THE UNIVERSITY
of EDINBURGH

Accelerating and Simulating Detected Physical Interactions

Floyd Mulenga Chitalu

Doctor of Philosophy

Institute of Perception, Action and Behaviour

CDT Pervasive Parallelism

School of Informatics

The University of Edinburgh

2020

Abstract

The aim of this doctoral thesis is to present a body of work aimed at improving performance and developing new methods for animating physical interactions using simulation in virtual environments. To this end we develop a number of novel parallel collision detection and fracture simulation algorithms.

Methods for traversing and constructing bounding volume hierarchies (BVH) on graphics processing units (GPU) have had a wide success. In particular, they have been adopted widely in simulators, libraries and benchmarks as they allow applications to reach new heights in terms of performance. Even with such a development however, a thorough adoption of techniques has not occurred in commercial and practical applications. Due to this, parallel collision detection on GPUs remains a relatively niche problem and a wide number of applications could benefit from a significant boost in proclaimed performance gains.

In fracture simulations, explicit surface tracking methods have a good track record of success. In particular they have been adopted thoroughly in 3D modelling and animation software like Houdini [124] as they allow accurate simulation of intricate fracture patterns with complex interactions, which are generated using physical laws. Even so, existing methods can pose restrictions on the geometries of simulated objects. Further, they often have tight dependencies on implicit surfaces (e.g. level sets) for representing cracks and performing cutting to produce rigid-body fragments. Due to these restrictions, catering to various geometries can be a challenge and the memory cost of using implicit surfaces can be detrimental and without guarantee on the preservation of sharp features.

We present our work in four main chapters. We first tackle the problem in the accelerating collision detection on the GPU via BVH traversal - one of the most demanding components during collision detection. Secondly, we show the construction of a new representation of the BVH called the *ostensibly implicit tree* - a layout of nodes in memory which is encoded using the bitwise representation of the number of enclosed objects in the tree (e.g. polygons). Thirdly, we shift paradigm to the task of simulating breaking objects *after* collision: we show how traditional finite elements can be extended as a way to prevent frequent re-meshing during fracture evolution problems. Finally, we show how the fracture surface–represented as an explicit (e.g. triangulated) surface mesh–is used to generate rigid body fragments using a novel approach to mesh cutting.

Lay summary

This thesis presents a body of work relating to simulations in virtual environments for various applications like motion picture generation. It will describe methods for improving the speed of collision detection using graphics processing units (GPU), and simulating brittle fracture effects in interacting physical objects.

In collision detection we often use all sorts of representations/approximations of objects in order to speed up computations at runtime. One such representation is the bounding volume hierarchy (BVH), which is a widely used tree data structure for collision detection. The BVH is a coarse and multi-resolution representation of an object's surface and serves to reduce the time to find the set of surface triangles that are in close proximity with another object. Thus, the BVH is very useful for accelerating collision queries.

Several ways exist to *build* and *traverse* the BVH in order to quickly find those intersecting triangles. Parallelism via GPUs is one approach which has been adopted in simulators like Bullet [24], and other library tool-kits for applications like cloth simulation. However, existing techniques can be complex and with ad-hoc schemes for harnessing the parallel performance of GPUs. Thus, collision detection on GPUs remains an open problem as further considerations on implementation and data structures are necessary to continue improving the performance of building and traversing the BVH.

For the fracture problem, we are interested in simulating how interacting rigid body objects will break into small pieces - e.g. after a collision. Several numerical methods exist for simulating these realistic and interesting fracture effects in 3D, which are based on continuum mechanics. Notably, those methods which avoid practical issues like frequent re-meshing of the domain when propagating a crack have grown in popularity. These so-called *re-meshing free* and *boundary element* techniques recast the simulation problem as one of tracking crack fronts in the domain, which is in contrast to the re-meshing route of traditional finite elements (i.e. 'FEM').

Though impressive, re-meshing free and boundary element techniques may also prove limited on objects with large surface-area-to-volume ratios, and they are dependent on implicit level set functions for crack representations to affect the resolution of volumetric discretization. Boundary elements may also lack the benefits of a volumetric discretisation like the ability to naturally specify spatially varying material properties to influence crack behaviour. Thus despite their popularity, the range of

objects for simulation can be constrained and, in some cases, retaining realistic–and ridge-like–sharp features of cracks can not be guaranteed easily.

We present our work in four main chapters. We first tackle the problem of accelerating collision detection on the GPU via BVH traversal - one of the most demanding components during collision detection. Secondly, we show the construction of a new representation of the BVH called the ostensibly implicit tree - a layout of data in memory which is encoded using the bitwise representation of the number of enclosed triangles in the tree. Thirdly, we shift paradigm to the task of simulating breaking objects *after* they have experienced a collision. This task focuses on simulating the continuum mechanics which lead to fracture and material separation while overcoming several limitations of prior methods. Building on this task we also explore a new way to cut objects with the simulated crack surfaces so that we can generate new rigid body fragments that exhibit interesting fracture patterns.

Acknowledgements

I would like to thank my supervisor Taku Komura for his help, support and guidance throughout my Ph.D. I am also grateful to Christophe Dubach, my second supervisor, for the interest in my work, the ideas, and guidance which he gave me throughout. Finally I would like to thank all of the other academics, reviewers, and researchers which I have met during my studies and who have selflessly shared their ideas, helped me learn new concepts, and given their time extremely generously.

My work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Chapter 2 describes materials which were published in the proceedings of I3D'18

- Floyd M. Chitalu, Christophe Dubach, and Taku Komura. 2018. *Bulk-synchronous parallel simultaneous BVH traversal for collision detection on GPUs*. In Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '18)

Chapter 3 describes materials which were published in the special issue of Computer Graphics Forum (CGF) journal containing the proceedings of The 40^{th} Annual Conference of the European Association for Computer Graphics (EuroGraphics'20).

- Floyd M. Chitalu, Christophe Dubach, and Taku Komura. *Binary Ostensibly-Implicit Trees for Fast Collision Detection*. Computer Graphics Forum. 2020.

Chapter 6 and Chapter 7 describe materials which were also published in the special issue of Computer Graphics Forum (CGF) journal (EuroGraphics'20).

- Floyd M. Chitalu*, Qinghai Miao*, Kartic Subr and Taku Komura. *Displacement-Correlated XFEM for Simulating Brittle Fracture*. Computer Graphics Forum. 2020.

To my family.

Twafuma ukutali.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Physical interactions of simulated dynamic systems in virtual environments are an essential component of many applications such as motion picture generation. Virtual physical interactions are common in existing and emerging applications in computer games, animation software for digital production and real-life simulators. The set of potential applications also sheds light on the value of performance, simplicity and robustness of their algorithms.

These algorithms find use within a wide ranging set of examples - from particle systems, to complexly deforming objects like cloth, and even rigid body interactions with fracture effects. Categorically, they may be organised–depending on the problem–according to whether they involve detecting collision contacts, resolving them, or (possibly) even simulating the dynamic behaviour of the interacting bodies in their continuum state - leading to e.g. fracture.

Yet, despite the plethora of use-cases and existing methods across various applications, the computational challenges of simulating physical interactions remain paramount. Two key challenges which are addressed in this thesis reside in addressing 1) the performance bottleneck of detecting collisions, and 2) the implementation challenge arising from resolving the dynamic changes in physical material due to high-impact collisions which lead to fracture. In particular, previous methods have demonstrated how collision detection performance can be further improved when parallelism is utilised via graphics processing units (GPU). For simulating high-impact collisions with fracture, recent methods have demonstrated how fractures can be simulated explicitly with surface meshes but without re-meshing problems associated with classic finite elements or mass spring systems.

1.1 Research questions and challenges

We investigate two thematic research questions:

What are the limitations of parallelising *collision detection*, and can new data structures and algorithms help to resolve these limitations?

Algorithms using the bounding volume hierarchy (BVH) and based on front-tracking [154] are common-place in literature for their ability to speed up parallel broad-phase collision detection. The approach has proved to work well but places stringent constraints on algorithm design to limit performance: a collision front must be managed explicitly, which is a source of overhead; there are additional storage costs; and, it leads to collision detection pipelines which do not permit full re-builds of the BVH(s) in question. In effect, refitting bounding volume extents is the only way to update the BVH at runtime, which can lead to severe quality degradation and thus, negatively affect performance. We wish to find an alternative approach wherein the dependance on front tracking for performance is alleviated and where the cost of BVH construction (time) is reduced to a negligible amount allowing for frequent rebuilds. Thus, can new–and perhaps, simpler–algorithms for BVH traversal and construction assist in this direction, and can new representations (in terms of storage and layout) for the BVH offer a solution to addressing these challenges?

Can we use finite elements to animate *breaking objects* after collision while mitigating inherent drawbacks?

Fracture simulations in computer graphics applications have been employed on a range of *simulation domains*. On tetrahedral meshes, the finite element method (FEM) has produced realistic results [107]. However, it is necessary to conform the domain (e.g. tetrahedral mesh) to the crack surfaces by re-meshing, which poses several challenges when treating evolving fractures. Thus, can we simulate fracture without requiring re-meshing *during crack propagation*, and while doing so, is it possible to minimise changes on the resulting systems of equations to be solved?

We are also motivated by a particularly interesting problem arising within fracture simulation: cutting objects (meshes) for generating fragments, which is essential and occurs frequently. Existing solutions require intermediate volumetric decompositions or level sets which can lead to ad-

ditional stability issues or high demands on storage without certainty on the preservation of geometry detail. Thus, can the cutting problem be solved in a general and suitably robust manner without depending on volumetric decompositions (or even explicit triangulations of meshes)?

1.2 Contributions

This thesis makes contributions in two strands relating to interacting physical objects in simulated environments: 1) BVH construction and traversal for parallel broad-phase collision detection using implicit-trees, and 2) brittle fracture simulation without re-meshing during crack propagation, and a cutting algorithm for computing mesh fragments.

Parallel Broad-phase Collision Detection: Part I investigates the application of implicit binary trees in broad-phase collision detection, by outlining two methods (BVH traversal and construction) and demonstrating the improvements in performance. Additionally, we show how the newly introduced data structure can be extended beyond binary trees.

The contributions of Part I are as follows:

1. A new simultaneous and parallel algorithm for traversing of multiple BVHs for pair-wise collision detection on the GPU. Under the assumption of an implicit tree we show how traversal can be simplified and improved, in terms or performance, using bulk-synchronous parallel processing (Chapter 2).

2. The ostensibly-implicit tree: An encoding of the implicit binary tree which is indexed like a heap but without memory padding constraints to account for missing data elements. Our design is based on the observation that for a given number of BVH nodes, an almost perfect (i.e., fully populated) binary tree can be completely determined, and the nodes missing for it to be "perfect" can be characterised through simple bit-manipulations. The missing (virtual) nodes can be stored in a bit-field, and this bit-field can be used to calculate the memory location of any node on the fly - such that we only require a fixed order of nodes, which is given by the Morton code (Chapter 3).

3. A novel indexing scheme for implicit tree structures with which any node can be determined as the descendant (or ancestor) of another by using only the labelling

of the nodes of the tree (Chapter 4).

Brittle fracture simulation with extended finite elements: In Part II we describe a new method for simulating brittle fracture using enriched finite elements and show complex results with rigid body animations and breakage into arbitrary shapes and sizes.

1. At the heart of our method is a technique known as the extended finite element method (XFEM) [27, 30, 96] which is commonly used in engineering applications to simulate fracture, particularly in 2D [120]. XFEM handles cuts by adding special shape functions to the approximation space of regular finite elements. For example, the method copes with singularities in the stress-field, around the crack-tip, by increasing the degrees of freedom of the system of equations to be solved. This *crack-tip enrichment* can improve accuracy, but is difficult to specify in 3D for general objects, increases computational complexity and potentially introduces instability. In computer graphics, XFEM has been used for simulating the dynamics of deformable objects with predefined cuts [64, 78] and for cutting and tearing of thin shells [70]. We present an adaptation of XFEM for brittle fracture in arbitrary objects without the need for crack-tip enrichment (Chapter 6).

2. We devise a method that performs cutting of surface meshes using the simulated crack mesh. The method is a general and robust surface-cutting algorithm that copes with concavities, without needing triangulation (Chapter 7).

1.3 Thesis Structure

The remainder of the thesis proceeds as follows.

Part I describes parallel collision detection methods using the BVH and newly proposed algorithms for traversal and construction. First, Chapter 2 describes our bulk-synchronous parallel algorithm for accelerating GPU based simultaneous BVH traversal using implicit binary trees. Chapter 3 then describes the *ostensibly-implicit* tree, which is a new way to store BVH data in memory but without the classical limitations of implicit trees - e.g. memory padding. In Chapter 3 we also describe an updated collision detection pipeline wherein the BVH(s) can be constructed at every simulation time step - thanks also to a new construction algorithm which we describe. A detailed

derivation of our implicit indexing scheme which is used in Chapter 2 and Chapter 3, is provided in Chapter 4.

In Part II we describe an adaptation of XFEM for computer graphics applications that can simulate crack-propagation within arbitrary volumes. A broad perspective overview of this part of the thesis is first given Chapter 5. Chapter 6 then presents a novel quasi-static brittle fracture simulation method on the basis of XFEM but without crack-tip enrichment. In Chapter 7, we describe a new mesh cutting algorithm (including source code) which is used frequently for computing fragments during simulation. We also present a number of examples demonstrating the effectiveness of the methods on a wide set of mesh geometries.

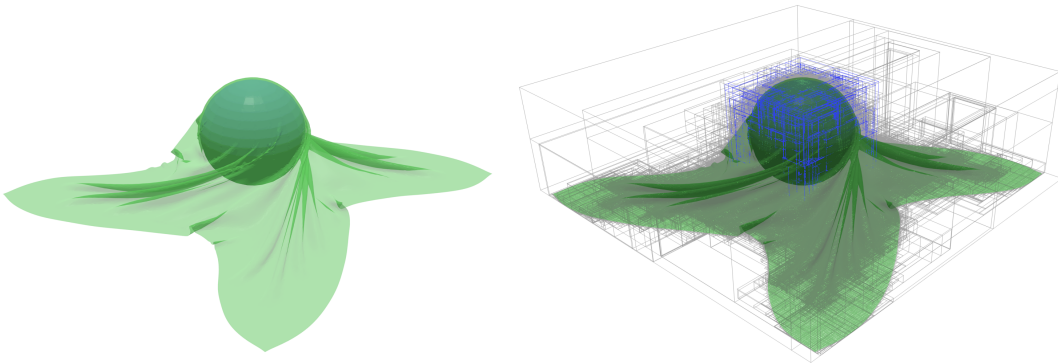
Part III concludes this thesis. Chapter 8 recapitulates the stated contributions and reflects on the directions for future work.

