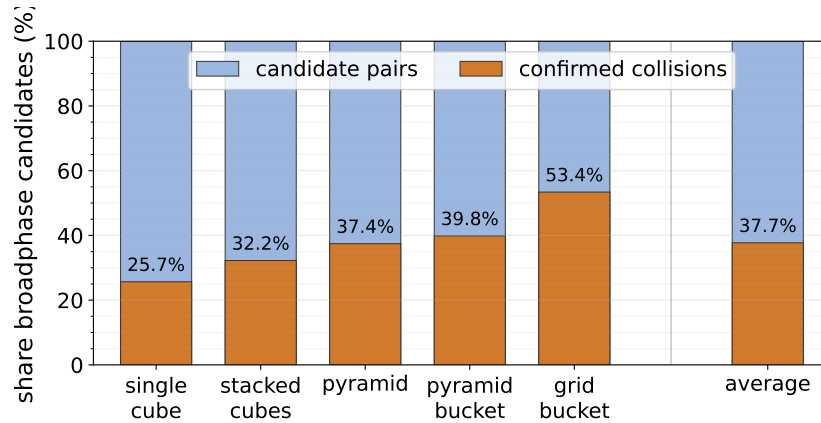


Figure 5.17: Average narrow phase confirmation ratio for broad phase candidate pairs. Candidate pairs are normalized per scene, and confirmed collisions are candidate pairs with detected overlap volume. The rightmost bar shows the average confirmed collision ratio over all scenes.

5.7.2 Discussion

The resolution Figure 5.16 shows that the narrow phase does not directly scale with raw pixel count. Very small images do not fully amortize the full grid, while very large images make each candidate pair substantially more expensive to process. The 32×32 case has the best measured projected-pixel throughput in this scene, but this is not automatically the best simulation setting. The final resolution still has to find a trade-off between runtime and detection accuracy depending on the use case.

5.8 Confirmed Collisions Ratio

The confirmed collision ratio gives an empirical indication of how effective AABB culling is for deforming Voronoi polyhedral cells. This is useful since relatively little work focuses specifically on collision handling for deformable polyhedral meshes. The measurement therefore quantifies how many broad phase candidate pairs, based on AABB overlaps, remain intersecting after the image-based narrow phase has approximated their overlap volume.

5.8.1 Results

Figure 5.17 shows the average ratio between broad phase candidate pairs and confirmed narrow phase collisions for all five test scenes. Each scene bar represents the broad phase candidates as 100%, while the confirmed-collision in orange shows the fraction for which the narrow phase detected an overlap volume. The separated bar on the right shows the average over all scenes.

Table 5.3: Discretization and material parameters used for the friction validation.

Elements/Object	λ	μ	α	β
10	1000.0	800.0	0.5	0.8

5.8.2 Discussion

The confirmation ratio shows that a considerable fraction of broad phase candidates does not lead to overlap. These candidate pairs still require a narrow phase invocation in the current pipeline. Since the image-based narrow phase is the most expensive stage, reducing unnecessary narrow phase launches is a promising optimization direction. A cheaper intermediate culling phase between broad phase and narrow phase could be considered in future optimizations.

5.9 Friction Validation

Friction is evaluated with a two-object test. A lower cube is fixed with Dirichlet boundary conditions and acts as the contact surface, while a smaller cube is placed on top of it. Instead of rotating the geometry, gravity is tilted so that the surface acts like a 30° tilted slope. For Coulomb friction, the angle of friction is

$$\mu_{\text{crit}} = \tan(30^\circ) \approx 0.577. \quad (5.3)$$

Coefficients below this value should slide, while coefficients above it should remain at rest. Table 5.3 lists the discretization and material parameters for this scene.

5.9.1 Results

Figure 5.18 shows the resulting motion for different friction coefficients. The qualitative trend is correct: low coefficients slide down the slope, while high coefficients strongly reduce sliding. The transition is shifted however. In this setup, $\mu_c = 0.5$ clearly slides and $\mu_c = 0.6$ still does not fully stick. Approximate sticking is observed only around $\mu_c = 0.75$ –0.8 and above, with slight residual drift potentially caused by observed elastic wobbling due to the material properties.

5.9.2 Discussion

The test supports the qualitative friction behavior but not an exact Coulomb behavior. A likely source of the shifted transition is the collision normal: the response stage uses the image-based narrow phase normal from pairwise overlap, not an analytic plane-contact normal. This is investigated further in Section 5.10. Small normal errors can change the split between normal and tangential components and therefore the effective friction limit.

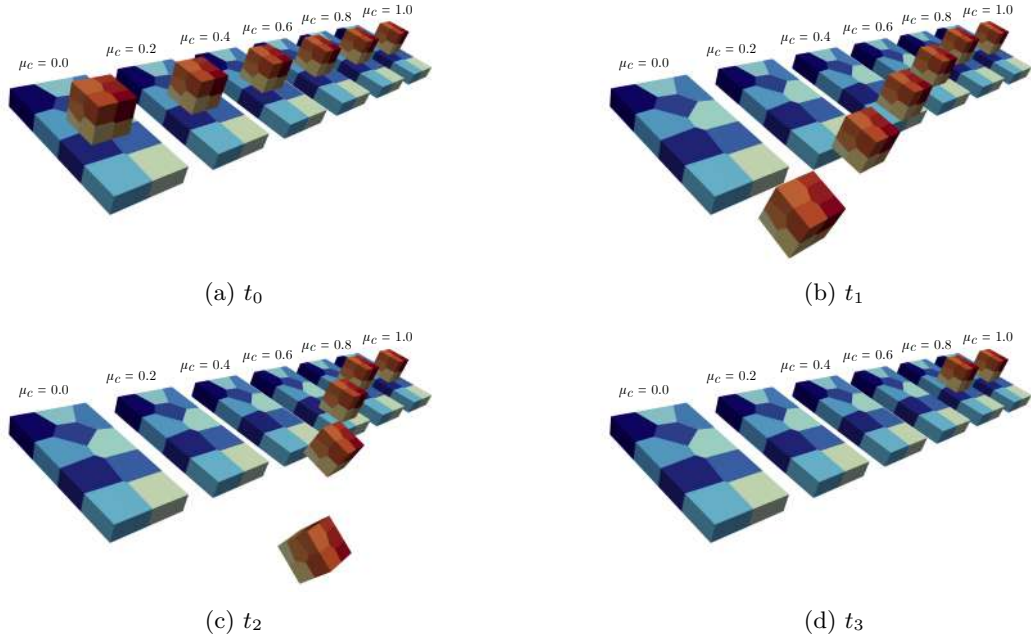


Figure 5.18: Friction validation using a fixed contact surface and tilted gravity equivalent to a 30° slope. The snapshots compare different friction coefficients over time.

Table 5.4: Discretization and material parameters used for the collision normal validation.

Elements/Object	λ	μ	α	β
10	250.0	200.0	0.5	0.8

5.10 Approximate Collision Normals

To evaluate the normal approximation, a small reference scene similar to the friction validation setup is used. A cube is placed on top of a fixed cube, producing a horizontal contact section. In this configuration, the expected collision normal is aligned with the vertical axis. The exported approximate normals point in the negative z direction, so each collision normal $\mathbf{n}$ is compared against the ground-truth direction $(0, 0, -1)$ using the angular deviation

$$\theta = \arccos \left(\frac{\mathbf{n}}{\|\mathbf{n}\|} \cdot (0, 0, -1) \right). \quad (5.4)$$

The measurements use the exported collision normals over 360 timesteps. Table 5.4 lists the discretization and material parameters, and Figure 5.19 shows the reference configuration and the exported normals.

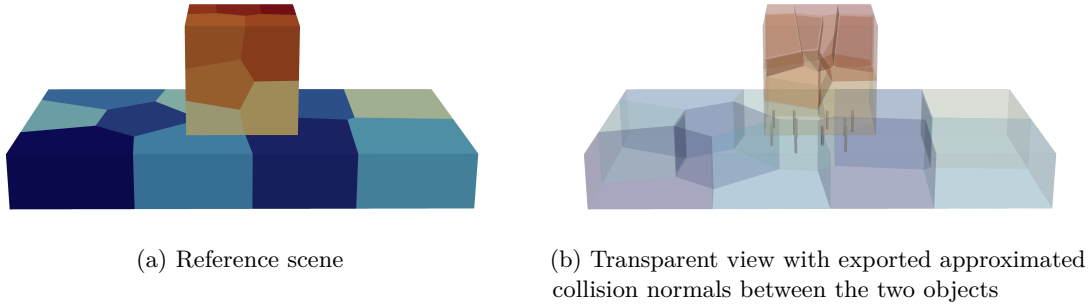


Figure 5.19: Cube-on-cube setup used to evaluate the approximated collision normals.