Part I

Accelerating Detected Physical Interactions

Chapter 2

Bulk-Synchronous Parallelism for Simultaneous BVH Traversal



(a) Cloth and Sphere Simulation (b) BVH Visualisation

Figure 2.1: *Example depiction of a simulated scene with BVH-based broad-phase collision detection between a cloth (non-rigid body) and a sphere (rigid body).*

2.1 Preface

The bounding volume hierarchy (BVH) (*cf.* Fig. 2.1(b)) is a common tree data structure for amortising the cost of finding intersections between objects [35]. Numerous techniques have been proposed to accelerate this *collision detection* using the BVH and GPUs to parallelise tree operations.

However, there are many challenges associated with parallelising tree operations on GPUs. Tree traversal is one such case wherein large amounts of asynchronous opera-

tions by individual processing threads occur. Traversing a BVH–as a task of pair-wise intersection tests–is a highly dynamic and data dependent problem facing susceptibility to control-flow divergence (branching) and inefficient data-access patterns - both of which are bad for GPU architectures.

In this chapter we will present a method to address these issues. We will describe an approach for simultaneously traversing a large number of BVHs using the bulk-synchronous parallel model [150] to ensure a uniform mode of execution, and balanced workloads across GPU threads. The method is a simple solution which is easy to implement, fast and operates entirely on the GPU by relying on a work expansion scheme that is utilising BVH structure to ensure large concurrent workloads. Experiments reveal that this method can achieve speedups of up to $7.1\times$ over the widely used "collision-streams" algorithm [138] in GPU based parallel collision detection. Thus, the method is beneficial for various applications that are involve vast intersection tests.

A brief overview of GPU architecture is provided in Appendix A.

2.2 Introduction

Collision detection has a wide-spectrum of practical applications including physics based animations, robotic motion planning, virtual disassembly, haptic rendering and ray-tracing. It is a well known and long studied problem of finding a–typically large–number of interactions at low computational cost. As a result, collision detection is at the core of many applications in computer science and engineering.

For such applications, an important case is *continuous collision detection* (CCD) wherein contacts between deformable objects at two discrete time steps are found using costly (but accurate) interpolations of smooth motion. In interactive programs, this computation must be performed at rates of 30–60Hz or higher - thus placing stringent requirements on algorithm design.

To address such requirements, the bounding volume hierarchy (BVH) is a common object-partitioning data structure [35]. The BVH offers coarse multi-resolution approximations of mesh geometry as shown in Fig. 2.1(b). In return, it inherently favours dynamic (deforming) geometry and it is relatively fast to build - while still providing sufficient approximations that require potentially less memory than e.g. uniform grids [93].

In spite of this potential benefit, the enclosed geometry may reach scales of tens- to hundreds-of-thousands of triangles or more which makes the prospect of just em-

ploying the BVH alone insufficient. Furthermore, BVHs can also degenerate if the enclosed geometry is relatively small, such that traversing entire BVHs becomes a redundant overhead.

Previous methods tackling the problem of optimising BVH based collision detection on the GPU [32, 84, 138, 141] offer in-part successful but also complex solutions which can suffer from GPU under-utilisation. They emulate the logic of conventional single-threaded CPU traversal by relying on thread-level private work-stacks and temporal coherence [87]. Temporal coherence is a concept describing the similarity between geometry states at consecutive simulation timesteps, which are used to jump-start traversal. The method is useful for reducing redundant intersection tests between bounding volumes.

However, such heuristics can serve to complicate a traversal algorithm and may thus constrain GPU performance. Work-stacks serve to reduce unfettered shared memory accesses and synchronisation costs, but their use can be a source of control-flow divergence, as well as load-imbalance which must be managed by a separate GPU kernel in the pipeline (see also discussions by Aila and Karras [1]). Moreover, previous methods have used work-stacks to effectively mimic recursion on the GPU because threads perform traversal *in-place*: Evaluation of pair-wise tests in BVH sub-trees is computed independently as threads push and pop intermediate BVH node-pairs to-and-from work-stack memory which creates the divergence in control-flow as a side-effect. Temporal coherence on the other hand, has a high memory footprint since it is based on explicitly storing the BVH node-pairs where traversal terminates. It is also a potential source of work-flow divergence because simply checking when and how to store such node-pairs contributes to the overhead of branching on GPUs.

So, in this chapter, we will present a simpler and faster method for simultaneously traversing a large number of BVHs for collision detection - in parallel and entirely on the GPU. The method is based on the bulk-synchronous parallel (BSP) model [150] were BVH traversal is reformulated as an iterative "fork and join" process to: (1) mitigate explicit load-balancing that requires using separate work-rebalancing kernels on the GPU, (2) minimise control-flow divergence by reducing the amount of work mapped to each thread and performing full-restarts from a user-specified entry level such as the root-level, and (3) allow for efficient memory access patterns that may be coalesced while seamlessly unifying synchronisation, communication and storage. Experiments performed on three UNC dynamic scene benchmarks (see Fig. 2.8) also reveal up to 7.1$\times$ speedup over the 'collision-streams' model [138], which is widely

adopted (see e.g. [31, 32, 140, 141]).

2.2.1 Chapter Contributions

The contributions form a simple solution, from using the topological structure of BVHs and simplified thread-level operations for reducing control-flow divergence, to efficiently traverse multiple BVHs in parallel on the GPU:

- A novel algorithm (Section 2.6) as alternative reformulation of simultaneous and parallel traversal of multiple BVHs for pair-wise collision detection on the GPU.
- Parametric workload expansion (Section 2.6.2): *Adaptive depth-stepping* and *static workload expansion* are introduced as key features for ensuring large concurrent workloads and controlling the rate of traversal, using the topological structure inherent in the traversed BVHs.
- A lightweight atomic synchronisation scheme to write intermediate BVH node-pairs to global memory using iterative buffered-writes (Section 2.6.3), which can be controlled based on the topological properties of BVHs and available hardware resources.

Organisation: The rest of the chapter proceeds as follows: After reviewing related work in Section 2.3, we first give an overview of our method in Section 2.4. We then describe how we store and retrieve BVH data at runtime in Section 2.5, and describe the core steps in our algorithm in Section 2.6. We present our experimental results in Section 2.7 and conclude in Section 2.8.

2.3 Related Work

In this section, we first review collision detection methods which are commonly used in physics-based animations. Next, we review related methods which tackle parallel collision detection on GPUs. We then briefly describe methods based on stack-less BVH traversal. Finally, we review related computational models for graph processing, which share some properties with the parallelisation strategy presented of this chapter.

2.3.1 Collision Detection in Physics-based Animation

Collision detection lends itself well to physics-based simulation problems for real-time and off-line use-cases [35]. It is particularly useful in relatively large scale problems involving complex rigid, and non-rigid objects such as cloth [13]. In non-rigid simulations, the complexity of interactions (which may include self-collisions) places emphasis on the need for efficient culling of triangle-triangle intersection tests which have a high computational cost. BVHs are a common data structure in many such works with their ability to quickly cull of the search space of potential interactions [147]. Numerous approaches including axis-aligned bounding boxes (AABB) [11], oriented bounding boxes (OBB) [46], discrete oriented polytopes (k-DOP) [75] have been introduced for this purpose, which function as approximations to the underlying geometric primitives that they enclose in the form of coarse bounding volumes. To obtain sufficient visual fidelity in simulation results it is crucial to detect collisions accurately, since a single missed collision may result in an invalid simulation or noticeable artefacts [14].

2.3.2 Parallel Collision Detection

Methods to accelerate collision tests through parallelism on GPUs are widely investigated. Early pioneering works such as that of Knott and Pai [76] made use of the parallel rasterisation capabilities of GPUs. The more recent methods, including Tang *et al*. [141] and others [138, 140, 159, 163], utilise the general purpose computational capabilities of modern GPUs to accelerate computation following the advent of parallel programming frameworks such as CUDA and OpenCL. Wong *et al*. [163] present a parallel adaptive scheme combining octrees and hierarchical grid structures for broad-phase collision detection with deformable objects. Weller *et al*. [159] recently introduce a CUDA based scheme, kDet, which uses hierarchical grid structures to find the set of potentially colliding pairs using polygon sizes.

For BVH-based schemes, mapping the traversal operation to GPUs is a challenging task as demonstrated by prior efforts which advocate for the use of more parallelism through many-core GPUs and multi-core CPUs [84, 139, 141]. As naïve approaches can easily result in hardware under-utilisation due to insufficient workloads, the most influential methods such as Lauterbach *et al*. [84] and other variants [32, 138, 140, 141] have relied on *front tracking* [139] (i.e. temporal coherence) for sustaining high workloads which is an ideal approach for GPUs to generate large workloads. In this ap-

proach, the bounding volume test tree (BVTT) [47] of BVH node-pairs where traversal terminates is explicitly cached and then reused as input for next time. In addition, optimisation strategies inspired by the model architecture of Aila and Karras [1] are common (though not explicitly stated). For example, thread-level private work-stacks are another common feature in these methods to improve memory access costs and minimise inter-thread synchronisation [1, 3]. However, work-stacks can lead to work-flow divergence and load-imbalances that require a separate GPU task to perform work redistribution between threads (see Lauterbach *et al*. [85] and [84] for details). Aila and Laine [3] also propose the alternative case for using persistent threads (in context of ray tracing) which has been shown to improve performance instead of the hardware work distribution mechanisms. Similar approaches have also been used in robotic motion planning [109, 111]. The related work of Hermann *et al*. [56] performs collision detection for motion planning using voxel maps maintained in GPU global memory.

The method described in this chapter shares some similarities with the aforementioned parallel collision detection methods but does not rely on work-stacks nor front-tracking. It adopts the BVTT as the primary input but distinctively expresses traversal as an iterative one-to-one mapping between threads and evaluated BVH node-pairs. Further, the focus of our method is the specific problem of handling pair-wise intersection tests between BVH nodes, whereas related methods are focused on the entire collision detection pipeline. Another distinction is that these related methods have not considered a case for the ability to use the topological information of BVHs to increase workloads at faster rates, since the maximum number of BVH node pairs created when two nodes intersect is constrained by the number of children per-node. In order to increase workloads at faster rates, these methods are required to change their BVH construction scheme and thus, traversal logic, in order to incorporate having a larger set of children per-node to speed up traversal rates.

2.3.3 Stackless Traversal

Some literature also describes methods which adopt stack-less traversal and with whom the presented method shares a similar design premise. Thrane *et al*. [148] proposed the first stack-less hierarchy traversal algorithm, which used a forward pointer called escape index to facilitate depth-first searches. This method is extended by Damkjær [63] to handle collision detection between two BVHs. Damkjær [63] also evaluated several algorithms in scenarios with different setups for performance, scalability and robust-

ness and found the dynamic stack algorithm to perform best in most cases due to the chosen descending heuristic (BV with larger volume traversed first). However, the method's traversal rule is unpredictable (the stack cannot always be omitted) and their is a lack of parallelism suitable for GPUs. Wang *et al*. [154] recently adopt Damkjær's alternative leaf algorithm [63] to generate the BVTT front and gather collision pairs, which gave competitive performance. However, the structure of the generated BVH is still an explicit one, which forces the classical way of BVH traversal.

Hapala *et al*. [52] present an iterative method for ray-tracing on CPUs and GPUs with backtracing and a state-machine to infer which nodes to process next. Barringer and Akenine-Möller [8] present a another stack-less approach with full restarts, while Laine [81] encodes the traversal trial using bit information. In this chapter, the presented method is entirely GPU based and strictly forward stepping with no notion of backtracing nor state-machines that are used during traversal. The topological structure of the BVH is instead used, which is encoded as memory locations and offsets, to infer the BVH nodes to traverse next and additionally using this information to increase workloads at faster rates.

2.3.4 Data Parallel Models for Graph Processing

Parallel BVH traversal shares many challenges with large scale graph problems on GPUs, where issues of load-imbalance (irregularity), control-flow divergence, non-coalesced memory access patterns are most common [86,95]. Harish *et al*. [53] present one of the earliest solutions to solve breadth first search (BFS), single source shortest path, and all-pairs shortest path, while later works such as Cederman *et al*. [16] and Tzeng *et al*. [149] also address issues of load imbalance at the thread-level. Aila *et al*. [1] investigated the related difficulties of divergence on GPUs in context in ray-tracing.

Recent work focuses on the design of general frameworks for different kinds of large graph structures on GPUs such as the scheduling model for irregular inhomogeneous workloads proposed by Steinberger *et al*. [129]. Khorasani *et al*. [74] present a CUDA based model focusing on minimising warp divergence by coarsening parallelism to CUDA warps. Other works have also investigated languages and frameworks for expressing such large-scale computations. Hou *et al*. [57] previously present a programming language for expressing the BSP model [150] on GPUs by addressing the challenge of producing efficient stream code and barrier synchronisation. The recent

Enterprise [90] and Gunrock [155] frameworks define an iterative BFS traversal of large graphs using the BSP model similar to the influential work of Merrill *et al*. [95]. In contrast, the inspired method which is presented in this chapter focuses on the specific problem domain of simultaneous BVH traversal for pair-wise collision detection but also borrows key ideas such as GPU based parallel BFS as a building block. Further, these methods are optimised in large part for massive load imbalance across vertices (as seen in scale-free graphs), but BVHs/BVTTs do not have that kind of imbalance. Thus, different optimisation decisions may be appropriate.

2.4 Method Overview

This section describes an overview of our parallel traversal algorithm with an illustrative overview shown in Fig. 2.2. The following paragraphs and sections assume that the BVH(s) in question are already constructed and reside within GPU memory. Thus, in our efforts we concern ourselves only with the task of traversing these BVH tree in parallel on the GPU.

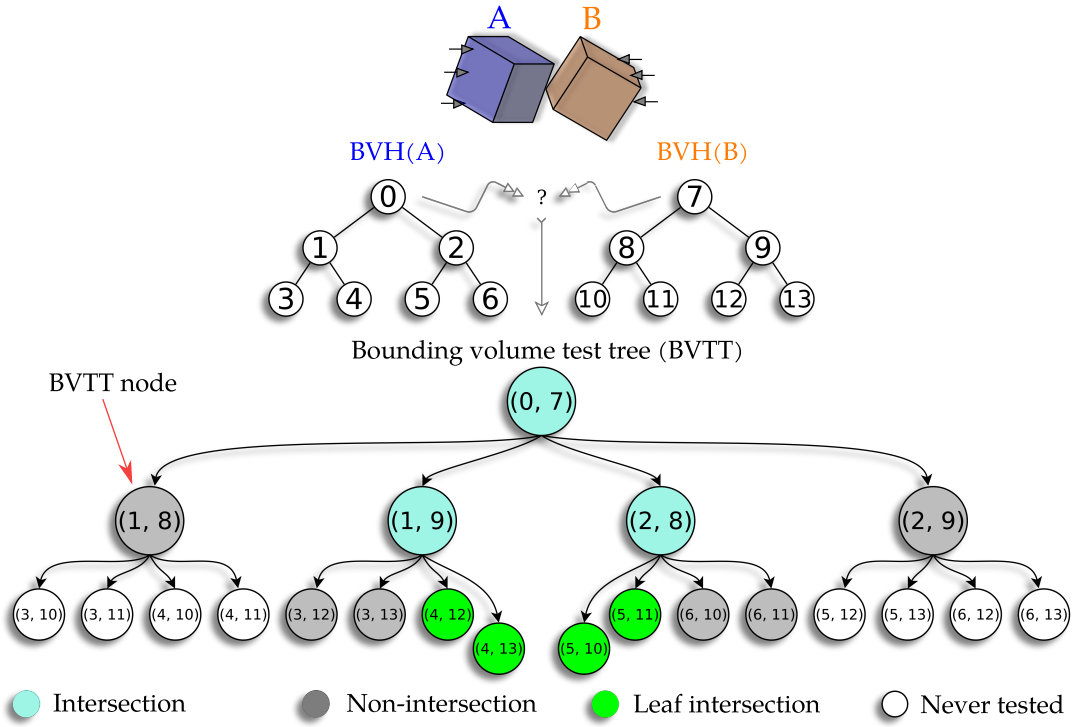


Figure 2.2: *Example illustration of a bounding volume test tree (BVTT) resulting from traversing two BVHs: All shaded nodes of the BVTT represent where intersection tests are performed.*

Within our traversal algorithm, we refer to a pair of BVH nodes which are tested for

intersection as a *BVTT node* and additionally refer to each such BVH node as an *entry node*: During traversal, a BVTT node is discarded after a bounding volume intersection test, such that if the result is true, the BVTT node is *expanded* by replacing it with a new subset of BVTT nodes. This new subset is constructed by pairing the descendants of each entry node with those of the other. Alternatively, pairings may be produced between either of the entry nodes and the descendants of the other if this entry node is a leaf. If the tested entry nodes do not intersect, no further intersection tests are performed with their descendants. The search for geometry that is in close proximity is complete if the BVTT node is a leaf-pair. In general, this process is repeated until *completion*, i.e. the state of reaching BVTT nodes where no further intersection tests can be performed.

In practice, we accelerate simultaneous BVH traversal by expanding the BVTT in a bulk-synchronous parallel manner (Section 2.6.1), where processing threads evaluate the BVTT at the same level simultaneously. The BVHs and BVTT are stored in global memory. The BVTT is maintained in an array that we call `srcFrontier` in a format that aides the parallel access (Section 2.5). At every iteration of expanding a BVTT, threads fetch BVTT nodes from `srcFrontier` and test for intersection between respective entry-nodes. If there is an intersection, the descendant nodes are paired and cached as the BVTT nodes for the next iteration in local shared memory. In order to increase the parallelism of this process, we start the expansion at a level deeper from the BVTT root, and pair descendants deeper in the BVH if there is an intersection between paired bounding volumes (Section 2.6.2). Once the local memory cache is full, newly paired descendants are flushed to another global memory array that we call `dstFrontier` (Section 2.6.3). Finally, `dstFrontier` and `srcFrontier` are swapped and the iteration for the next BVTT level is repeated until there are no more BVTT nodes in `srcFrontier`.

2.5 Data Storage and Runtime

This section describes the BVTT and BVH node representations that are used to enable efficient storage and runtime access for our topologically driven workload expansion scheme described in Section 2.6. In Section 2.5.1, we describe how the BVTT is stored in memory for runtime access. In Section 2.5.2, we then describe how each BVH is ordered in memory under the assumption of an implicit tree structure. Finally we describe how BVH data is accessed at runtime by using memory offsets and node

indices in Section 2.5.3.

2.5.1 BVTT Storage and Representation

In this section, we briefly describe how the BVTT nodes are stored with global memory using `srcFrontier` and `dstFrontier`. This method underpins our bulk-synchronous parallel traversal scheme which is describe in Section 2.6.1.

In practice, a BVTT node is represented as an index-pair where each index is a location of a BVH node in the global memory. We refer to each index as an *entry* as it identifies an entry node of the current and next working set of BVTT nodes inside `srcFrontier` and `dstFrontier`. The BVTT is stored as a large contiguous array in order for threads to access GPU global memory in contiguous and aligned blocks [36] (see Fig. 2.5). The availability of vector load/store instructions on certain GPU architectures allows for efficient bandwidth utilisation which can be beneficial since address accesses of each thread can be combined with single memory transaction issued due to the one-to-one sequential and aligned access to memory [22].

2.5.2 BVH Storage and Representation

Having described how BVTT nodes are stored in Section 2.5.1, we now describe our BVH storage scheme under the assumption of implicit trees in this section.

We propose a novel representation and indexing scheme for BVH nodes that enables instant computation of the BVH that a node belongs to, and the descendants of this node. BVHs are stored compactly in a contiguous array at known offsets with the first at the zeroth offset, whereby the employed topology representation is an implicit binary-tree that is full and complete with nodes stored in a pre-order traversal manner (*cf.* Fig. 2.2, "BVH(A)"). For simplicity, we assume that each BVH is padded by rounding the number of leaf-nodes to the nearest power-of-two to enable *implicit indexing* of the descendants of any node. Though padding can potentially result in a higher memory footprint, only bounding volume information is stored per-node (its "payload"). Further, information that is referencing geometry which is associated with each leaf-node can be stored separately and inferred implicitly by using the relative position of each leaf node on its level in the BVH.

Given an entry-node with implicit index i, its k-th descendant that is n levels ($n \geq 0$)

deeper than i can be inferred by

$$j = (2^n i + 2^n - 1) + k, \quad k \in \left(0 \,..\, 2^n\right], \tag{2.1}$$

where $i, 0 \leq i \leq N-1$ is the position of node i relative to the root node of a BVH with N nodes, and j is the k-th descendant descendant of node i. This representation is strictly forward-stepping and infers the descendants of a node using statically known formulae and index information which is used as described in Section 2.6.2. Readers are referred to Chapter 4 for the full derivation of Eq. (2.1).

2.5.3 Layout Arrays

In this section, we describe additional low-overhead and simple data structures which we use to aid data access as we simultaneously traverse multiple BVHs.

An additional set of arrays, termed *layout-arrays*, is also maintained to store minimal BVH metadata which is used for computing positional offsets of nodes relative to the root of their BVH in memory, and their descendants at runtime. Layout arrays have the same capacity as the number of BVHs being evaluated, and store low-cost information such as offsets and depths (see Fig. 2.3). Layout arrays can be pre-computed once, e.g. on the host (CPU), during initialisation, and then uploaded to the GPU since all information about each BVH may be known at this time.

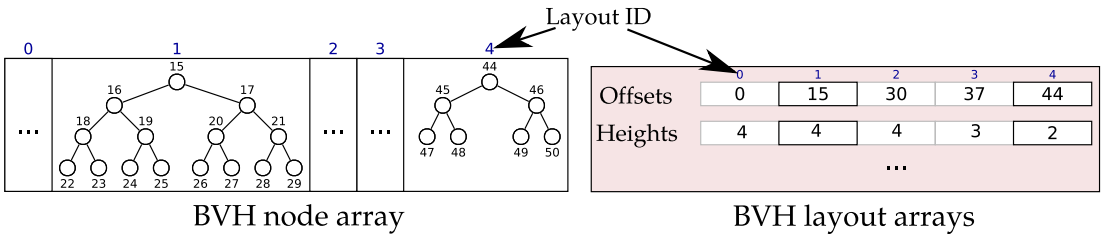


Figure 2.3: *BVH nodes are stored compactly in one memory buffer (BVH storage array) with addition set of of small arrays holding metadata about each BVH which is used to infer node descendants at runtime.*

Determining BVH Information on the GPU: To compute the information of a BVH node (e.g. descendants) from a given BVTT node, a unique ID corresponding to each BVH is required since the entry of this BVH node is just a memory location (index). We refer to this unique ID of each BVH as the *layout ID*. The layout ID is used to access layout arrays for the information about the BVH containing a given entry-node.

We compute the layout ID of a given entry-node by performing a slightly-modified form of the lower-bound binary search [23] over the layout array of BVH offsets,

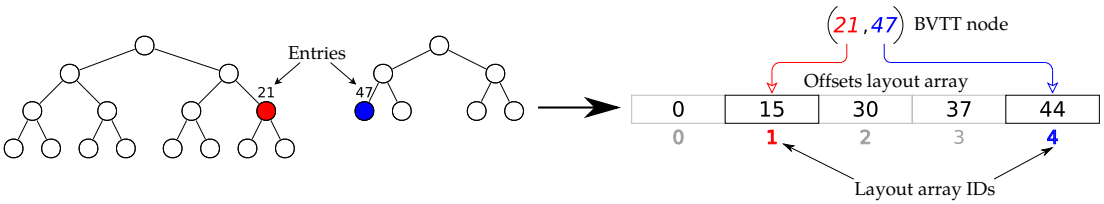


Figure 2.4: *Computing the* layout ID *of an entry-node's BVH given an entry's value e.g. 21 or 47: We search for the lowest insertion index of an entry in the layout array of BVH offsets using a lower-bound binary search and subtract one from this insertion index.*

using the entry's value (memory index) as the search target (see Fig. 2.4). The layout ID gotten from this binary search is then used to access layout arrays for information of the BVH containing the entry-node (e.g. its depth). The overhead of performing this search operation is negligible since it is done in fast GPU local memory with $O(\log_2 N)$ complexity, thanks to the binary search. Note also that, our method can handle both cases of static and dynamically changing BVHs since we only require that BVHs follow our storage representation. In addition, the implicit representation of BVHs greatly simplifies the construction process which is ideal and lends itself well to parallel construction methods on GPUs [83] (see also Chapter 3).

2.6 Algorithm

This section provides the core descriptions of how simultaneous BVH traversal is implemented on the GPU using the BSP model. We first describe the general steps to perform GPU traversal in Section 2.6.1 and then describe the aforementioned topology-driven workload expansion scheme in Section 2.6.2. Finally, a description of how intermediate BVTT nodes are written to global memory at the end of each iteration on the GPU is provided in Section 2.6.3.

2.6.1 Parallel Traversal

We provide details of our traversal algorithm in this section, exploring the individual steps necessary to traverse multiple BVH trees simultaneously on the GPU.

Our method evaluates the intersection of BVHs by iteratively expanding the BVTT using breadth-first search (BFS) as the core parallel primitive for traversal. The steps of Algorithm (1) outline the pseudocode of the method. The host (e.g. CPU thread) will invoke the GPU by calling `gpu_traversal()` in an loop which will terminate

Algorithm 1: Iterative Bulk-Synchronous Traversal

```
   // @arguments
1  // [input]  srcFrontierDef:  1st kernel
2  // [input]  srcFrontier
3  // [output] dstFrontier
4  HOST traverse (srcFrontierDef, srcFrontier, dstFrontier, ...)
5      converged ← False
6      src ← srcFrontierDef // initial BVTT nodes
7      dst ← dstFrontier
8      do
9          gpu_traversal(src, dst,...)
10         if src == srcFrontierDef then
11             src ← srcFrontier
12         synchronise()
13         count ← dstFrontierSzRequest()
14         if count == 0 then
15             converged ← True
16         else
17             dstFrontierSzReset()
18             swap(src, dst)
19     while converged ≠ True
20     return

1  GPU gpu_traversal (srcFrontier, dstFrontier, ...)
2      ▷ Phase 1:  read
3      data ← read(global_id, srcFrontier, ...)
4      ▷ Phase 2:  traverse
5      if intersection(...) then
6          expandBVTT(...)
7      ▷ Phase 3:  write
8      write(dstFrontier, ...)
9      return
```

when the traversal operation is complete. After invoking the GPU, the host must wait for the current iteration to complete which is represented by a call to `synchronise()`. Once the GPU has finished, the host will then read the new number of BVTT nodes from the GPU using the function `dstFrontierSzRequest()` which is a GPU-to-Host memory copy command for a single four-byte integer value. The value read by the host determines the workload size for the next iteration and will be used to check if the traversal operation has completed.

The contents of `srcFrontier` and `dstFrontier` in Algorithm (1) are distinguished to be read- and write-only, respectively, in order to implement *double buffering*, which is used to alias the output of one iteration as input for the next (see Fig. 2.5). Swapping will occur at the end of each iteration on the host with all data remaining on the GPU.