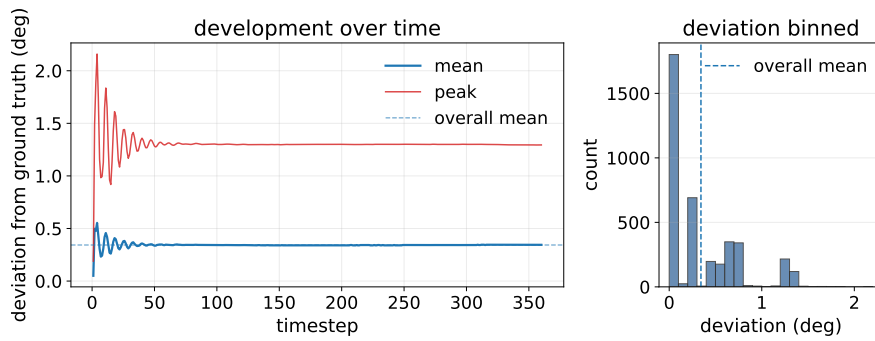


Figure 5.20: Deviation of the image-based collision normals from the expected vertical direction in the cube-on-cube contact test.

5.10.1 Results

Figure 5.20 shows the mean and peak angular deviation per timestep, together with the distribution over all exported normals. Across 3958 normals, the average deviation is 0.343° and the largest measured deviation is 2.159° at timestep 4.

5.10.2 Discussion

The normal approximation is close to the expected direction in this controlled test, but it is not exact. Deviations of one to two degrees are small visually, but they can still change the split between normal and tangential components in the friction response. This supports the interpretation from Section 5.9: the shifted friction threshold is potentially influenced by the image-based normal approximation.

5.11 Summary

In summary, the evaluation shows that the proposed pipeline detects and resolves collisions in all benchmark scenes introduced in Section 5.1. The pairwise overlap measurements

in Section 5.5 further show that the response generally reduces the detected overlap after new contacts are introduced. Most measurements use a narrow phase resolution of 8×8 , which provides a good trade-off between accuracy and performance for evaluating the benchmark scenes. The tetrahedral SAT comparison in Section 5.6 shows that increasing the resolution improves detection: 64×64 achieves the highest measured accuracy of 99.66%, whereas still higher resolutions reduce the remaining false negatives but introduce more false positives. The runtime experiment in Section 5.7 additionally finds the highest projected-pixel throughput at 32×32 . The appropriate resolution therefore depends on whether speed or detection accuracy is more important.

The runtime measurements in Section 5.2 show that, at 8×8 , the average collision pipeline runtime ranges from 27.9ms per timestep for the single cube to 338.3ms for the stacked cubes. The narrow phase accounts for roughly 71.0–91.0% of this time and is therefore the main performance bottleneck, while the broad phase contributes 6.4–22.9% and collision resolution 2.6–6.1%. Active-element classification and neighborhood filtering substantially reduce unnecessary narrow phase work, but the confirmed-collision ratio in Section 5.8 shows that further intermediate culling could still be beneficial. The persistent-memory and kernel-resource analyses in Sections 5.3 and 5.4 show that persistent allocations, rather than per-block kernel resources, form the practical memory limit. The largest evaluated scene uses approximately 850 MiB at its recorded peak including the solver, with the solver accounting for about 700 MiB. Within the collision pipeline, the pair-dependent narrow phase buffers dominate and retain their peak allocated capacity.

The validation test scenes also identify qualitative limitations. The normal approximation evaluated in Section 5.10 remains close to the reference direction in the controlled cube-on-cube test, with an average angular deviation of 0.343° . The friction test in Section 5.9 reproduces the expected qualitative trend, but the observed sticking threshold is shifted from the ideal value. Moreover, the qualitative observations in Section 5.1.1 show that dense benchmark configurations do not reach an equilibrium. Together, these findings provide the basis for the conclusions drawn in the following chapter, which also discusses the corresponding directions for future work.

Conclusion

This thesis presented a GPU-based collision handling pipeline for deformable polyhedral meshes in a VEM simulation framework. The main challenge was that the simulated objects are not represented by simple tetrahedra, but by irregular polyhedral elements whose faces can deform, become non-planar, and form non-convex shapes. As a result, many standard collision detection methods are not applicable without additional decomposition or restrictive geometric assumptions.

Our proposed pipeline combines a BVH-based broad phase, including topology-based neighborhood filtering, with an image-based narrow phase and a position-based collision resolution. The broad phase generates candidate element pairs from per-element AABBs and avoids unnecessary narrow phase tests between topologically adjacent elements. Together with the active-element filtering used by the simulation framework, this significantly reduces the number of narrow phase invocations. The evaluation confirms the importance of this filtering because the narrow phase is the dominant runtime cost, while the confirmed collision ratio indicates that further culling could reduce the remaining candidate pairs, as discussed in Sections 5.2 and 5.8.

For narrow phase detection, our work adapted the image-based volume approach of Faure et al. [FBAF08] to the GPU simulation setting by implementing a custom CUDA compute rasterizer. The method operates pair-locally on the AABB intersection region, rasterizes centroid-fan triangles into depth-sorted fragment lists, and estimates pairwise overlap volume, volume gradients, and collision normals. This makes it possible to handle potentially non-convex polyhedral cells without relying on convex decomposition. The comparison against a tetrahedral SAT reference confirms the detection accuracy of the method, while the resolution study shows the trade-off between detection accuracy and runtime; see Sections 5.6 and 5.7.

The collision response uses the approximate overlap volume and gradients in a position-based projection step. This directly reduces interpenetration during the current iteration

and feeds the applied correction back into the solver as collision forces. The pairwise overlap measurements show that the response generally reduces the detected overlap over time. The method also approximates collision normals from the overlap gradients and uses them for Coulomb-style friction. The friction and normal validation tests show qualitatively plausible behavior, but also reveal deviations from ideal friction and small errors in the approximated normals, see Section 5.9 and 5.10. In addition, the benchmark scenes show that the current response does not always reach a static equilibrium.

Overall, the implementation demonstrates that image-based collision detection can be integrated into a GPU-based polyhedral VEM framework and can produce visually plausible collision response for irregular volumetric elements. At the same time, the evaluation makes clear where the method remains approximate: finite raster resolution can lead to missed or false detections, the centroid-fan surface representation assumes sufficiently regular face geometry, and the approximate collision normals and current response formulation affect friction and settling behavior. The runtime, memory, and kernel resource measurements also show that there is room for performance optimization, especially in the narrow phase. The result should therefore be understood as a promising foundation for collision handling on deformable polyhedral meshes, with several clear directions for further improvement.

6.1 Future Work

The most immediate optimization target is the number of narrow phase invocations. The evaluation shows that many broad phase candidate pairs are rejected only after the expensive image-based narrow phase, see Section 5.8. Introducing an intermediate culling stage could therefore be beneficial. For example, a mid phase based on convex hulls could reject additional pairs before rasterization. Since the narrow phase dominates runtime, even a moderately effective and efficient mid phase could greatly improve performance.

The collision resolution itself could also be revisited. The current position-based correction is inexpensive and fits naturally into the existing iterative solver, but it remains an approximate response. As observed in Section 5.1.1, the benchmark simulations do not always reach an equilibrium, with the remaining motion becoming particularly visible in confined configurations with dense contact. Future work could therefore investigate barrier method contact formulations or other constraint formulations that provide stronger non-penetration guarantees. Such methods would likely be more expensive, but could provide a more robust response in persistent contact configurations.

Another direction revolves around the surface representation used for rasterization. The centroid-fan triangulation is simple and symmetric, but it assumes that face geometry remains sufficiently regular. For non-convex faces or strongly deformed faces, alternative triangulations such as rest-space ear clipping with fixed connectivity could be investigated. Similarly, the current redistribution of centroid gradient contributions to face vertices uses uniform weights; non-uniform weighting schemes could be investigated.

The friction model would benefit from more accurate collision normals. The current normal approximation is derived from the image-based overlap computation, which is convenient because it does not require separate geometric contact computations. However, the normal validation in Section 5.10 suggests that small deviations occur and can potentially lead to unwanted behavior later in the pipeline, as also observed in the friction validation in Section 5.9.

The broad phase could be improved further by exploiting temporal coherence. Instead of refitting or rebuilding the BVH based only on the current configuration, AABBs could be expanded according to the velocity of the elements. This could reduce the need for hierarchy updates, although it would also increase the number of broad phase candidate pairs and would therefore have to be balanced against the additional narrow phase costs.

The discrete time aspect of the current pipeline also creates room for continuous collision detection (CCD). The current method evaluates collisions only at discrete configurations, which means that fast motion of elements can still miss collisions between timesteps. A possible approach would be to construct swept AABBs from previous and current element positions and use them in the broad phase. Candidate pairs could then be tested at interpolated configurations to estimate the time of impact more accurately before applying collision response.

Finally, the pipeline could be extended to self-collision handling within individual polyhedral cells. The current pipeline detects collisions between element pairs, but it does not address cases where a strongly deformed element intersects itself. This would require additional tests on the surface representation of a single element and a response formulation that maps the resulting correction back to the element vertices without introducing unstable internal forces.

Overview of Generative AI Tools Used

OpenAI's ChatGPT 5.3 was used during the research phase to help with the gathering and understanding of scientific papers. The tool was used as a conversational partner to discuss papers section by section and to clarify concepts that were difficult to understand.

Anthropic's Claude Code with Model *Opus 4.6* and *OpenAI's Codex* with model *GPT-5.5* were used during the implementation phase to debug and optimize CUDA and C++ code. Helper functions, for instance for exporting geometry data like normals or triangulations as `.vtk` files or intermediate rasterization textures for visual debugging, were generated using these aitoools.