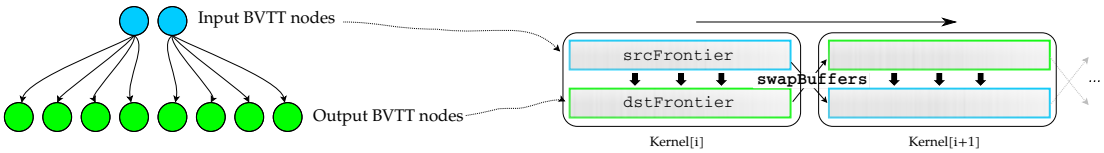


Figure 2.5: *Two buffers are used as main storage for BVTT nodes, the first buffer* `srcFrontier` *holds the input nodes that are evaluated for intersection while the second buffer* `dstFrontier` *stores the output nodes from BVTT expansion. The output of one iteration becomes the input of the next as we maintain all traversal data on the GPU.*

Operations Done by GPU Threads

To start traversal on the GPU, the host will launch approximately as many GPU threads as there are BVTT nodes in `srcFrontier` (*cf.* Fig. 2.5). This will be either the starting amount of default BVTT nodes if it is the first iteration, or resulting amount of the last iteration returned by `dstFrontierSzRequest()`. In phase 1 of Algorithm (1), each thread will read a BVTT node from `srcFrontier` into private register memory and then subsequently read the bounding volume information of each entry-node to perform intersection tests. Phase 2 defines the main body of computation performed by a thread: Using the BVTT node information that is now in private register memory, a thread will then proceed to evaluate it for intersection followed by expansion of the BVTT with new BVTT nodes if the entry-nodes are found to intersect. Finally, in phase 3, threads collectively copy the new BVTT nodes to `dstFrontier` for the next iteration.

2.6.2 Work Expansion

In this section, we describe our workload expansion scheme which based on the traversed tree topological. Thus, we will introduce the concepts of *static workload expansion* and *adaptive depth-stepping* which are used to overcome GPU under-utilisation resulting from the small workloads of testing higher levels of BVHs, and to control the rate of traversal.

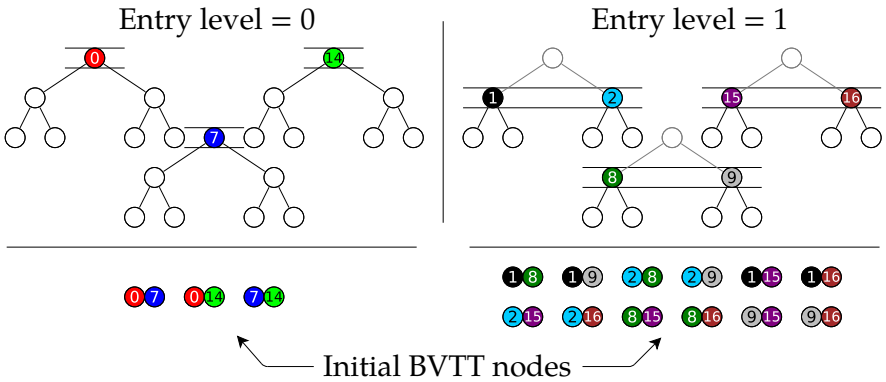


Figure 2.6: *An example of* static workload expansion *(excluding self-collisions) where the de-facto entry-level (e.g. root level) is deferred to descendants at deeper levels in the BVHs to create larger input size for the first iteration(s).*

Static Workload Expansion

Evaluating the levels closest to the root nodes can yield small workloads compared to the ideal size required to maximise GPU utilisation. To increase workloads for the initial kernel(s), evaluation of BVTT nodes that are constructed from the root nodes is deferred to those constructed from their descendants at lower levels. Fig. 2.6 provides an illustrative example of deferring the *entry-level* of three implicit hierarchies.

Given an entry-level $l_e, 0 \leq l_e \leq H-1$ of a BVH with height H, we compute its nodes using Eq. (2.1) with $n = l_e$ and $i = 0$. Accounting for the actual memory locations of each node (i.e. c_j) amounts to simply adding the storage offset of the BVH. Once the entry-level of each BVH is computed, the set of BVTT nodes evaluated in the first iteration of traversal is then obtained by pairing every node in the entry-level of one BVH to those of another. Deferring the entry-level yields approximately $2^{2l_e}E$ BVTT nodes, where

$$E = \frac{N(N-1)}{2} + S$$

is the number of collision checks between N BVHs, with S representing the number of self-collision checks. Such an increment can average-out the workloads over multiple

iterations while also reducing total number of kernels since entry-level BVTT nodes can be pre-computed on the host and uploaded once to the GPU as `srcFrontierDef` (Algorithm (1)) which is then recycled as a starting point for traversal.

Adaptive Depth-Stepping

Recall that expanding the BVTT is the process of creating new BVTT nodes from the descendants of every pair of BVH nodes that are found to intersect. We introduce the concept of adaptive depth-stepping to infer the distance to such descendants while accounting for any differences between the depths of tested BVHs. So, in what follows, the term *depth-step* refers to the (jumping) distance from a BVH node to its descendants that is computed at runtime: This allows us to (1) continue sprouting the descendants in one BVH while reaching the leaves of another, (2) further increase workloads at faster rates while reducing the number of kernels to complete traversal, and (3) tune for performance when writing to global memory.

We compute the depth-step by:

$$\Delta d = \min(\mu, \Delta l), \quad 0 \leq \Delta d \leq H - 1, \tag{2.2}$$

where

$$\Delta l = (H-1) - \lfloor \log_2(i+1) \rfloor,$$

is the distance to the leaf-level of the BVH containing a BVH node i, and H is the height (depth) of this BVH. The variable μ, $1 \leq \mu \leq H-1$ is the user-specified *parameter of expansion*, which is used to control the maximum possible depth-step threads are permitted to use. (Note that Δd is zero if an entry-node is a leaf). Once Δd is known, the descendants of an entry-node are then determined by using Eq. (2.1) with $n = \Delta d$, which is then followed by BVTT-expansion.

2.6.3 Writing Traversal Output

Having described our workload expansion scheme in Section 2.6.2, we now describe how we write BVTT nodes to `dstFrontier`, which holds the output. The approach is designed upon the BSP philosophy for fully utilising the massive parallelism of modern GPUs.

All new BVTT nodes written to `dstFrontier` will be first accumulated in local shared memory and then flushed in coarse-grained chunks to global memory to prevent individual thread access to global memory as shown in Fig. 2.7. Thread-groups are

used to achieve this by using an iterative *write-wait-flush* memory update scheme. Algorithm (2) outlines the steps of how the threads `T` that performed expansion as part of a group `G` copy their collective subset of BVTT nodes to `dstFrontier`. A fixed-size region `Q` is allocated in local memory. At runtime this region is filled (written to) and then flushed (copied from) iteratively until `G` has copied all collective BVTT nodes to `dstFrontier`. At each iteration, `G` write to `Q` with flushing done to asynchronously copy the accumulated BVTT nodes from local to global memory[1].

Algorithm 2: Synchronised writes

```
   global dst_offset                          // size of dstFrontier
1  local base_offset
2  i ← 0
3  checkpoint ← 0
4  do
5      if num_data > 0 then
6          c ← atomic_add(C, num_data) - checkpoint
7          if c < κ then
8              r ← capacityQ − c                  // remaining space
9              w ← min(num_data, r)                 // amount written
10             write(Q, c, data, w)
11             num_data ← num_data - w
12     synchronise_group()
13     s ← min(C - checkpoint, κ)                    // queue size
14     if s > 0 then
15         checkpoint ← C                           // Q ≡ ∅
16         if local_id == 0 then
17             base_offset ← atomic_add(dst_offset, s)
18         synchronise_group()
19         async_copy(dstFrontier, base_offset, Q, s)
20     else
21         break
22     i ← i + 1
23 while (i × capacityQ) < M
```

The chosen thread group size and allocated size of `Q` have a direct effect on the number of iterations taken to copy all BVTT nodes to global memory, which is also dependent on the maximum possible output. In a given traversal kernel, the total number of iterations to copy all BVTT nodes of a group `G` to `dstFrontier` is determined

[1] We used the OpenCL built-in function `async_work_group_copy`

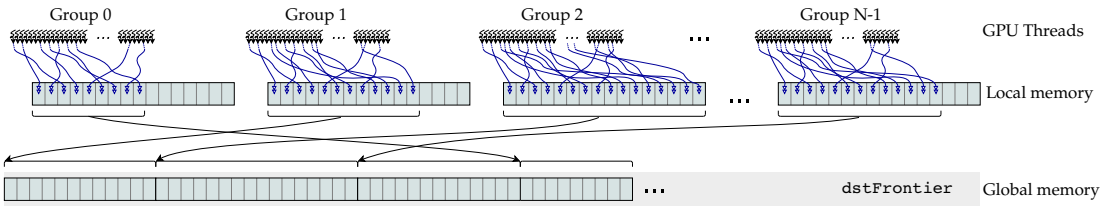


Figure 2.7: *Updating* `dstFrontier` *using iterative buffered copies from local to global memory.*

by

$$\texttt{iters} = \begin{cases} 1 & : \text{if } \texttt{M} \leq \texttt{capacityQ} \\ \lceil \frac{\texttt{M}}{\texttt{capacityQ}} \rceil & : \text{if } \texttt{M} > \texttt{capacityQ} \end{cases} \tag{2.3}$$

where $\texttt{capacityQ}, 1 \leq \texttt{capacityQ}$, is the parameter to control the capacity of the local memory region `Q` and $\texttt{M} = 2^{2\mu} \times \texttt{groupSize}$ is the maximum possible output size of a group (i.e. if all threads in the group performed expansion), where `groupSize` is the parameter to control the thread group size.

Synchronising Threads During Shared Write-Access

Writing to the local memory region `Q` can impact performance since it is a shared resource. A local shared variable `C` is used as a counter which allocated per thread-group. `C` is an atomic offset to the shared fixed-size region `Q` and it is accessed by threads with data to write. At each iteration, threads `T` in a group compete for write-access to `Q` by atomically adding to the counter `C` (line 5). Each successful thread reserves a region to write its BVTT nodes such that those obtaining a valid offset that is within the bounds of `capacityQ` will then asynchronously write to `Q` (lines 6-10).

In essence, the threads that have data to write in the current iteration simultaneously contribute toward computing the offset of their collective output relative to a common base address in `dstFrontier`. This base address is computed by the first thread of the group as a final step before flushing, which is done by using one global atomic add operation after `Q` is filled (lines 12-20). Since `Q` is the sole interface to global memory, lone-thread accesses to global memory is reduced significantly, which can be more expensive to synchronise as workloads increase. We note that this scheme is in fact similar to Garanzha *et al*. [41], however they use the first thread in a batch (CUDA warp) to compute the base offset into a global memory region whereas we use the first thread in a group.

A Heuristic for Choosing the Parameter of Expansion

It may at times prove difficult to choose the parameter of expansion μ given all others and the hardware constraints which must be considered. This parameter has a direct effect on a number of features in the presented method by effectively providing a fine level of control over the rate of traversal. To facilitate the choice of μ, a simple formula is proposed in order to estimate a maximum value $\bar{\mu}$ subject to size constraints on `capacityQ`. The purpose is to at-least guarantee a minimum number of threads that will write *all* their BVTT nodes into in `Q` a single iteration. Assuming the worst-case, where every group thread writes $\beta = 2^{2\mu}$ BVTT nodes, $\bar{\mu}$ can be computed by

$$\bar{\mu} = \left\lfloor \log_4 \left(\frac{\texttt{capacityQ}}{\alpha} \right) \right\rfloor, \quad 4 \leq \texttt{capacityQ} \tag{2.4}$$

where α, $1 \leq \alpha \leq \left\lfloor \sqrt[4]{\texttt{capacityQ}} \right\rfloor$ is a user-specified value for the minimum number of group threads $\in$ `T` guaranteed to write all their BVTT nodes in single iteration. Thus, the guaranteed threads will collectively write $\alpha \times \beta$ BVTT nodes to `Q` in the current iteration, such that the n^{th} thread to atomically offset `C`, where $n = \lceil \frac{\texttt{capacityQ}+1}{\beta} \rceil$, will write at-most

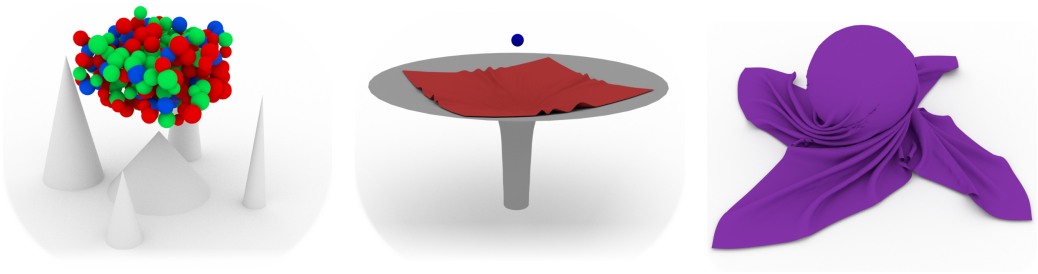
$$\texttt{capacityQ} \pmod{\beta}$$

BVTT nodes to `Q` and the rest will be written in the next iteration.

2.7 Results

This section describes the results of experiments and evaluates the presented method. A `C++` software prototype implemented with OpenCL 1.2 was analysed using the AMD Radeon R9 280X (3GB VRAM, 32KB Local memory) and NVIDIA Geforce GTX 960 (4GB VRAM, 49KB Local memory) GPUs. Three benchmarks (*cf.* Fig. 6.1) from the UNC Dynamic Scene Benchmarks dataset [25] where used for evaluation purposes: *NBody* (Fig. 7.8(a)) has the largest number of objects at 305 with a total of 146K triangles, it is a rigid-body simulation involving many interacting objects without self collisions. *Funnel* (Fig. 2.8(b)) is a soft+rigid body simulation and is the smallest benchmark made up of four low-resolution meshes (total of 18.5K triangles). In this benchmark, the primary interactions occur between the cloth and funnel. *Cloth-Ball* (Fig. 2.8(c)) is another soft+rigid simulation with two objects that have 92K triangles in total. We use simple axis-aligned bounding boxes (AABB) storing one triangle per leaf-node. During experiments, the kernels are executed at-least eight times to reduce

potential noise in time measurements because of system *warm-up* overhead. However, no significant differences were observed between test runs.



(a) NBody (b) Funnel (c) Cloth-Ball

Figure 2.8: *UNC Dynamic Scene Benchmarks used for evaluation* [25].

2.7.1 Performance

Table 2.1 summarises the performance of the presented algorithm, and considers both inter- and intra-object intersection tests for the case of Cloth-Ball and Funnel. The presented results are based on the heaviest workloads (colliding leaf-nodes pair) experienced at the most demanding time-step in each benchmark.

The presented method is able to perform parallel simultaneous queries in real-time. Execution time is fastest on *Funnel* with query time under 1ms. *Cloth-Ball* takes the longest time (6.43*ms* on the GTX 960). This benchmark has the largest workloads with over 3.1*million* overlapping leaf-node pairs due to the self-collisions induced by the cloth's motion. For this benchmark, our method is able to complete traversal within 6.5*ms*. *NBody* has the lowest number of leaf-node overlaps because it is a rigid body simulation. Its BVHs are approximately twice as much slower to evaluate than *Funnel* due to the larger number of objects (305), and hence, the resulting BVTT. The results reveal our method's strong ability to exploit large scale parallelism on GPUs to quickly evaluate a large number of BVTT nodes for pair-wise collision detection.

Table 2.1: *Our performance results for simultaneous parallel BVH traversal involving inter- and intra-object collisions.*

				Time (ms)	
Benchmark	#Triangles	#Objects	Colliding Pairs	Geforce GTX 960	Radeon R9 280
Cloth-Ball	92K	2	3.1e6	6.43	3.08
Funnel	19K	4	3.14e5	0.99	0.57
N-Body	146K	305	1.176e5	2.42	1.16

Performance comparison against collision-streams model [138]

A performance comparison was also carried out against the "collision-streams" model by Tang *et al*. [138]. Comparisons were done using the time-step (i.e. frame) with the heaviest workloads on each benchmark. We did not include intra-object collisions for Cloth-Ball and Funnel to ensure that workloads fit in our global memory buffers for Tang *et al*. Also, a trimmed down implementation of the collision-streams model was used which performed only pair-wise collision queries to ensure a fair comparison. Exact front-tracking (not *deferred*) was used together with *stream registration* based on segmented locking mechanism (see Tang *et al*. [138] for details). According to Tang *et al*. [138], deferred front tracking simply trades memory overhead for additional runtime computations.

To emulate the BVTT node cache (moving front) used by Tang *et al*. [138], experiments where setup as follows: For each benchmark, we extract a pair of keyframes $(k_t, k_{t+\Delta t})$ which are consecutive in time, with each key-frame k representing the geometry of a particular time-step t in that benchmark. Next, we build the BVH of each mesh in the benchmark for k_t and $k_{t+\Delta t}$. The BVTT node cache is then created by traversing the BVHs of the meshes of k_t until completion and saving the BVTT nodes where traversal terminates as described by Tang *et al*. [138]. Performance comparisons were then performed using BVHs constructed from $k_{t+\Delta t}$ since it is possible to use the BVTT node cache built from k_t as input for traversal at $t + \Delta t$, thereby allowing the collision-streams model to have a valid cached input set from a 'previous' time-step. We have not included the cost of work redistribution for the collision-streams approach in our evaluation.

Fig. 2.9 shows speedup were comparisons are based on BVH traversal times to find the set of potentially colliding triangle pairs. Performance of our method on all benchmarks is faster with an average speedup of 4.4× on the R9 280X and 4.3× on the GTX 960. The highest speedup is on NBody at 7.1× for R9 280X (6.2× on the GTX 960) which has the largest workloads in our comparison setup. In general, we found that adapting the streams model on arbitrary GPU architectures is non-trivial due to its dependence on the available amount of local memory for the work-stacks and exploiting L1 caches. Our method is an efficient and much simpler option for mapping traversal to GPUs

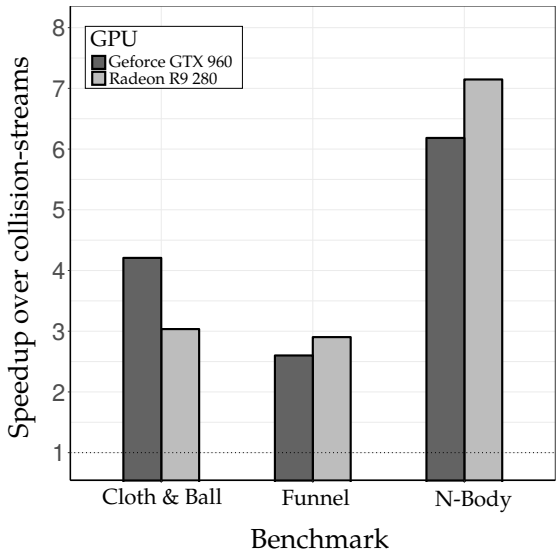


Figure 2.9: *Speedup over the "collision-streams" model* [138]

2.7.2 Parameter Effects and Trade-Offs

The explorable nature of the exposed parameters can make finding correlations between their configurations and the resulting performance unintuitive with no obvious settings. This section provides a brief discussion and analysis of the effects of changing parameter values.

Parameter of Expansion and the Entry Level

Here we briefly discuss the effects of the entry level l_e and the parameter of expansion μ on execution time for each benchmark (with intra-object collision tests for Cloth-Ball and Funnel). Increasing μ reduces the number of kernels but care must be taken when making the choice of value. For the evaluated range $(1-4)$, making further increments beyond $\mu = 3$ produces a drastic slow-down where the execution-time is on average $3\times$ to $5\times$ slower than choosing a value between 1 and 3. Generally, a choice of smaller values of μ e.g. 2, is a suitable for the case of reducing execution time, even though this choice requires of more kernels to complete traversal. l_e was found to most-useful for statically reducing the number of kernels to complete traversal while providing sufficiently large workloads for the GPU. There are some limitations on the exploitation of l_e however, since its configuration must account for the number of BVHs tested to control the resulting input size. On the Nbody simulation, a more rapid (exponential) performance drop with l_e is observed as compared to the other benchmarks due to the faster rate of increase in the initial input size. For example,

setting $\mu = 2$ and making increments on l_e from 1 to 5 results in a sharp change in execution time from 3*ms* to 16*ms* respectively on the GTX 960.

Local Memory and Thread-group Sizes

The allocated local memory size `capacityQ` of the fixed-size region `Q` and thread group size `groupSize` also have an effect on performance and its scaling properties. Fig. 2.10 shows the results for the change in execution time relative to `capacityQ` and `groupSize`, respectively. Setting either parameter to the highest tested value (e.g. `capacityQ` $= 2^9$ and `groupSize` $= 2^8$ on the R9 280X) while maintaining the other at a minimum (e.g. 2^1) showed slower performance in most cases with the exception of the NBody simulation on the R9 280X. More generally, similar behavioural patterns are observed on both GPUs with the GTX 960 appearing a little more constrained in terms of the optimal choices of `capacityQ` and `groupSize`. Configurations that use mid-range values are sufficient to obtain good performance relative to the worst case for each benchmark. The method favours medium-to-large thread groups (`groupSize` $\geq 2^5$) and allocated local memory size (`capacityQ` $\geq 2^6$) for good performance. The results of Fig. 2.10 are a demonstration of the importance of the trade-offs to be made through the parameters due to their influence on scheduling, which is crucial for portability.

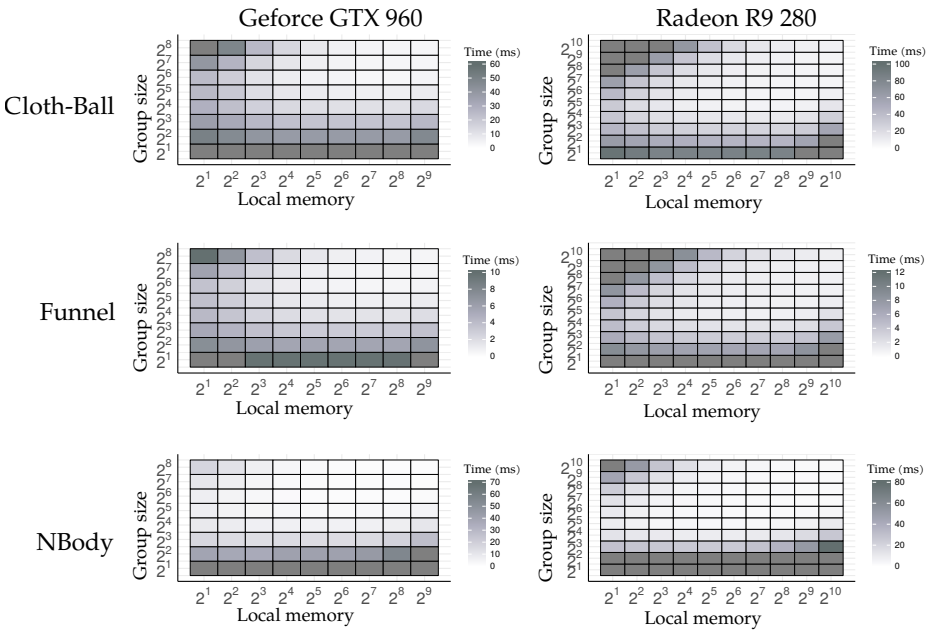


Figure 2.10: *The effects of the allocated local memory size* `capacityQ` *and the chosen thread group size* `groupSize` *on the execution time.*

2.8 Conclusion

This chapter has presented a simple and alternative method for simultaneously traversing a large number of BVHs for collision detection on GPUs. The method utilises the bulk-synchronous parallel (BSP) model to overcome the irregular and data-dependent nature of traversal. The simplicity stems from the use of implicit tree topology to harness the parallelism of GPUs. The chapter described how–using the implicit structure of the BVH–a topologically-driven workload expansion scheme is created. This scheme provides control over the rate of traversal while also increasing workloads for the initial kernel(s). In addition, a simple global memory updating algorithm was described, one that can be controlled to adapt algorithm performance based on the available hardware resources. This approach can likewise be extended with more complex lock-free synchronisation mechanisms using scan primitives such as prefix-sum [122]. The presented method can evaluate complex hierarchies in real-time, and with speed-ups of up-to 7.1× over the widely used "collision-streams" model.

On Limitations and Future Work

The presented algorithm faces a number of limitations. For example, it minimises the compute workload per thread while potentially increasing the DRAM traffic as a side-effect: Threads perform just one intersection test, such that in order to perform it, they need to stream data from global memory. The BVH node array is sparsely populated due to padding, which can easily cause excessive L2 cache and global memory traffic. Such padding can, in the worst-case, also double the storage requirements per BVH subject to the number of leaf nodes. In addition, the specific design strategy of using a one-to-one mapping between threads and BVTT nodes may not utilise the benefits of GPU caches because there is no opportunity for the reusing BVTT nodes from `srcFrontier`: The initial read operation of phase 1 (see Algorithm (1)) is effectively a *cold start* with no opportunity for explicit data reuse since little temporal locality exists when reading BVTT nodes and BVH node data.

2.9 Postscript

This chapter presented a new BVH traversal algorithm influenced by work in related literature which has partly solved the traversal problem on GPUs. Since our method is fully based on implicit trees it not only reduces the complexity of prior methods,

but also improves the performance. This also shows how developments made in e.g. parallel graph algorithms on GPUs, can be likewise adopted to new domains and can have a notable contribution to future research.

Chapter 3 will describe a novel BVH representation to address some of the limitations, which is integrated into a full collision detection pipeline and is shown to outperform the state-of-the-art.

Chapter 3

Binary Ostensibly-Implicit Trees for Fast Collision Detection

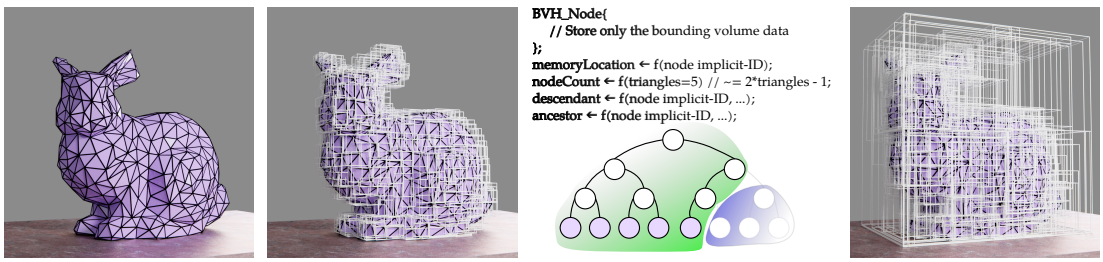


Figure 3.1: *We present a fast encoding of bounding volume hierarchies (BVH) for broad-phase collision detection with low-memory usage. Our method can generate trees supporting canonical indexing of implicit trees without the need for padding memory. We achieve this by observing that for a given number of objects, an almost-perfect binary tree can be completely determined, and the nodes missing for it to be "perfect" can be characterised through simple bit-manipulations. The figure shows a sequence where a BVH is constructed over a mesh using our novel representation. A minimal number of nodes are stored which can be indexed like the heap data structure.*

3.1 Preface

Chapter 2 described an efficient way to traverse multiple bounding volumes hierarchies (BVH) during broad-phase (or mid-phase) collision detection. However, the described procedure assumed that the traversed BVHs are perfect implicit trees which can impose severe restrictions storage when trees are far from balanced. This chapter will describe new method for representing implicit binary trees to eliminate these restrictions while

also overcoming a number of limitations which were highlighted in Chapter 2. We will present a simple, efficient and low-memory technique, targeting fast construction of bounding volume hierarchies (BVH) for broad-phase collision detection. This is achieved using a novel representation of BVH trees in memory. A mapping of the implicit index representation to compact memory locations is developed, based on simple bit-shifts, to then construct and evaluate bounding volume test trees (BVTT) during collision detection with real-time performance. The topology of the BVH tree is modelled implicitly as binary encodings which allows for determining the nodes missing from a complete binary tree using the binary representation of the number of missing nodes. The simplicity of this technique allows for fast hierarchy construction achieving over $6\times$ speedup over the state-of-the-art. Making use of these characteristics, not only it is feasible to rebuild the BVH at every frame, but that using this technique, it is actually faster than refitting and more memory efficient.

3.2 Introduction

Computer graphics researchers have developed diverse methods for accelerating GPU-based broad-phase collision detection by constructing bounding volume hierarchies (BVHs) and evaluating their intersections by expanding bounding volume test trees (BVTT) [35, 45]. Since BVH construction and BVTT expansion are expensive operations, techniques such as BVH refitting and BVTT front tracking are widely adopted to reduce the runtime cost.