OpenAI's Codex with model *GPT-5.5* was used during the evaluation phase to help with the creation of helper functions for exporting measurement and evaluation data to `.csv` files. The tool was also used to generate Python scripts for transforming the exported data into figures.

OpenAI's Codex with model *GPT-5.5* was used during the writing phase as a language-editing assistant to improve formulations and readability. The structure and content of each chapter or section were outlined manually first. Based on the provided content the ai-generated formulations were then repeatedly reviewed, revised, and iterated upon to ensure that the final text remained faithful to the original content and only the tone and formulations were adapted.

List of Figures

1.1	Simulation of a fractured cuboid stack	3
2.1	BVH over Voronoi element AABBs	7
2.2	Body relationships expressed by the gap function	11
3.1	Collision handling pipeline overview	18
3.2	Pair-local narrow phase view volume	20
3.3	Collision-active elements and centroid-fan triangulation	21
3.4	Depth-sorted fragment list	24
3.5	Overlap texture visualization	25
3.6	Approximate collision normals	27
4.1	Indexing of irregular element data	33
5.1	Single-cube benchmark scene	44
5.2	Benchmark scene snapshots	45
5.3	Full GPU runtime breakdown	46
5.4	Single-cube runtime	46
5.5	Stacked-cubes runtime	47
5.6	Pyramid runtime	47
5.7	Pyramid-bucket runtime	47
5.8	Grid-bucket runtime	47
5.9	Persistent GPU memory including solver buffers	49
5.10	Persistent collision-pipeline memory	50
5.11	Per-block collision-kernel resources	51
5.12	Pairwise overlap volume	53
5.13	Tetrahedral SAT accuracy test scene	55
5.14	Tetrahedral SAT and image-based detection counts	55
5.15	Tetrahedral SAT and image-based classification metrics	56
5.16	Narrow phase resolution and runtime	57
5.17	Broad phase candidate confirmation ratio	58
5.18	Friction validation	60
5.19	Collision normal validation setup	61
5.20	Collision normal deviation	61

List of Tables

5.1	Evaluation scene parameters	44
5.2	Collision pipeline runtime by scene	46
5.3	Friction validation parameters	59
5.4	Collision normal validation parameters	60

List of Algorithms

4.1	One iteration of the collision handling pipeline.	32
4.2	block reduction of thread values.	34
4.3	Two-pass BVH traversal.	35
4.4	Rasterizer axis loop for one candidate-pair block.	37

Bibliography

- [AEF22] Sheldon Andrews, Kenny Erleben, and Zachary Ferguson. Contact and friction simulation for computer graphics. In *ACM SIGGRAPH 2022 Courses*, SIGGRAPH '22, New York, NY, USA, 2022. Association for Computing Machinery. DOI: 10.1145/3532720.3535640.
- [Bar89] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, page 223–232, New York, NY, USA, 1989. Association for Computing Machinery. DOI: 10.1145/74333.74356.
- [BBTV25] Stefano Berrone, Andrea Borio, Gioana Teora, and Fabio Vicini. GEDiM: GEometry for DIcretization MEthod library, May 2025. DOI: 10.5281/zenodo.15481736.
- [BBTV26] Stefano Berrone, Andrea Borio, Gioana Teora, and Fabio Vicini. POLY-DIM: A C++ library for POLYtopal DIcretization Methods. *Computer Physics Communications*, 320:109937, 2026. DOI: 10.1016/j.cpc.2025.109937.
- [BET14] Jan Bender, Kenny Erleben, and Jeff Trinkle. Interactive Simulation of Rigid Body Dynamics in Computer Graphics. *Computer Graphics Forum*, 33(1):246–270, 2014. DOI: 10.1111/cgf.12272.
- [BWS98] G. Baciú, Wingo Sai-Keung Wong, and Hanqiu Sun. RECODE: an image-based collision detection algorithm. In *Proceedings Pacific Graphics '98. Sixth Pacific Conference on Computer Graphics and Applications (Cat. No.98EX208)*, pages 125–133, 1998. DOI: 10.1109/PCCGA.1998.732079.
- [CL16] G. Capannini and T. Larsson. Adaptive collision culling for large-scale simulations by a parallel sweep and prune algorithm. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '16, page 1–10, Goslar, DEU, 2016. Eurographics Association. DOI: 10.2312/pgv.20161177.

- [CLMP95] Jonathan Cohen, Ming Lin, Dinesh Manocha, and Madhav Ponamgi. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. *Proceedings of ACM Interactive 3D Graphics Conference*, 1, 03 1995. DOI: 10.1145/199404.199437.
- [EL01] Stephen A. Ehmann and Ming C. Lin. Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition. *Computer Graphics Forum*, 20(3):500–511, 2001. DOI: 10.1111/1467-8659.00543.
- [FBAF08] François Faure, Sébastien Barbier, Jérémie Allard, and Florent Falipou. Image-based collision detection and response between arbitrary volume objects. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, page 155–162, Goslar, DEU, 2008. Eurographics Association. DOI: 10.5555/1632592.1632615.
- [FPRJ00] Sarah Frisken, Ronald Perry, Alyn Rockwood, and Thouis Jones. Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics. *ACM SIGGRAPH*, 07 2000. DOI: 10.1145/344779.344899.
- [Fü26] Füricht Elias, Hahn David. Project Report: Fast Polyhedral Dynamics, 2026. Internal Student Project Report.
- [GDY25] Chris Giles, Elie Diaz, and Cem Yuksel. Augmented Vertex Block Descent. *ACM Trans. Graph.*, 44(4), July 2025. DOI: 10.1145/3731195.
- [GI93] M. A. Ganter and B. P. Isarankura. Dynamic Collision Detection Using Space Partitioning. *Journal of Mechanical Design*, 115(1):150–155, 03 1993. DOI: 10.1115/1.2919312.
- [GJK88] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 4(2):193–203, 1988. DOI: 10.1109/56.2083.
- [GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: a hierarchical structure for rapid interference detection. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 171–180, New York, NY, USA, 1996. Association for Computing Machinery. DOI: 10.1145/237170.237244.
- [Hah26] Hahn David. Notes on the Virtual Element Method (VEM), 2026. Unpublished Internal Communication Document.
- [HTG03] Bruno Heidelberger, Matthias Teschner, and Markus Gross. Real-Time Volumetric Intersections of Deforming Objects. volume 2003, pages 461–468, 01 2003.

- [HTG04] Bruno Heidelberger, Matthias Teschner, and Markus Gross. Detection of Collisions and Self-collisions Using Image-space Techniques. *Journal of WSCG*, 12, 01 2004.
- [Hub95] P.M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995. DOI: 10.1109/2945.466717.
- [KGS98] Dong-Jin Kim, L.J. Guibas, and Sung-Yong Shin. Fast collision detection among multiple moving spheres. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):230–242, 1998. DOI: 10.1109/2945.722297.
- [KHI⁺07] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and Richard Rowe. Collision detection: A survey. In *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 4046–4051, 2007. DOI: 10.1109/ICSMC.2007.4414258.
- [KHM⁺98] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Vehicular Communications*, 4(1):21–36, January 1998. DOI: 10.1109/2945.675649.
- [LFS⁺20] Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. Incremental potential contact: intersection-and inversion-free, large-deformation dynamics. *ACM Trans. Graph.*, 39(4), August 2020. DOI: 10.1145/3386569.3392425.
- [LK13] Fuchang Liu and Young Kim. Exact and Adaptive Signed Distance Fields Computation for Rigid and Deformable Models on GPUs. *IEEE transactions on visualization and computer graphics*, 12 2013. DOI: 10.1109/TVCG.2013.268.
- [LMM10] C. Lauterbach, Q. Mo, and D. Manocha. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Computer Graphics Forum*, 29(2):419–428, 2010. DOI: 10.1111/j.1467-8659.2009.01611.x.
- [MHHR07] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *J. Vis. Comun. Image Represent.*, 18(2):109–118, April 2007. DOI: 10.1016/j.jvcir.2007.01.005.
- [MMC16] Miles Macklin, Matthias Müller, and Nuttapong Chentanez. XPBD: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*, MIG ’16, page 49–54. Association for Computing Machinery, 2016. DOI: 10.1145/2994258.2994272.