Refitting is an operation to build the BVH once or at regular intervals and then resize bounding volume extents or perform local restructuring. Notably, refitting has inherent limitations because the spatial agglomerative structure of the objects which are enclosed within the BVHs is likely to change (potentially drastically) as commonly seen with deformable objects such as cloth and volumetric simulations. Failure to sufficiently capture this spatial structure can degrade performance and worsen runtime storage costs due to an increase in the number of overlapping bounding volumes.

BVTT front tracking, which is an approach to cache the BVTT [75] between frames, can be detrimental for GPU processing because it has a high memory cost and will complicate traversal logic. Also, front tracking assumes that the BVH structures will remain unchanged across simulation frames - otherwise the cached fronts are invalidated by structural changes to the BVH. This assumption does not hold well for scenes involving deformable objects.

One possible way to circumvent these issues is to construct the BVHs and BVTT from scratch at every frame, without refitting or front tracking. However, the bottleneck then becomes the representations that are commonly used for BVH data structures. Most BVH-based methods on GPUs [6, 69, 154] explicitly compute and store the connectivity between nodes, which introduces indirection (see Fig. 3.2), and will affect the construction time due to added overheads. Aside from the fact that nodes must store this connectivity, traversing these BVH trees from one node to a descendant several levels deep requires using loop constructs and memory lookups which can significantly drop GPU performance. Alternatively, existing implicit structures, which do not require storing the connectivity, may either waste a lot of memory due to padding [19], or they may not suit GPU architectures for the construction [20].

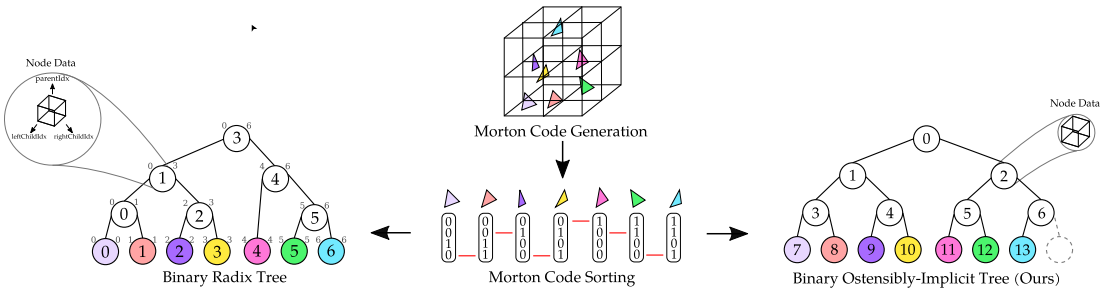


Figure 3.2: *Illustrative comparison with the binary radix tree (left)* [6]. *Each leaf node is associated with a 4-bit Morton code which is in lexicographical order, and initially generated from the position of an object (e.g. triangle). Our technique (right) offers an implicit structure which is derived from the total number of objects, supporting fast indexing, and without memory padding. The explicit structure of the radix tree is encoded in the linear range of keys covered by an internal node which affects storage costs and limits construction performance.*

This chapter will present a fast, memory-friendly, parallel broad-phase collision detection approach to construct and traverse large-scale hierarchies - which is characterised by using an implicit binary tree for final topology, and a novel way to encode this (logical) tree layout. Our method is supported by the notion of an *ostensibly-implicit* tree data structure as illustrated in Fig. 3.1 (middle) and Fig. 3.2 (right), which is a novel implicit binary tree structure specially designed to achieve fast construction and traversal on GPU architectures. In this structure, the BVHs are represented by series of implicit binary trees. The relationship between nodes can be computed by closed-form descriptions which can be implemented efficiently in hardware using fast bit-shifting operations. We also provide formulae to associate node indices with respective memory locations, which results in compact memory storage and fast ac-

cess for construction and traversal. It supports fast bottom-up construction based on Morton codes, which is more suitable for modern parallel architectures compared to heap-based top-down constructions. Our method achieves a construction rate of over 4.7 billion nodes per second and is over 6× faster than the state-of-the-art solution [6].

Our evaluation with the UNC dynamics benchmarking suite [25] shows that our collision detection pipeline is 1.3× faster than the state-of-the-art [154] while using 5× less memory and re-building BVHs every frame. This is achieved by sidelining the use of monolithic BVHs for the entire scene in favor of BVH-BVH tests where traversal workloads scale according to the proximity between meshes. These savings are also due to our simplified setup in which explicit BVTT front tracking is avoided to mitigate inhibitive memory costs.

3.2.1 Chapter Contributions

The contributions of this chapter are summarised as follows:

- *Compact Implicit Tree* – We represent the BVH as a novel layout called the *ostensibly-implicit tree*, decoupling storage costs from the implicit structure and enabling fast construction (Section 3.4). Thus, we use an implicit binary tree for representing topology which is encoded using bitwise logical operations.
- *Construction* – We offer a fast $O(n)$ algorithm which maps well to GPU architectures and without complex tracking of radix key-ranges (Section 3.5).
- *Lightweight Collision Detection Pipeline* – We present a simple and fast broad-phase collision detection pipeline where we construct the BVH and BVTT from scratch at every frame (Section 3.6).

Organisation: The rest of the chapter proceeds as follows: We review related work in Section 3.3, then describe the binary ostensibly-implicit tree for representing BVHs in Section 3.4. Our fast construction algorithm is described in Section 3.5 which we use to build our collision detection pipeline that is described in Section 3.6. We present our experimental results in Section 3.7 and conclude the chapter in Section 3.8 with a discussion.

3.3 Related Work

In this section, we first review methods for constructing BVHs in parallel. Next, we review methods based on implicit tree structures to optimise search problems in related areas, and simpler tree updates. Finally, we review GPU-based approaches for handling collision detection.

3.3.1 BVH Construction

Fast BVH construction is a common problem for collision detection and ray tracing [72, 82, 152]. Our work shares much in common with recent efforts which focus on a multitude of acceleration strategies and trade-offs between construction time verses BVH quality. Lauterbach *et al*. [83] introduce the Linear BVH (LBVH) sorting objects along the Z-curve to facilitate partitioning and significantly improve construction time. Since its introduction LBVH has been extended numerous times and has inspired the construction algorithm presented in our work (see also [42, 69, 112]).

In general, fast construction is achieved with a loss in BVH quality. The BVH quality of these solutions will fall short of the gold standard making them useful especially when the number of queries is relatively small as in collision detection. Karras [69] has presented a technique for depth-first ordered binary radix trees and building the entire tree in $O(n)$ time. The algorithm maps well to GPUs by addressing the shortcomings of prior methods (see e.g. [42]) which generated the hierarchy sequentially for individual tree levels. Conversely, Karras [69] required separate kernels to generate the hierarchy and fit bounding volumes. A bottom-up strategy is proposed by Apetrei [6] which is known to be the fastest, requiring one GPU kernel to build the hierarchy and calculate bounding volume extents. The method is relatively efficient but complex, requiring an analysis of the split positions of internal nodes for establishing a connection between their indices and the ranges of Morton codes that they cover. Our reliance on a topologically implicit structure means that we surpass requirements to establish explicit node-connectivity which is in contrast to the approach of Karras [69] and Apetrei [6].

3.3.2 Implicit representations

Several implicit BVH representations have been proposed in literature which are related to the data structure layout that we describe. Eisemann *et al*. [33] present an implicit representation for partitioning object space to reduce storage costs similar to

the bounding interval tree (BIH) [151]. Their BVH is implicit in the sense that node bounding volumes are inferred at runtime from a set of bounding triangles and only storing a few indices. The minimal bounding volume hierarchy (MBVH) [9] is another implicit structure in the form of a full and complete binary tree for BVH compression. However, while storage per node is reduced, the total number of elements is a constant maximum $2N-1$ nodes. Conversely, we store the minimal number of BVH nodes for a given set of objects to reduce memory costs while retaining the benefits of implicitly-indexed trees.

Cline *et al*. [20] present the well-related lightweight implicit BVH which is indexed like a heap. Their non-parallel solution for generating an implicit tree is done in a top-down manner by recursively splitting the leaf nodes into half - such an operation is not well suited to GPU architectures [83]. The generated tree is also less flexible since leaf nodes may not reside on the same level. In particular, the number of objects enclosed by each node must be known before the lightweight-BVH can be initialised requiring at-least two 'passes' for construction - object partitioning requires that the number of objects in each internal node is known by summing the number of nodes in its children. Their approach will also calculate the total number of BVH nodes using the amortised cost of leaf nodes resulting in additional bookkeeping which will degrade the construction performance. Conversely, we only need to know the number of objects to infer the implicit tree structure. Further, our approach offers an exact closed-form solution to calculate the number of nodes given the number of objects - which can be done using trivial bit-manipulations as described in Section 3.4.

3.3.3 Simpler Tree Updates

In collision detection problems, simpler BVH update strategies such as refitting and selective restructuring are common [77, 82, 84]. This choice is motivated by speed, and in-part by the fact that these strategies are well suited for generalised front-tracking [75] which would otherwise require significant bookkeeping when BVHs are rebuilt from scratch (see e.g. Wang *et al*. [154]). However, a degradation of BVH quality is also inevitable when accumulated deformations within dynamic scenes cause significant increases in the overlap among child bounding volumes. Worse yet, in the case of breakable objects, refitting and selective restructuring are insufficient and a full reconstruction is needed.

Intermediate solutions such as Kopta *et al*. [77] use hybrid methods which heurist-

ically track sub-trees to rebuild (see also [40, 72]). Kopta *et al*. [77] propose a well-related incremental update scheme by combining refitting with local restructuring to modify sub-trees via rotations, and node splitting. However, they still advocate for a full rebuild when extreme degenerations occur, which has seen recent application within GPU-based collision detection [154].

3.3.4 Parallel Collision Detection on GPUs

Since Lauterbach *et al*. [84]'s early work on BVH-based broad-phase collision detection on GPUs, research has taken a number of approaches to accelerate this notoriously difficult task. Within parallel graphics, these methods range from those accelerating collision tests with the BVH, to spatial hashing schemes formulated to obviate the bounding volume test tree (BVTT) in favour of a lower memory footprint and a guaranteed worst-case number of intersecting polygon pairs (see works by [137, 142, 158, 163]). However, these latter approaches are limited in three ways: the grid size is an important factor in the overall performance, some pipelines need to be coupled with normal-cone culling to sustain performance ([137]), and there are restricted opportunities to exploit frame-to-frame coherence for which our approach can be readily extended. Though spatial hashing methods are widely explored, BVH based methods still comprise much of collision detection approaches. Spatial coherence based methods where among the first and performed broad-phase collision detection as a caching scheme with collision-fronts [84, 110]. However, these methods are also limited: GPU parallelism can only be exploited with a sufficiently large collision-front, and traversal logic relies on thread-level private work-stacks which constrain performance due to divergence between threads (see e.g. [85, 138]). Most notably, the reliance on a collision-front for performance is memory intensive and assumes that the underlying BVH structure will remain fixed between frames, otherwise fronts are invalidated.

A number of methods offer in-part successful solutions to solving the memory problem of caching collision fronts. Tang *et al*. [138, 141] have addressed the memory problem by deferring collision fronts to fit BVH node pairs in memory and propagating the BVTT in localised sets of nodes, respectively. However, these approaches curb the memory problem heuristically, they are complex, and their success-rate is not thoroughly investigated. A simpler approach is discussed in [154] which accounts for the order between BVH node pairs to cull up-to 25% of redundant self-collision tests, and

is well suited for the implicit tree setting.

Wang *et al*. [154] present one of the fastest BVH-based methods by ordering and restructuring the collision front and BVH, while using stackless depth-first search (DFS) traversal. Their method proposes to use a histogram sort and auxiliary data structures to reduce random data access patterns arising from front updates, while providing a quality metric to mitigate BVH degradation. Unfortunately, the benefits of restructuring are offset by its cost in scenes with large deformation and their selective restructuring of BVHs yields a complex pipeline since changes must be reflected within the collision fronts. To simplify traversal, we ([19]) present a non-cached front approach using implicit trees to traverse BVHs from pre-specified levels to the leaves, thereby circumventing the drawbacks of explicit collision-front tracking. Notably, this method is promising but requires implicit structures that are prohibitively expensive due to padding which we overcome in this chapter.

3.4 The Binary Ostensibly-Implicit Tree

In this section, we introduce and describe our novel ostensibly-implicit tree layout for reducing the memory costs of BVHs which are stored as implicit tree structures but without need for post-processing to compact data (Fig. 3.3, Section 3.4.1). We also describe a mapping between the perfect implicit tree layout and ours, linking implicit index labels to actual data in memory (Section 3.4.2). For convenience, we assume a binary tree layout (e.g. Fig. 3.4), but the concept is extendable to arbitrary *arity* (Appendix B).

Keep in mind that the layout description provided in this section is used to construct a BVH (Section 3.5) from a mesh representing an object that is will be tested for collision as described in Section 3.6 (see also Chapter 2).

3.4.1 Tree Layout

With a perfect binary tree layout that is full, one can completely remove all pointers and store the pointer-less nodes in an array. This layout is determined by a parameter t, which could be the number of objects such as triangles. However, when t is a non-power-of-two, space still has to be allocated by introducing *virtual nodes* which accommodate for unused elements.

The heap data structure (e.g. [20]) can eliminate virtual nodes but requires post-

processing which will affect construction performance, and nodes may have to store additional reference data.