- [MOK95] Karol Myszkowski, Oleg G. Okunev, and Toshiyasu L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11:497–511, 1995. DOI: 10.1007/BF02439645.
- [NOB16] Rahul Narain, Matthew Overby, and George E. Brown. ADMM $\supseteq$ projective dynamics: Fast simulation of general constitutive models. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '16, pages 21–28. Eurographics Association, 2016. DOI: 10.2312/sca.20161219.
- [NVI26] NVIDIA Corporation. NVIDIA’s Thrust. <https://github.com/NVIDIA/cccl>, 2026. Part of the CUDA Core Compute Libraries (CCCL). Accessed: 2026-07-26.
- [PF01] Ronald N. Perry and Sarah F. Frisken. Kizamu: a system for sculpting digital characters. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001. DOI: 10.1145/383259.383264.
- [Pin88] Juan Pineda. A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph.*, 22(4):17–20, June 1988. DOI: 10.1145/378456.378457.
- [SF91] Mikio Shinya and Marie-Claire Forgue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):132–134, 1991. DOI: 10.1002/vis.4340020408.
- [SGK18] Breannan Smith, Fernando De Goes, and Theodore Kim. Stable Neo-Hookean Flesh Simulation. *ACM Trans. Graph.*, 37(2), March 2018. DOI: 10.1145/3180491.
- [SR19] Ygor Rebouças Serpa and Maria Andréia Formico Rodrigues. Flexible Use of Temporal and Spatial Reasoning for Fast and Scalable CPU Broad-Phase Collision Detection Using KD-Trees. *Computer Graphics Forum*, 2019. DOI: 10.1111/cgf.13529.
- [TBW09] Daniel J. Tracy, Samuel R. Buss, and Bryan M. Woods. Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal. In *2009 IEEE Virtual Reality Conference*, pages 191–198, 2009. DOI: 10.1109/VR.2009.4811022.
- [TKH⁺05] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino. Collision Detection for Deformable Objects. *Computer Graphics Forum*, 24(1):61–81, 2005. DOI: 10.1111/j.1467-8659.2005.00829.x.
- [TMY⁺11] Min Tang, Dinesh Manocha, Sung-Eui Yoon, Peng Du, Jae-Pil Heo, and Ruofeng Tong. VolCCD: Fast continuous collision culling between deforming

- volume meshes. *ACM Trans. Graph.*, 30(5), October 2011. DOI: 10.1145/2019627.2019630.
- [vdB98] Gino van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1–13, January 1998. DOI: 10.1080/10867651.1997.10487480.
- [Wal26] Ingo Wald. cuBQL: A CUDA BVH build-and-query library. <https://github.com/NVIDIA/cuBQL>, 2026. Accessed: 2026-06-16.
- [WC21] Monan Wang and Jiaqi Cao. A review of collision detection for deformable objects. *Computer Animation and Virtual Worlds*, 32(5):e1987, 2021. DOI: 10.1002/cav.1987.
- [WDZ17] René Weller, Nicole Debowski, and Gabriel Zachmann. kDet: Parallel Constant Time Collision Detection for Polygonal Objects. *Computer Graphics Forum*, 36(2):131–141, 2017. DOI: 10.1111/cgf.13113.
- [WLZ14] Tsz Ho Wong, Geoff Leach, and Fabio Zambetta. An adaptive octree grid for GPU-based collision detection of deformable objects. *The Visual Computer*, 30(6–8):729–738, June 2014. DOI: 10.1007/s00371-014-0954-1.
- [WRRH17] P. Wriggers, B. D. Reddy, W. Rust, and B. Hudobivnik. Efficient virtual element formulations for compressible and incompressible finite deformations. *Computational mechanics*, 60(2):253–268, April 2017. DOI: 10.1007/s00466-017-1405-4.
- [WTMT18] Xinlei Wang, Min Tang, Dinesh Manocha, and Ruofeng Tong. Efficient BVH-based Collision Detection Scheme with Ordering and Restructuring. *Computer Graphics Forum*, 37(2):227–237, 2018. DOI: 10.1111/cgf.13356.
- [XPW24] Bing-Bing Xu, Fan Peng, and Peter Wriggers. Stabilization-free virtual element method for 3D hyperelastic problems. *Comput. Mech.*, 75(6):1687–1701, May 2024. DOI: 10.1007/s00466-024-02501-4.
- [Yu22] Yue Yu. mVEM: A MATLAB Software Package for the Virtual Element Methods. <https://arxiv.org/abs/2204.01339>, 2022.

Appendix A: Notes on the Virtual Element Method

This appendix contains the internal notes by David Hahn [Hah26], which provide a more detailed derivation of the Virtual Element Method simulation framework underlying this thesis.

Notes on the Virtual Element Method (VEM)

DAVID HAHN

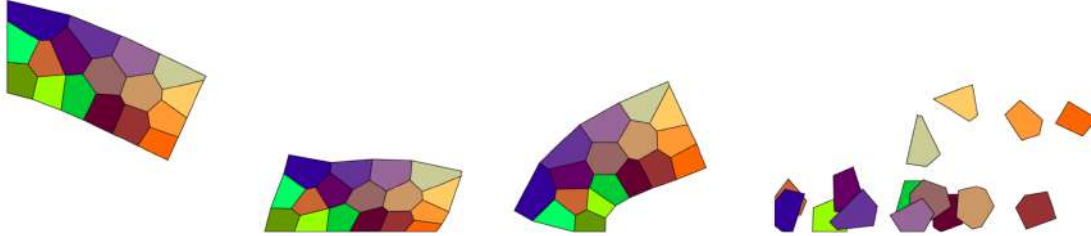


Fig. 1. Example frames from a 2D simulation: a rectangular object hangig from a wall (left) is then dropped (removing boundary conditions), bounces on the floor with friction (middle), and eventually breaks into separate pieces (removing constraints that “glue” neighbouring elements together, right).

This is a collection of internal notes describing the virtual element method in 3D using linear displacement approximation and hyperelastic material models. We employ a very basic VEM formulation (§1) with “dof-dof” stabilization term, built by projecting into the linear monomial basis, that penalizes all non-linear deformations, but is exactly invariant to (linearized) rotation and translation. From there, we formulate a parallel Gauss-Seidel position-based solver (§2) to simulate dynamic elasticity.

1 Linear Virtual Elements

The virtual element method (VEM) is a generalization of the well-known finite element method (FEM) that allows for more flexibility in terms of element and mesh topology. Most notably, it can support polygonal (2D) and polyhedral (3D) elements, and directly handle T-junctions (or hanging nodes) in the mesh. On the flip side, the basis functions for these elements are not defined (or available) in closed form, so evaluating the solution in the interior of an element is somewhat more involved. We note that more advanced VEM formulations exists, see e.g., [Xu et al. 2024], but here we focus on a basic linear version that is relatively easy to understand and implement, and guarantees invariance to rigid motion (i.e., does not cause any ghost forces under large-scale motion of elements). The formulation we summarize in the following is primarily based on the mVEM code [Yu 2022] (although we do not directly use any of their code, but reimplement the method for C++/CUDA and also present snippets from a Matlab demo implementation). The key idea of linear VEM—in terms of simulating elastic deformations—is to find a piecewise constant approximation (average) of the deformation gradient per element. Clearly, a general polyhedral element (as opposed to a simplicial element, such as a triangle in 2D or tetrahedron in 3D) has more deformation modes (even if all edges remain linear) than can be represented by such a deformation gradient. Therefore, these modes, which do not contribute to the elastic energy, must be penalized in order to obtain a unique solution. In the following, we first describe how to compute the deformation gradient (§1.1), evaluate the elastic energy and forces (§1.2), and finally how to construct a corresponding stablilization term §1.3 that penalizes the zero-energy modes.

1.1 Deformation gradient

The virtual element literature typically first defines an approximation space over an element and then constructs projection operators that define the element integrals in terms of the element’s degrees of freedom, such that the explicit

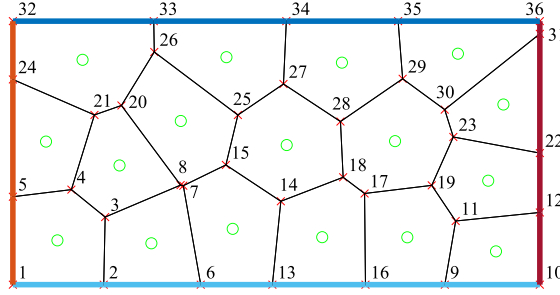


Fig. 2. Example 2D (Voronoi) mesh: elements are oriented polygons (i.e., vertex indices are listed in counter-clockwise order for each element), nodes are marked as red crosses, element centroids as green circles; edges on the boundary are grouped by colour.

definition of element basis functions (as would be used in a finite element formulation) is not required. Here, we want to avoid going into the theory of VEM and instead focus on the practical aspects that are relevant to our implementation.

As a first step, we consider a polygonal element in 2D (theoretically it is required to be star convex, but we won't go into the details here); we will extend this approach to polyhedra in 3D later. We now focus on a single element, which in 2D is simply an ordered list of vertex indices, alongside a list of corresponding vertex coordinates; Fig. 2 shows an example of a polygonal mesh (in this case a Voronoi mesh, which is a common source of polygonal meshes, but not a strict requirement). For any one element, we need to approximate the (linear) deformation, i.e., we want to know the *average* deformation gradient over the element $\mathcal{E}$:

$$\mathbf{F} = \frac{1}{|\mathcal{E}|} \int_{\mathcal{E}} \frac{d\mathbf{X}}{d\mathbf{x}} d\mathbf{x},$$

where $\mathbf{X}$ is the world-space location of a material-space point $\mathbf{x}$ and the integration proceeds over the element $\mathcal{E}$ in material space (with $|\mathcal{E}|$ being the element's area in 2D, or volume in 3D). In principle, this integral could be computed by splitting the element into triangular (or tetrahedral) subelements; however, following Eq. (15) in [Wriggers et al. 2017] we can also use the divergence theorem yielding the boundary integral form:

$$\mathbf{F} = \frac{1}{|\mathcal{E}|} \int_{\partial\mathcal{E}} \mathbf{X} \mathbf{n}^T d\tilde{\mathbf{x}}, \quad (1)$$

where $\partial\mathcal{E}$ denotes the element's boundary and $\mathbf{n}$ is the outward unit surface normal at $\mathbf{x}$. Note that $\mathbf{X} \mathbf{n}^T$ is the outer product resulting in a $d \times d$ matrix (where d is 2 or 3, the dimension of the space). In this way, we avoid the need to interpolate world-space positions based on the element's degrees of freedom in the interior of the element; we only need such interpolation along the edges of the element.

For linear elements in 2D, the deformed coordinates along each edge can simply be interpolated by $\mathbf{X} = \xi_i \mathbf{X}_i + (1 - \xi_i) \mathbf{X}_{i+1}$, where $\mathbf{X}_i$ are the world-space vertex positions, i and $i + 1$ refer to the indices of the first and second vertex of each edge respectively, and $0 \leq \xi_i \leq 1$. Consequently, in 2D we can compute the integral in Eq. (1) by summing over all (oriented) edges and applying midpoint quadrature. Because the normal is constant per edge and the displacement is interpolated linearly over the edge, the midpoint rule produces the exact solution:

$$\mathbf{F}^{2D} = \frac{1}{2|\mathcal{E}|} \sum_i \ell_i (\mathbf{X}_i + \mathbf{X}_{i+1}) \mathbf{n}_i^T,$$

where i refers to the edge connecting vertices i and $i + 1$ with normal $\mathbf{n}_i$ and (rest-space) length ℓ_i . (Note that in practice, the sum must also contain a term for the "last" edge from $i_{\max}$ to 1 as the polygon must be closed.)

Algorithm 1 Deformation gradient operator in 2D

```

1  Gk = zeros(2*2, 2*n_k);
2  for i=1:n_k
3      ni = i+1; if i==n_k, ni=1; end % find next node "i+1"
4      % edge data
5      e = (co_el(i,:) - co_el(ni,:))'; % edge vector (from rest-space coordinates)
6      l = norm(e); % length
7      n = [0 1; -1 0]*e/l; % normal
8      % construct N_i
9      Ni = zeros(4,2*size(d_el,1));
10     Ni(1:2,[2*i-1 1 0] 2*ni-[1 0]) = [eye(2) eye(2)];
11     Ni = [n * Ni(1,:); n * Ni(2,:)];
12     % sum up G_k
13     Gk = Gk + l / area / 2 * Ni;
14 end

```

By rearranging terms, we can write this sum as a matrix-vector product:

$$\tilde{\mathbf{F}}_k^{2D} = \mathbf{G}_k \mathbf{u}_k, \quad \mathbf{G}_k = \frac{1}{2|\mathcal{E}_k|} \sum_i \ell_i \mathbf{N}_i,$$

where $\mathbf{u}_k$ refers to the vector formed by stacking all vertex displacements of element k , i.e. $\mathbf{u}_k = (\mathbf{X}_i)$, $i \in \mathcal{E}_k$, $\tilde{\mathbf{F}}$ refers to the deformation gradient unrolled into a vector (of size d^2 , i.e., 4 in 2D, 9 in 3D). Furthermore, $\mathbf{N}_i$ is a $d^2 \times dn_k$ matrix (with n_k the number of vertices in the k -th element), where the normal vector of the i -th edge is arranged in the appropriate entries such that the matrix-vector product $\mathbf{N}_i \mathbf{u}_k$ reproduces both $\mathbf{X}_i \mathbf{n}_i^\top$ and $\mathbf{X}_{i+1} \mathbf{n}_i^\top$ terms; see also Alg. 1 for a Matlab implementation computing $\mathbf{G}_k$ in 2D. In this form, it is also easy to see that

$$\mathbf{G}_k = d\tilde{\mathbf{F}}_k^{2D}/d\mathbf{u}_k \stackrel{\text{unrolled}}{=} d\mathbf{F}_k/d\mathbf{X}_k.$$

1.1.1 Extension to 3D. The main difference when moving from 2D to 3D is that Eq. (1) now must be integrated over the faces of a polyhedral element, rather than the edges of a polygon. We assume that each polyhedron has planar faces in the undeformed rest configuration (but they do not need to remain planar in the deformed state). Consequently, each face is a planar polygon with constant face normal vector and the deformation gradient is

$$\mathbf{F}^{3D} = \frac{1}{|\mathcal{E}|} \sum_i a_i \bar{\mathbf{X}}_i \mathbf{n}_i^\top,$$

where $|\mathcal{E}|$ is now the element's volume, a_i is the area of the i -th face with normal $\mathbf{n}_i$. As in 2D, we employ midpoint quadrature here, which exactly integrates the linear function $\bar{\mathbf{X}}_i \mathbf{n}_i^\top$ over the face, where $\bar{\mathbf{X}}_i$ is the “average” deformed position of the face, which we determine as follows. Starting from Eq. (13) in [Wriggers et al. 2017], we first compute a linear least-squares fit for the displacements at each vertex of the face. To this end, we map each (undeformed) face into the xy -plane such that its centroid is mapped to $[0; 0]$, the local origin (and the face normal points along $+z$):

$$\mathbf{x}^f = \begin{pmatrix} \mathbf{n}_e^\top \\ \mathbf{t}^\top \end{pmatrix} (\mathbf{x} - \mathbf{c}_f),$$

Algorithm 2 Deformation gradient operator in 3D

```

1  % input: elements, rest coordinates, element ID, face centroids, face areas, element volumes
2  function G = computeDefGradOp(el, co, k, fc, fa, ev)
3      fcs = el{k}; % faces of element k
4      n_fc = length(fcs); % number of faces
5      [elnd,~,fc_el_idx] = unique(horzcat(fcs{:})); % unique list of nodes in element k
6      G = zeros(3*3, 3*length(elnd)); % initialize
7
8      fc_start = 0; % track start index in fc_el_idx
9      for j = 1:n_fc % loop faces
10         fcnd = fcs{j}; % node indices of face j
11         n_nd = length(fcnd); % number of nodes in face j
12
13         % face normal, edge normal, edge tangent
14         t = co(fcnd(2,:),:)-co(fcnd(1,:),:); t = t./norm(t);
15         e = co(fcnd(1,:),:)-co(fcnd(end,:),:);
16         fn = cross(t,e); fn=fn./norm(fn);
17         n = cross(fn, t); n=n./norm(n); % should already be unit length (just in case)
18
19         % project nodes of face into local 2d coords (centroid to origin)
20         co_f2d = (co(fcnd,:)-fc{k}(j,:))*[n; t]';
21
22         % compute linear least-squares fit a + b*x + c*y --> matrix A = [1 x y]
23         A = [ ones(n_nd, 1), co_f2d(:,1), co_f2d(:,2) ];
24         da = [1 0 0]*((A'*A) \ A'); % coefficients to compute constant part (da/du)
25
26         % local index of nodes (face j) into elnd array (all nodes of element k)
27         f_elnd = fc_el_idx(fc_start+(1:n_nd));
28         fc_start = fc_start + n_nd;
29
30         G([1 4 7],3*f_elnd-2) = G([1 4 7],3*f_elnd-2) + (fn' * da) * fa{k}(j) / ev(k); % x
31         G([2 5 8],3*f_elnd-1) = G([2 5 8],3*f_elnd-1) + (fn' * da) * fa{k}(j) / ev(k); % y
32         G([3 6 9],3*f_elnd ) = G([3 6 9],3*f_elnd ) + (fn' * da) * fa{k}(j) / ev(k); % z
33     end
34 end

```

where $\mathbf{t}$ is the unit tangent vector to the face's first edge, $\mathbf{n}_e$ the in-plane normal to the first edge (sometimes also "bitangent") and $\mathbf{c}_f$ the face centroid (in rest space). We then compute fitting coefficients $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$ such that:

$$\mathbf{a} + \mathbf{b}\mathbf{x}_1^f + \mathbf{c}\mathbf{x}_2^f \approx \mathbf{X},$$

in a least-squares sense, where $\mathbf{x}_1^f$ refers to the first (local x) component of $\mathbf{x}^f$. Following common linear least-squares fitting, we compute

$$(\mathbf{a} \ \mathbf{b} \ \mathbf{c})^T = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{X}_i, \quad \text{where } \mathbf{A} = \begin{pmatrix} 1 & \mathbf{x}_1^{f_i} & \mathbf{x}_2^{f_i} \end{pmatrix},$$

where $\mathbf{X}_i$ refers to the deformed vertex coordinates of the i -th face (collected into an $n_i \times 3$ matrix, with n_i the number of nodes forming this face) and $\mathbf{x}^{f_i}$ are the stacked coordinates of this face's vertices in the local 2D space. Note that the centroid of the face in the local 2D space is at the origin and therefore $\mathbf{a} = \bar{\mathbf{X}}_i$ is the average deformed position over this face. In practice, we do not compute this fitting every time we evaluate the deformation gradient, but instead precompute the coefficients that result in the constant part $\mathbf{a}$, i.e., $\mathbf{a}'_i = d\mathbf{a}/d\mathbf{X}_i$, which is the first row of $(\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top$:

$$\mathbf{a}'_i = d\mathbf{a}/d\mathbf{X}_i = (1 \ 0 \ 0) \left((\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \right).$$

Now that we have $\bar{\mathbf{X}}_i = \mathbf{a}'_i \mathbf{X}_i$, we can write the element's deformation gradient by summing over all faces and again rearrange terms into a matrix-vector product:

$$\mathbf{F}^{3D} = \frac{1}{|\mathcal{E}|} \sum_i a_i (\mathbf{a}'_i \mathbf{X}_i) \mathbf{n}_i^\top \rightarrow \tilde{\mathbf{F}}_k^{3D} = \mathbf{G}_k \mathbf{u}_k,$$

where, as in the 2D case, $\mathbf{G}_k$ collects all coefficients to be multiplied by the deformed coordinates to obtain the (unrolled) deformation gradient. The construction of $\mathbf{G}_k$ follows a similar process as in 2D, iterating over all faces of the current element, with the additional step of computing the fitting coefficients $\mathbf{a}'_i$ per face and then summing up all data into the appropriate matrix entries; Alg. 2 demonstrates this process.