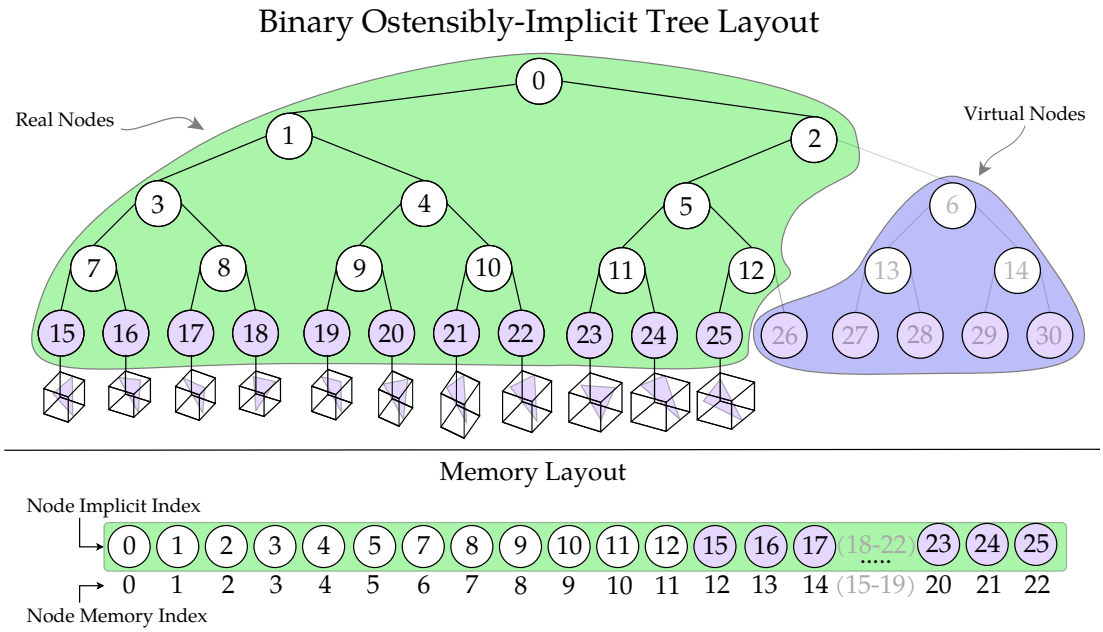


Figure 3.3: *Our tree representation is defined by so-called real and virtual nodes. The real nodes are the actual data (i.e. bounding volumes), and virtual nodes are simply non-existent placeholders. The layout is a* left-leaning *implicit structure, where the real nodes occupy the left-most slots on each level and are implicitly assigned a memory index.*

We resolve this problem using an implicit layout which is free from post-processing (Fig. 3.3) to eliminate all virtual nodes *and* explicit pointers. The idea is to produce a perfect implicit binary tree layout where the virtual nodes are brought to the right-hand-side (see Fig. 3.3, blue nodes), and are then encoded as a series of smaller perfect trees. With this representation, we provide an analytical form to map the remaining real nodes sequentially into the memory, and thus can minimise the memory usage to the size of the real nodes since virtual nodes are not materialised in memory.

Power-Sum Decomposition

We now describe how to decompose the number of objects t into a tree of the real nodes and a series of implicit binary trees of the virtual nodes. This decomposition is used to map implicit indices to compacted memory locations.

To intuitively illustrate the representation of our layout, observe that the residual number of leaves in an implicit binary tree which are virtual nodes is

$$L_v = 2^{\lceil \log_2 t \rceil} - t, \tag{3.1}$$

giving a total count of $L_c = t + L_v = 2^{\lceil \log_2 t \rceil}$ leaves, such that $\log_2(L_c) - \lceil \log_2(L_c) \rceil = 0$. Thus, the total number of nodes in the perfect binary tree will be

$$N_c = 2L_c - 1. \tag{3.2}$$

With N_c, we then seek to find the total number of *real* nodes

$$N_r = N_c - N_v, \tag{3.3}$$

where N_v is the total number of virtual-nodes (refer to Fig. 3.3).

We compute N_v following the observation that L_v may be expressed as a sum of powers-of-two. This observation gives a decomposition of L_v which yields a set

$$\mathcal{X}(L_v) = \{x \in \mathbb{N} \mid x = 2^y,\ y \in \mathbb{N},\ y \leq \lfloor \log_2(L_v) \rfloor\}.$$

More generally, we define

$$\mathcal{X}(L_v) = \{2^{y_1}, 2^{y_2}, \ldots, 2^{y_N}\},\ y_i \in \mathcal{Y}(L_v), \tag{3.4}$$

where $\mathcal{Y}(L_v) = \{y_1, y_2, \ldots, y_N\}$, such that

$$\begin{aligned} y_1 &= \lfloor \log_2(L_v) \rfloor, \\ y_2 &= \left\lfloor \log_2\left(L_v - 2^{y_1}\right) \right\rfloor, \\ &\ldots \\ y_N &= \left\lfloor \log_2\left(L_v - \sum_{i=1}^{N-1} 2^{y_i}\right) \right\rfloor. \end{aligned}$$

The set $\mathcal{X}(L_v)$ is optimal in the sense that it is defined using the largest powers-of-two summing to L_v.

Having decomposed L_v by finding the powers of two which sum up to it, the general analytical form for N_v given $\mathcal{X}(L_v)$ is then evaluated by

$$\begin{aligned} N_v &= \sum_{k=1}^{N} 2x_k - 1, \quad x_k \in \mathcal{X}(L_v), \\ &= 2\left(\sum_{k=1}^{N} x_k\right) - N \end{aligned} \tag{3.5}$$

which will evaluate N_v as a finite sum of perfect implicit-tree sizes containing only virtual nodes as shown in Fig. 3.3. $N = |\mathcal{X}(L_v)|$ is the cardinality of the set $\mathcal{X}(L_v)$, representing the total number of powers of two which sum to L_v.

Binary Encoding

Our approach so far offers a general solution requiring several steps in order to evaluate the total number of real nodes N_r by first determining the number of virtual nodes N_v. We now describe an implementation utilising bit-wise operations to refactor these formulas as simple and fast one-line calculations. In particular, we extensively rely on a function `count_set_bits` to count the number of non-zero bits in a given integer's binary representation. (Note: such a function is easily accessible as a standard language compiler intrinsic e.g. `popc` in CUDA).

Thus, in practice we evaluate Eq. (3.5) by

$$N_v = 2L_v - \texttt{count\_set\_bits}(L_v), \tag{3.6}$$

following a key observation that the i-th non-zero bit, $0 \leq i$, in the binary representation of L_v uniquely identifies a corresponding sub-tree of virtual nodes with 2^i leaves. This sub-tree will have $2 \times 2^i - 1$ nodes. Consequently, by summing over all set bits we arrive at the solution. Also, from Eq. (3.2), (3.3) and (3.6), the exact total number of real node nodes in the tree is

$$N_r = 2t - 1 + \texttt{count\_set\_bits}(L_v). \tag{3.7}$$

We use these solutions to map the implicit index of each node to a unique memory location as described next (Section 3.4.2).

3.4.2 Mapping Implicit Indices to Memory Locations

We now describe a method to compute a mapping between the implicit index of a real node and the location in memory where it is stored - providing a complete solution for generalised pointer-less traversal with *zero indirection*. We use the term "implicit index" to refer to the numerical label given to each node in the perfect tree in breadth first search (BFS) order as shown in Fig. 3.4.

For a given real node, its location in memory is determined by its implicit index, depth level, and the number of virtual leaves in its tree as described in Section 3.4.1. To define our memory mapping, let i be the implicit index of a real node which is at level $l_i = \lfloor \log_2(i+1) \rfloor$, $(0 \leq l_i \leq \bar{l})$, where $\bar{l} = \lceil \log_2 t \rceil$ is the leaf level. Further, let

$$L_{vl} = \left\lfloor \frac{L_v}{2^{\bar{l}-l}} \right\rfloor \equiv L_v \gg (\bar{l} - l) \tag{3.8}$$

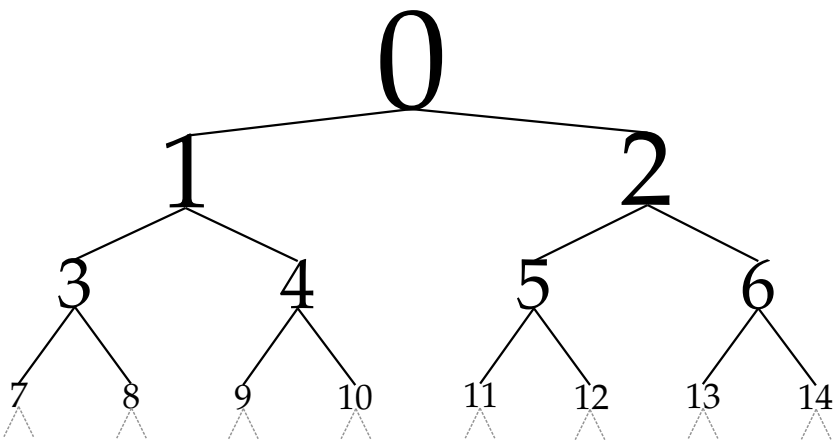


Figure 3.4: *A perfect binary tree which is full and complete with implicit-indices (labels) following a pre-order traversal pattern i.e. breadth-first search labelling. Our algorithm assumes this layout where leaf nodes occupy the deepest level.*

be the number of virtual nodes at level l due to the consecutive and approximate halving of the number of virtual nodes at each level when moving up tree from $\bar{l}$ by $\bar{l}-l$ levels, where » is the bitwise right-shift operator. Thus, the memory location of i is computed by

$$i_m = i - N_{vl}, \tag{3.9}$$

where

$$N_{vl} = 2L_{vl} - \texttt{count\_set\_bits}(L_{vl}), \tag{3.10}$$

which is similar to Eq. (3.6), but with L_{vl} computed as in Eq. (3.8) using $l = l_i - 1$.

Intuitively, our goal in Eq. (3.9) is to account for the number of virtual nodes above l_i from which a memory location can be determined given i (thanks to BFS labelling). Eq. (3.9) provides a seamless solution for bridging between the perfect implicit tree (Fig. 3.4) and our layout Fig. 3.3. The solution is simple and fast (due to bitwise encoding) offering an indirection-free description of data layout in memory.

With these properties, our layout is particularly attractive since it is compact by eliminating the nuances of explicit and/or padded tree structures. The node data (i.e. the 'payload') is smaller compared to the case of including child (and parent) pointers, or when the tree really is fully padded (or perhaps just a few nodes off on the short side from being full). Also, a level l, $0 \leq l$ is only completely filled with real nodes iff

$$2^l < \left\lfloor 2(t-1) \,/\, 2^{\lfloor \log_2 t \rfloor} \right\rfloor.$$

This is in contrast to data structures such as the heap in which all levels, except possibly the last, are filled.

3.5 BVH construction

We now describe a method to construct a BVH using the ostensibly-implicit tree layout. The constructed tree is parametrised by a set of triangles which define a mesh representing an object. Our motivation is to construct the tree very fast to enable frequent rebuilds for the task of collision detection (Section 3.6) - finding the colliding pairs of triangles between two or more objects.

The basic idea of our approach is to utilise Morton order [98] and a specific node layout (which is implicit in our case) to establish a mapping between GPU threads and BVH nodes. Here, we lay emphasis on a GPU implementation since our target application is parallel collision detection, but the method is easily extendable to other implementations (e.g. single or multi-threaded CPU). In this approach, we simplify and extend Apetrei's method [6], but mapping *entire* GPU thread-groups to sub-trees and without explicit tracking of radix-key ranges. Section 3.5.1 first provides a high-level perspective on how the layout is derived from the number of objects. We then describe the algorithm and two implementations in Section 3.5.2.

3.5.1 Hierarchy construction

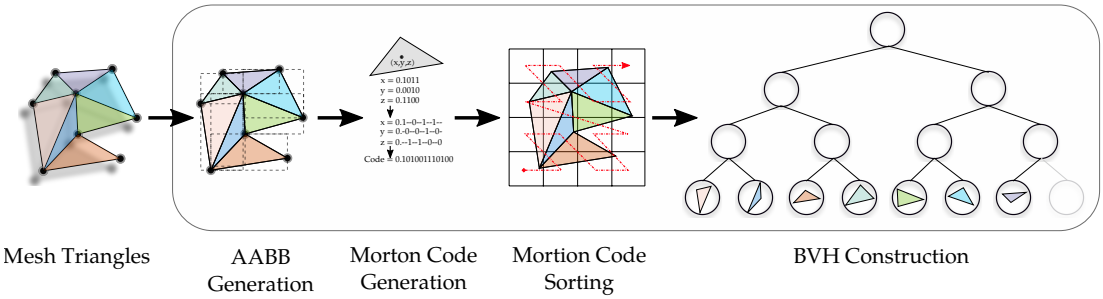


Figure 3.5: *BVH construction pipeline.*

The summary of the construction process of the hierarchy is shown in Fig. 3.5. The objects (triangles in our case) are assigned to the leaf nodes, their bounding volumes are computed then and Morton codes are computed based on their centre's 3D positions such that spatially adjacent nodes are given closer codes on the Z-curve. Leaf nodes are then sorted using the corresponding Morton codes as in [83]. Next, we walk up the tree one level at a time processing internal nodes until reaching the root. We have used the parallelisation strategy by Karras [69] but extended to further maximise parallelism and guarantee optimal usage of local shared memory while ensuring $O(n)$ complexity (thanks to the implicit layout). We now describe this process.

3.5.2 GPU Kernel Implementation

Given the sequence of objects sorted according to their Morton code, we construct the BVH while saving only bounding volumes to memory.

Algorithm Steps

Algorithm (3) provides a general outline to construct an ostensibly-implicit BVH (multi-kernel version illustrated in Fig. 3.6). We assume that internal-nodes and leaf-nodes are stored separately since leaf bounding boxes are already computed during Morton code evaluation. Threads start from a unique node on the *construction entry-level*, which is the first level processed when the kernel starts - executing as many threads as there are real nodes on this entry-level. Each thread then walks up the tree computing the parent node, and memory location as described in Eq. (3.9) (see also: lines 17 and 18). Additional indexing parameters, such as relative positions and sub-tree level, are inferred directly using implicit indices and thread IDs.

A group of threads is mapped to a sub-tree which is processed independently from the rest (line 1). A subtree is assigned to a group based on the given size and ID of the group thanks to the implicit layout (see Fig. 3.6). A group is defined by mapping threads to nodes of a subtree on the entry-level (lines 4 to 11) before proceeding to iteratively compute bounding volumes at higher levels (lines 16 to 26). When the entry-level is the second-last level of the tree, a thread will process its node by reading the array of leaf bounding volumes using the sorted triangle-IDs at relative positions determined by the thread global-ID. Otherwise, the thread will access the bounding volumes of the left and right child (which are internal nodes) in order to process current node. For operations that are localised to a group of threads, we also utilise local shared memory to effectively cache the computed bounding volumes - permitting fast access when processing the next level, which is guaranteed until the subtree root node is processed.

Each internal node is processed by exactly one thread by using atomic operations (line 19) to synchronise bounding volume updates. Threads are terminated if they are first to reach a node which is not on the entry level and has a right child - otherwise they remain active. The active thread will proceed to evaluate this node and continue until termination or reaching the root of the subtree.