1.2 Elastic forces and stiffness

Having computed the (average) deformation gradient per element, $\mathbf{F}_k$ (or equivalently, its vector-unrolled form $\tilde{\mathbf{F}}_k$), we can now evaluate any common (hyper-)elastic constitutive model to find the deformation energy, elastic forces, and tangent stiffness, in the same way as in a standard finite element method. Specifically, we choose a strain energy density function $\Psi(\mathbf{F})$, such as the invertible NeoHookean material model by Smith et al. [2018], leading to the elastic deformation energy

$$E = \int_{\Omega} \Psi(\mathbf{F}(\mathbf{x})) \, d\mathbf{x} = \sum_k |\mathcal{E}_k| \Psi(\mathbf{F}_k).$$

Furthermore, the elastic forces are defined as the negative gradient of this energy

$$\mathbf{f} = -\nabla_{\mathbf{u}} E = -\left(\frac{dE}{d\mathbf{u}} \right)^\top = -\sum_k |\mathcal{E}_k| \left(\frac{\partial \Psi}{\partial \tilde{\mathbf{F}}_k} \frac{d\tilde{\mathbf{F}}_k}{d\mathbf{u}_k} \right)^\top = -\sum_k |\mathcal{E}_k| \mathbf{G}_k^\top \frac{\partial \Psi}{\partial \tilde{\mathbf{F}}_k}^\top. \quad (2)$$

Here, the derivative of the deformation gradient is simply $\mathbf{G}_k^\top$, while the partial derivative of the energy density is typically computed in closed form depending on the choice of Ψ . For completeness, we list the steps to compute (the derivatives of) Ψ using Matlab's symbolic math toolbox in Alg. 3. Note that we use the convention that $\partial \Psi / \partial \tilde{\mathbf{F}}_k$ is a $1 \times d^2$ row-vector, and consequently $\mathbf{G}_k = d\tilde{\mathbf{F}}_k/d\mathbf{u}_k$ is a $d^2 \times dn_k$ matrix. Similarly, the tangent stiffness matrix is the negative Jacobian of the forces with respect to the degrees of freedom:

$$\mathbf{K} = -J(\mathbf{f}) = \frac{d^2 E}{d\mathbf{u}^2} = \sum_k |\mathcal{E}_k| \mathbf{G}_k^\top \frac{\partial^2 \Psi}{\partial \tilde{\mathbf{F}}_k^2} \mathbf{G}_k, \quad (3)$$

where $\partial^2 \Psi / \partial \tilde{\mathbf{F}}_k^2$ is the symmetric $d^2 \times d^2$ Hessian of the energy density. In practice, we first compute per-element energy, force, and stiffness components $E_k^{\text{elast}} = |\mathcal{E}_k| \Psi(\mathbf{F}_k)$, $\mathbf{f}_k^{\text{elast}} = -|\mathcal{E}_k| \mathbf{G}_k^\top (\partial \Psi / \partial \tilde{\mathbf{F}}_k)^\top$, and $\mathbf{K}_k^{\text{elast}} = |\mathcal{E}_k| \mathbf{G}_k^\top (\partial^2 \Psi / \partial \tilde{\mathbf{F}}_k^2) \mathbf{G}_k$, respectively, and then assemble the global system $(\mathbf{K}, \mathbf{f})$ as in a finite element approach. Finally, note that in the case of a simplicial element (triangle in 2D, tetrahedron in 3D), this construction exactly simplifies to the common linear finite element method (because the deformation gradient in a linear simplex is already constant, therefore equal to

Algorithm 3 Symbolic calculation of NeoHookean material energy density derivatives.

```

1  % using Matlab R2024b and sym toolbox v24.2
2  d = 3;
3  F_vec = sym('F_', [d*d 1], 'real');
4  F      = reshape(F_vec, d, d);
5  syms la mu real; % Lamé parameters \lambda and \mu
6  assumeAlso(la>0); assumeAlso(mu>0);
7
8  J = det(F);
9  C = F'*F;
10 IC= trace(C);
11
12 % correction coefficients according to [Smith et al., 2018]
13 alpha = 1 + mu/la - mu/(4*la);
14 delta = 1;
15
16 % energy density [Smith et al., 2018]
17 Psi = mu/2*(IC-d) + la/2*(J-alpha)^2 - mu/2*log(IC+delta);
18 % alternative: standard NeoHookean (non-invertible)
19 % Psi = mu/2*(IC-d) - (mu+la/2)*log(J) + la/4*(J^2 - 1);
20
21 Psi = simplify(Psi);
22 dPsi_dF = simplify( gradient( Psi, F_vec ));
23 ddPsi_ddF = simplify( jacobian(dPsi_dF, F_vec ));

```

its average constructed in this formulation). In the following section, we consider the case of general polygonal or polyhedral elements.

1.3 Stabilization term

When computing the per-element elastic forces (and stiffness) for polygonal (2D) or polyhedral (3D) elements that are “larger” than the corresponding simplices (in the sense that they contain more vertices, i.e., more degrees of freedom), these elements can undergo deformations such that (even though all edges remain straight lines) the average deformation gradient (i.e., the linear deformation approximation) does not change. More formally, there exist some $\mathbf{u}_k^0 \neq 0$, which are *not* rigid transformations, such that $\mathbf{G}_k \mathbf{u}_k^0 = 0$. In order to arrive at a well-defined system, these non-linear deformation modes (which allow an element to deform without producing a corresponding elastic response) need to be regularized by introducing an appropriate penalty, which in the VEM literature is often referred to as a stabilization term (or simply stabilizer).

While a large variety of different approaches (including some higher-order deformation approximations that avoid the classical stabilization approach) exist in the VEM literature, here we focus on a simple construction that reliably stabilizes the system and is strictly invariant to rigid transformations. We follow the so-called “dof-dof” stabilization approach, specifically, the idea outlined in Eq. (38) of [Yu 2022]. The overall strategy is to construct a matrix that quantifies rigid translation, rigid (linearized) rotations, and linear strain (i.e. stretching and shearing), then splitting the nodal displacements into these components and a remainder (the non-linear deformation modes) and penalizing

Algorithm 4 Constructing the stabilizer matrix in 3D.

```

1 % co_el assumed centered and scaled vertex coordinates of element k
2 D = [1+0*co_el(:,1) co_el(:,1) co_el(:,2) co_el(:,3)];
3 D = blkdiag(D,D,D); D([1:3:end 2:3:end 3:3:end],[1:3:end 2:3:end 3:3:end]) = D;
4 G_ = zeros(12, 3*n_nd); % n_nd the number of nodes in element k
5 G_(1:3,:) = repmat(eye(3), 1, length(e_lnd)); % translation op. (repeated identity blocks)
6 G_(4:12,:) = Gk; % deformation gradient operator
7 Pk = D * ((G_*D) \ G_); % projection through monomial basis
8 I = eye(3*n_nd);
9 Sk = (I - Pk)' * (I - Pk);

```

the latter. To simplify the construction, we note that the deformation gradient already contains not only the linear deformation, but also the linearized rotations (i.e., it is the responsibility of the material model defined by Ψ to deliver rotation-invariant elastic forces). Therefore, the operator that quantifies these modes per element is G_k and we only need to augment it to also measure rigid translations, which can be done simply by $T_k = (\mathbf{I}_{d \times d} \mathbf{I}_{d \times d} \dots)$, with n_k repeated identity blocks (n_k being the number of vertices in element k). We then concatenate T_k and G_k into $\tilde{G}_k = (T_k^T G_k^T)^T$, consequently, computing $\tilde{G}_k \mathbf{u}_k$ measures translation alongside the deformation gradient in vector form.

Following again Wriggers et al. [2017], we construct the scaled monomial basis matrix D_k which contains the nodal values of the coordinates $[1, \hat{x}, \hat{y}, \hat{z}]$, where $\hat{x}, \hat{y}, \hat{z}$ are computed from the rest-space coordinates (x, y, z) for each vertex as $\hat{x} = (x - \mathbf{c}_x)/h$, where $\mathbf{c}$ is the element's centroid (and analogously for $\hat{y}$ and $\hat{z}$). In other words, we shift the vertex coordinates such that the centroid moves to the origin, and scale by h , an estimate of the element's diameter (which could be the largest distance between any node pair within the element, or alternatively $h = \sqrt[3]{|\mathcal{E}_k|}$). In our experiments, the exact choice for h has little influence on the overall quality of the solution, although it might affect the conditioning of the resulting system matrix somewhat. Finally, note that when constructing D_k , the order in which individual degrees of freedom appear in $\mathbf{u}_k$ (and correspondingly in $\tilde{G}_k$) needs to be taken into account, so that each entry is placed correctly; refer to Alg. 4 for details. In the end, we have D_k such that $\tilde{G}_k D_k$ maps the scaled monomial basis into the space of translations and linear deformations, whereas its inverse $(\tilde{G}_k D_k)^{-1}$ maps translation and deformation into the scaled monomial basis, from where they can be transformed into the standard Cartesian space by D_k . Consequently, the projection operator $P_k = D_k (\tilde{G}_k D_k)^{-1} \tilde{G}_k$ maps world-space vertex locations (per element) to the closest locations described purely by rigid translation and linear deformation (including linearized rotation) of this element. Finally, we can construct the stabilization term by introducing an energy based on the squared difference in vertex coordinates before and after this projection by first defining

$$\mathbf{S}_k = (\mathbf{I} - \mathbf{P}_k)^T (\mathbf{I} - \mathbf{P}_k), \quad (4)$$

and then the stabilization energy, force, and stiffness respectively, introducing a stabilization factor α :

$$E_k^{\text{stab}} = 0.5\alpha \mathbf{u}_k^T \mathbf{S}_k \mathbf{u}_k, \quad \mathbf{f}_k^{\text{stab}} = -\alpha \mathbf{S}_k \mathbf{u}_k, \quad \mathbf{K}_k^{\text{stab}} = \alpha \mathbf{S}_k. \quad (5)$$

While various choices for α exist in the literature, we have observed good results for $\alpha = \text{tr}(\partial^2 \Psi / \partial \tilde{\mathbf{F}}_k^2) / d^2$, i.e., the mean diagonal of the elasticity Hessian. During system assembly, we simply add the stabilizer energy, force, and stiffness to the corresponding quantities from the elastic material model.

2 Parallel iterative dynamics solver

Now that we have covered the spatial discretization and calculation of elastic and stabilization forces, we can build a dynamics simulation solving Newtonian motion with any common time discretization scheme. Here, we focus on implicit BDF2 time stepping which leads to the following equations of motion:

$$\mathbf{M} \frac{d^2 \mathbf{u}}{dt^2} = \mathbf{f}(\mathbf{u}) \quad \rightarrow \quad \frac{1}{4h^2} \mathbf{M}(9\mathbf{u}_{j+1} - \hat{\mathbf{u}}) = \mathbf{f}(\mathbf{u}_{j+1}), \quad \hat{\mathbf{u}} = 24\mathbf{u}_j - 22\mathbf{u}_{j-1} + 8\mathbf{u}_{j-2} - \mathbf{u}_{j-3}, \quad (6)$$

where $\mathbf{M}$ is the mass matrix, $\mathbf{u}$ is the stacked vector of all degrees of freedom, $\mathbf{f}$ collects all forces (elasticity, stabilization, and external forces), and subscripts denote time steps of (constant) size h . For simplicity, we use a diagonally lumped mass matrix and assume constant density within each element, evenly distributing each element’s mass to all its vertices:

$$\mathbf{M} = \text{diag}(m_i \mathbf{I}_{d \times d}), \quad m_i = \sum_{k: i \in V_k} \rho_k |E_k| / n_k,$$

where i refers to each (global) mesh node, whereas V_k refers to the set of vertices and $n_k = |V_k|$ the number of vertices of element k with mass density ρ_k .

In a standard Newton solver, we would write Eq. (6) as

$$\mathbf{r}(\mathbf{u}_{j+1}) = 0 = \frac{1}{4h^2} \mathbf{M}(9\mathbf{u}_{j+1} - \hat{\mathbf{u}}) - \mathbf{f}(\mathbf{u}_{j+1}),$$

initialize $\tilde{\mathbf{u}} = \mathbf{u}_j$, then iteratively update

$$\tilde{\mathbf{u}} \leftarrow \tilde{\mathbf{u}} - \beta(d\mathbf{r}/d\tilde{\mathbf{u}})^{-1} \mathbf{r}(\tilde{\mathbf{u}})$$

until convergence, and finally set $\mathbf{u}_{j+1} = \tilde{\mathbf{u}}$. Here β is a step size (typically 1, unless a line search method is used to ensure convergence) and the linear system matrix is

$$\frac{d\mathbf{r}}{d\tilde{\mathbf{u}}} = \frac{9}{4h^2} \mathbf{M} + \mathbf{K}^{\text{elast}}(\tilde{\mathbf{u}}) + \mathbf{K}^{\text{stab}}(\tilde{\mathbf{u}}).$$

This classical approach, however, requires the linear system to be updated in every iteration, assembled into a global (sparse) matrix and then solved (typically using an iterative method). Most of these steps are not easy to parallelize when attempting to employ modern GPU hardware, that in principle, supports a very high number of parallel operations.

More recent approaches, such as (extended) position-based dynamics, (ADMM) projective dynamics, or (augmented) vertex block descent [Bouaziz et al. 2014; Giles et al. 2025; Macklin et al. 2016; Narain et al. 2016] have introduced various highly parallel solvers that basically employ some form of parallel Gauss-Seidel updates. While these solvers typically require many more iterations than the classical Newton method, their improved parallelism combined with their stability (making partially converged solutions acceptable for graphics applications) have enabled (near) real-time simulations. Here, we formulate a similar approach building on these ideas, that (1) first updates each element independently in parallel and then (2) repositions vertices shared among elements to the average location of their corresponding element-local intermediate positions, as illustrated in Fig. 3.

2.1 Parallel iterative solver

When solving directly for the (global) vector of unknown degrees of freedom $\mathbf{u}$, this parametrization—akin to reduced coordinates in articulated dynamics—already “hardcodes” the constraint that all elements sharing a common node must have that node in the same (world-space) position, or in other words, each node in the mesh has a well-defined location. In order to enable parallel computation, we move to a “maximal coordinates” point of view, where each element k maintains a local vector of node positions $\mathbf{u}_k$ for the corners of this element. In the first step of the solver, we first

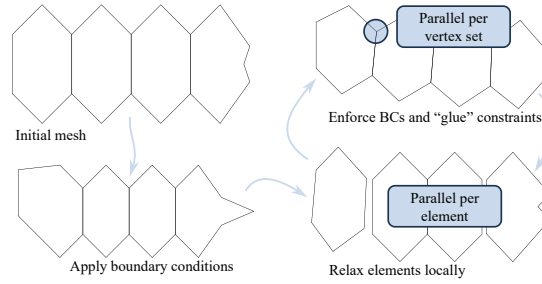


Fig. 3. Overview of our parallel iterative solver approach: Starting from the rest-space mesh (top left), we first apply Dirichlet boundary conditions on the mesh nodes (bottom left). We then iteratively update the vertices of each element independently, relaxing the local elastic forces (bottom right), and finally enforce both boundary conditions and vertex connectivity (“glue”) constraints. The first stage can be done in parallel for all elements, the second stage in parallel for all vertex sets. (Each set contains copies of the same global vertex after the per-element local update.)

initialize the local unknowns $\tilde{\mathbf{u}}_k$ from a ballistic prediction at the beginning of each time step, using the BDF2 velocity discretization:

$$\tilde{\mathbf{u}}_k = (\mathbf{u}_j + \gamma h \mathbf{v}_j)_k, \quad \mathbf{v}_j = (3\mathbf{u}_j - 4\mathbf{u}_{j-1} + \mathbf{u}_{j-2})/(2h),$$

with $\gamma \in [0, 1]$ and then update the local unknowns for each element in parallel by

$$\tilde{\mathbf{u}}_k \leftarrow \tilde{\mathbf{u}}_k - \beta \operatorname{diag} \left(\frac{9}{4h^2} \mathbf{M}_k + \mathbf{K}_k(\tilde{\mathbf{u}}_k) \right)^{-1} \left(\mathbf{M}_k \frac{1}{4h^2} (9\tilde{\mathbf{u}}_k - \hat{\mathbf{u}}_k) - \mathbf{f}_k \right), \quad (7)$$

where $\mathbf{K}_k$ and $\mathbf{f}_k$ combine the (per-element) elastic and stabilizer force and stiffness respectively. Note that, following the idea of a Gauss-Seidel iteration, we only invert the diagonal of the (local) system matrix, rather than the entire matrix. While VBD actually inverts the local system matrix in each update (which in their case is a 3×3 matrix, as they only update one node at a time), we observe that this approach is highly unstable when updating all nodes of the element at once (even when adding a regularization term to the matrix). Conversely, the Gauss-Seidel approach is quite stable when choosing a step size $\beta < 1$ (typically around 0.5–0.9).

After each parallel per-element update, the nodes shared among adjacent elements end up in different locations, therefore we must enforce mesh connectivity as an explicit constraint. We can project the node positions onto this constraint by moving all local nodes that represent copies of the same global node to their average position. To formalize this process, we consider sets of local indices $V_i = \{(k, l)\}$ such that the l -th segment (size d) of the k -th element’s local vector refers to the global node i , i.e., $\mathbf{X}_i = (\mathbf{u}_k)_l$. We then compute the average position for the global node

$$\bar{\mathbf{X}}_i = \frac{1}{|V_i|} \sum_{(k,l) \in V_i} (\tilde{\mathbf{u}}_k)_l.$$

Additionally, we enforce Dirichlet boundary or contact conditions on $\bar{\mathbf{X}}_i$ and then move all local copies to that location

$$(\tilde{\mathbf{u}}_k)_l = \bar{\mathbf{X}}_i \quad \forall (k, l) \in V_i. \quad (8)$$

We then iterate between Eq. (7) and (8) until either a maximal number of iterations has been reached, or the difference $\|\tilde{\mathbf{u}} - \mathbf{u}_j\|$ is below a given tolerance. Finally, the end-of-time-step state is set to $\mathbf{u}_{j+1} = \tilde{\mathbf{u}}$, the stacked vector of all constraint-projected node positions; Fig. 3 illustrates this process.

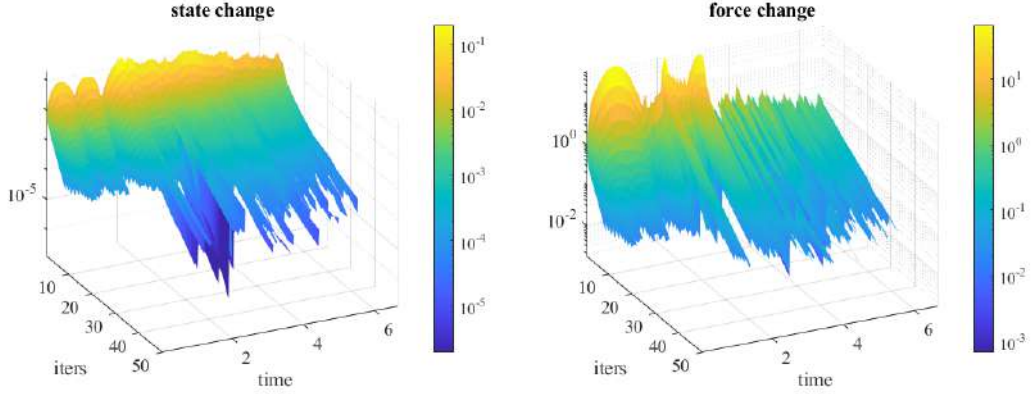


Fig. 4. Convergence plots showing absolute change (L_2 norm) of positions and (constraint) forces for the example in Fig. 1.

2.2 ALM forces and friction

So far, the solver we have formulated in the previous section follows a strictly “position-based” approach to enforcing constraints: nodes are simply moved to new locations such that they satisfy constraints, which could be Dirichlet boundaries, non-penetration contact constraints, or mesh connectivity (“glue”) constraints. However, there are two downsides to this approach: first, convergence is sometimes limited (either too slow, or nodes oscillate between iterations but do not converge properly), and second, we do not compute forces applied to the system by the constraints. While the former may be acceptable, as the simulation still remains stable, when running with a limited number of iterations, the latter is particularly important for Coulomb friction, where the tangential force limit depends on the normal contact force. Computing accurate forces also enables extensions like fracture simulation for future work.

Including Lagrange multipliers (forces due to constraints) $\mathbf{f}_k^D$, $\mathbf{f}_k^g$, $\mathbf{f}_k^c$ for Dirichlet boundary conditions, glue constraints, and contact (normal force) respectively, as well as a friction (tangential) force $\mathbf{f}_k^f$ to each element k , the total force in Eq. (7) is then:

$$\mathbf{f}_k = \mathbf{f}_k^{\text{elast}} + \mathbf{f}_k^{\text{stab}} + \mathbf{f}_k^D + \mathbf{f}_k^g + \mathbf{f}_k^c + \mathbf{f}_k^f. \quad (9)$$

We update the constraint forces in each iteration using an augmented Lagrange multiplier (ALM) approach. Let’s first consider Dirichlet boundaries and mesh connectivity: just before we update a local node’s position according to Eq. (8), we compute a corresponding penalty force and increase the corresponding Lagrange multiplier:

$$(\mathbf{f}_k^D)_l \leftarrow (\mathbf{f}_k^D)_l + ((\tilde{\mathbf{u}}_k)_l - \mathbf{X}_l^D) k_D, \quad (\mathbf{f}_k^g)_l \leftarrow (\mathbf{f}_k^g)_l + ((\tilde{\mathbf{u}}_k)_l - \tilde{\mathbf{X}}_l) k_g,$$

where k_D and k_g are penalty factors (typically of similar magnitude as the material’s elastic modulus) for Dirichlet conditions and glue constraints respectively. Naturally, Dirichlet boundaries take precedence here, i.e., if a global node is constrained to a specific location $\mathbf{X}_l^D$ all corresponding local nodes will be moved exactly to this location, and the connectivity constraint for these nodes is satisfied by definition.

Next, we consider contacts with a known static obstacle, such as the planar floor in the example of Fig. 1. We assume that the obstacle is given as a (continuously differentiable) signed-distance field $g_c(\mathbf{X})$ with outward normal vector $\mathbf{n}_c = dg_c/d\mathbf{X}$, and we enforce the contact constraint

$$g_c(\mathbf{X}) \geq 0.$$

In the case of a planar floor, it is sufficient to enforce this constraint directly on the world-space positions of the mesh nodes $\mathbf{X}_i$. For more complex scenarios, like general obstacle shapes, or self-collisions of the deformable object etc., a more elaborate collision detection approach would be needed. In principle, any penalty-based collision detection and resolution framework could be employed. Returning to our simplified contact model, similarly to Dirichlet boundaries, we update the contact (normal) force as an augmented Lagrange multiplier whenever $g_c(\tilde{\mathbf{X}}_i) < 0$ with penalty factor k_c :

$$(\mathbf{f}_k^c)_I \leftarrow (\mathbf{f}_k^c)_I - \mathbf{n}_c g_c(\tilde{\mathbf{X}}_i) k_c,$$

and then project the node position to the contact constraint by $\tilde{\mathbf{X}}_i \leftarrow \tilde{\mathbf{X}}_i - g_c(\tilde{\mathbf{X}}_i) \mathbf{n}_c$ just before the node update, Eq. (8).

Finally, now that we have a good estimate of the normal force due to contact for each (local) node, $(\mathbf{f}_k^c)_I$, we compute a friction force following the well-known Coulomb model. We first compute a trial force opposing the tangential velocity $(\tilde{\mathbf{f}}_k^f)_I$ and then clamp this force to the Coulomb limit to obtain the final friction force as

$$(\tilde{\mathbf{f}}_k^f)_I = -(\mathbf{I} - \mathbf{n}_c \mathbf{n}_c^T)(\tilde{\mathbf{v}}_k)_I h k_c, \quad (\mathbf{f}_k^f)_I = \max(\|(\tilde{\mathbf{f}}_k^f)_I\|, \mu_f \|(\mathbf{f}_k^c)_I\|) (\tilde{\mathbf{f}}_k^f)_I / \|(\tilde{\mathbf{f}}_k^f)_I\|, \quad (10)$$

where μ_k is the coefficient of friction and $(\tilde{\mathbf{v}}_k)_I$ is the BDF2 velocity due to $(\tilde{\mathbf{u}}_k)_I$.

Overall, we compute forces due to constraints as augmented Lagrange multipliers, add Coulomb friction, and attain convergence in terms of both primary (position) and dual (ALM) variables, as shown in Fig. 4.

2.3 Future work

[ToDo]

- Reformulate our approach as in ADMM (looks fairly similar, might be interesting to work out the theory).
- Warm start ALM forces at start of time step?
Could be dangerous on non-smooth contact and friction forces, but might work on Dirichlet BCs and glue constraints.
- Collisions other than flat floor in practice.
- Fractures (separation tensor approach of O'Brien based on glue forces should be doable).
- Viscous damping (should be fairly straightforward based on the ADD paper).

References

- Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective dynamics: fusing constraint projections for fast simulation. *ACM Transactions on Graphics* 33, 4 (July 2014), 1–11. doi:10.1145/2601097.2601116
- Chris Giles, Elie Diaz, and Cem Yuksel. 2025. Augmented Vertex Block Descent. *ACM Transactions on Graphics* 44, 4 (July 2025), 1–12. doi:10.1145/3731195
- Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games (MiG '16)*. ACM, 49–54. doi:10.1145/2994258.2994272
- Rahul Narain, Matthew Overby, and George E. Brown. 2016. ADMM $\supseteq$ projective dynamics: fast simulation of general constitutive models. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (Zurich, Switzerland) (SCA '16)*. Eurographics Association, Goslar, DEU, 21–28.
- Breannan Smith, Fernando De Goes, and Theodore Kim. 2018. Stable Neo-Hookean Flesh Simulation. *ACM Transactions on Graphics* 37, 2 (March 2018), 1–15. doi:10.1145/3180491
- P. Wriggers, B. D. Reddy, W. Rust, and B. Hudobivnik. 2017. Efficient virtual element formulations for compressible and incompressible finite deformations. *Computational Mechanics* 60, 2 (April 2017), 253–268. doi:10.1007/s00466-017-1405-4
- Bing-Bing Xu, Fan Peng, and Peter Wriggers. 2024. Stabilization-free virtual element method for 3D hyperelastic problems. *Computational Mechanics* 75, 6 (May 2024), 1687–1701. doi:10.1007/s00466-024-02501-4
- Yue Yu. 2022. mVEM: A MATLAB Software Package for the Virtual Element Methods. (April 2022). arXiv:2204.01339 [math.NA] doi:10.48550/ARXIV.2204.01339