Algorithm 3: High-level ostensibly-implicit BVH construction

Input : tIntArr - internal-node bounding-box array
Input : tEntryLev - level from which to begin aggregation
Input : meshFaceCount - number of faces in input mesh
Input : tLeafArr - mesh-order real-leaf bounding-box array
Output: tIntArr

```
1  parallelfor foreach group do
2      tLevPos = global_id // ∈ [0,t)
3      tNode = (2^tEntryLev − 1) + tLevPos
4      if tEntryLev == tLeafLev - 1 then
5          lBB = get_leftchild_bbox(tLeafArr, tNode, ...)
6          if rightChildIsReal then
7              rBB = get_rightchild_bbox(tLeafArr, tNode, ...)
8      else
9          lBB = get_leftchild_bbox(tIntArr, tNode, ...)
10         if rightChildIsReal then
11             rBB = get_rightchild_bbox(tIntArr, tNode, ...)
12     tNodeBB = merge(lBB, rBB)
13     write_bounding_box(tIntArr, tNodeBB, ...)
14     tLevMin = tEntryLev − log_2(group_size)
15     tLev = tEntryLev
16     while tLev ≥ tLevMin do
17         tLevPos = global_id/2^(tEntryLev-tLev)
18         tNode = (2^tLev − 1) + tLevPos
19         if rightChildReal AND firstThreadToReach(tNode) then
20             terminate()
21         lBB = get_left_child_bv(tArr, tNode, ...)
22         if rightChildReal then
23             rBB = get_right_child_bv(tIntArr, tNode, ...)
24         tNodeBB = merge(lBB, rBB)
25         write_bounding_box(tIntArr, tNodeBB, ...)
26         tLev = tLev - 1
```

Implementation

We propose two implementations for our construction algorithm distinguished by how they synchronise threads using GPU global memory. The first is the multi-kernel implementation following a bulk-synchronous parallel (BSP) approach [95] to synchronise threads using *only* local atomic operations when processing nodes. Global barriers (e.g. multiple kernel launches) synchronise thread-groups by unifying communication and storage after the sub-tree root is processed as shown in Fig. 3.6. This approach favours building large trees (e.g. more than 2^{17} triangles as shown in Fig. 3.10). The second implementation is single-kernel construction (similar to Karras [69]) which also uses one group-thread to update the sub-tree root node. However, nodes above the sub-tree are processed using global atomic operations to synchronise threads from different groups to build the entire tree in one kernel. Single-kernel construction is to be most useful with relatively smaller meshes where the overhead of global atomics is negligible due to having less demanding parallel workloads in terms of global memory accesses.

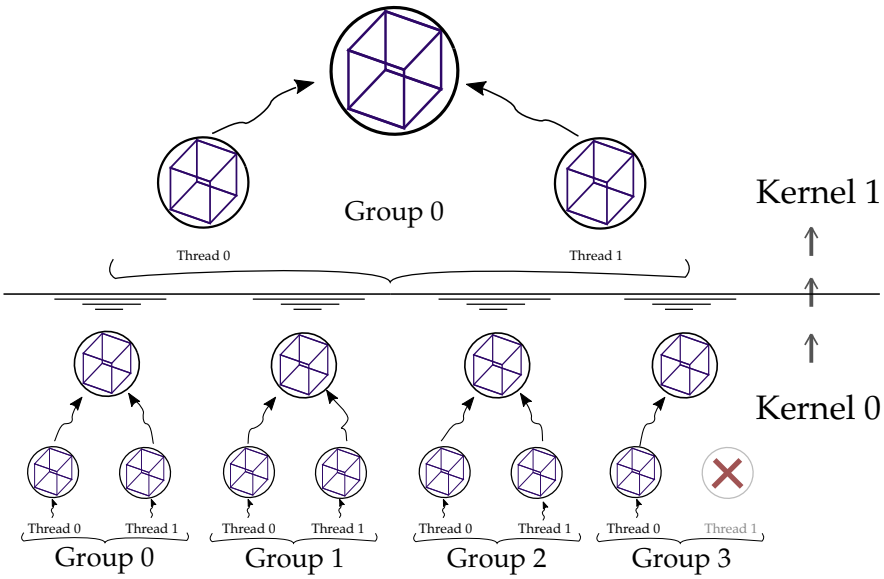


Figure 3.6: *Parallel tree construction where groups of threads are mapped to independent sub-trees.*

GPU Scheduling

By knowing the maximum size of a group, the height of the subtree in each kernel can be determined, and thus we can compute the total number of kernels to schedule.

Given the total number of real leaf nodes and the preferred number of threads in a group, we show that the remaining parameters needed to run multi-kernel construction can be pre-computed for seamless batch scheduling. These *scheduling parameters* are computed by

$$r_{k+1} = \left\lceil \frac{r_k}{g_k} \right\rceil \tag{3.11}$$

$$t_{k+1} = \frac{t_k}{g_k} \tag{3.12}$$

$$g_{k+1} = \begin{cases} g_k & \text{if} \quad g_k \leq t_{k+1} \\ 2^{\lfloor \log_2(r_{k+1}) \rfloor} & \text{otherwise.} \end{cases} \tag{3.13}$$

For each kernel k: r_k is the total number of real nodes at the corresponding entry level; t_k is the total number of threads; and g_k is the number of threads per group.

Initial parameter values are set either by the user or depending on configurations. As we assume that each leaf node stores one triangle, r_1 is the number of triangles in the BVH being constructed. Next, $g_1 = \min(g_{\texttt{user}}, L_c)$ is set by the user, and with the condition that $g_{\texttt{user}}$ is a power of two. Our condition on g_1 ensures that we can calculate $t_1 = g_1 \left\lceil \frac{r_1}{g_1} \right\rceil$ to allow seamless mapping of thread-groups to subtrees which have leaves whose total is a power of two.

As a result of this seamless mapping (and thanks to the implicit layout), the total number of kernels can be reduced by an exponential factor. Fig. 3.7 shows the number of kernels as a function of the initial group size g_1. This ability for control is useful in performance tuning: a reduction in the number of kernels will not necessarily lead to an improvement in performance gain but offers the ability to adapt to hardware. For example, in our experiments we found that the multi-kernel algorithm was particularly fastest when using either $g_{\texttt{user}} = 2^5$ or $g_{\texttt{user}} = 2^6$ threads per group.

3.5.3 Summary

As seen in this section, the GPU thread will require only the implicit index of the node to determine a path to the root thanks to the implicit layout. Further, our approach guarantees all synchronisation between threads in a group to be done using only local atomics which will reduce overhead. In particular, all the memory locations are directly determined from the implicit representation. In contrast, state-of-the-art methods [6, 42, 69, 83, 112] require tracking radix-key ranges as a part of the bottom-up reduction and using them to deduce the index of parent nodes. This requires additional memory

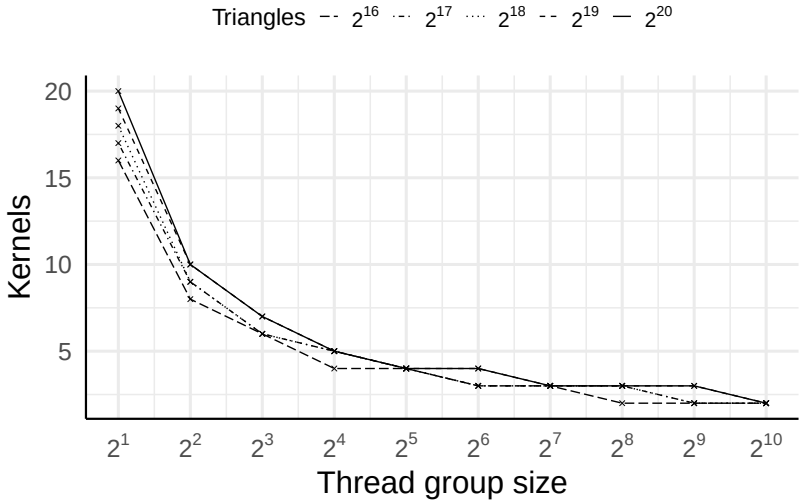


Figure 3.7: *Effect of the number of threads in a group on the number of BVH construction kernels.*

accesses which inevitably leads to lower performance as our experimental results will show.

3.6 BVH Traversal for Collision Detection

Having described how we represent and construct a BVH in Section 3.4 and Section 3.5, in this section we describe extensions to the traversal method described in Chapter 2. These extension are used to then form a complete pipeline for detecting collisions between interaction objects which is shown to outperform the state of the art (Section 3.7).

Thus, as a target application, we describe how our data structure can assist parallel collision detection. In contrast to *refitting* approaches, where one must forego full BVH maintenance to enable collision-front tracking, our approach allows one to maintain up-to-date BVHs of a given scene, knowing that the underlying polygons will be sufficiently captured and at minimal cost. This *broad-phase* collision detection is particularly important since it serves to cull the search space of costly polygon intersection tests.

Chapter 2 (Chitalu et al. [19]) described a simple but fast method for simultaneously traversing multiple BVHs on GPUs. In this approach BVH data is accessed like the heap but extended to allow arbitrary jumps to descendants for maximising GPU workloads. Thus, we adopt that algorithm and extend it using the ostensibly-implicit

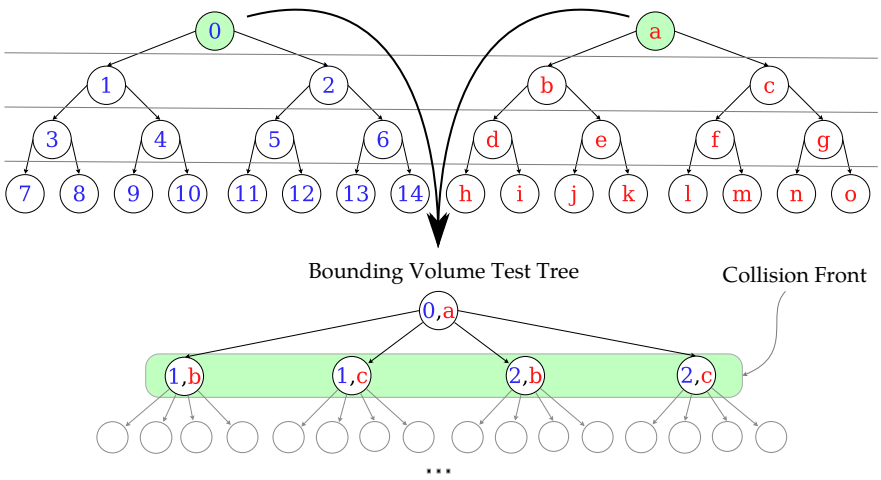


Figure 3.8: *Simultaneous BVH tree traversal. Pairwise collision detection is performed with multiple BVH-BVH tests at the same time producing one collision-tree.*

tree layout with further improvement to BVH traversal. Traversal is accelerated with the BSP model [95], decomposing the task into a series of iterative *level-synchronous* kernels (see Fig. 3.8).

Quick Access to Ancestors and Descendants

Quick access to ancestors and descendants are essential operations for BVH traversal. The implicit tree structure provides analytical solutions for accessing the ancestors, descendants and siblings in $O(1)$ time. See Chapter 2 and Chapter 4 for the details.

Expanding the BVTT

We perform explicit BVH-BVH tests and corresponding BVH levels (and henceforth, the BVTT) are explored before the next. In our representation, a BVTT node is a pair of integers which encode a BVH ID and an implicit index to form a node descriptor, so that the node data can be quickly accessed and tested for further intersections. Each kernel will map threads to unexplored parts of the resulting BVTT in an input queue.

The BVTT is managed within GPU global memory and used as the input for next kernel, which simplifies the operation as threads can be mapped to a small fixed number of work elements which are evaluated to produce new BVH node pairs that will be processed by the next kernel. Compared to front tracking [84, 138, 154], less data is marshalled in and out of global memory (Fig. 3.11) because the average BVTT sprouting size for each tested pair of BVH nodes is less than a factor of 2^{2n}, where n is the depth-step (jumping) parameter (Chapter 4). The resulting BVTT computation can be

done very efficiently even when starting from the root, and performs better than BVTT front tracking as shown in our experimental results (Section 3.7).

BVH Storage

In memory, multiple hierarchies are stored compactly in a consecutive manner as an array (similarly to what is described in Chapter 2). We use a memory layout that is similar to the well-known compressed sparse row (CSR) format consisting of two arrays termed *layout-arrays* [19]: `face_counts` keeps the number of primitives in each BVH tree and `offsets` keeps the starting offset of a BVH in memory. In contrast to Chapter 2, the new layout arrays are accessed using BVH IDs which are encoded within the node-descriptor of a collision-front node rather than being searched for at runtime (e.g. using binary search), where a descriptor is an element of the pair defining a BVTT node.

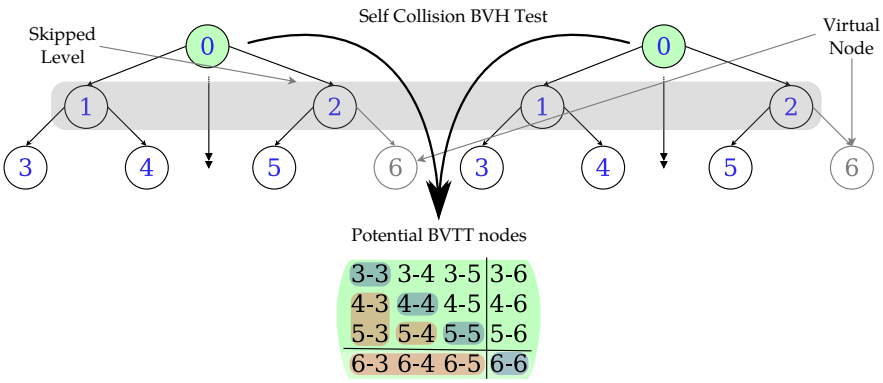


Figure 3.9: *Self-collision BVH test with depth-step $n = 2$. We compute valid BVTT nodes by accounting for from the number of virtual nodes. The number of new collision front nodes produced following a successful intersection amount to the same size as the upper/lower triangular entries.*

BVTT Node Expansion Size

The output size for an intersecting pair of BVH nodes is computed using the properties of a N×N index-matrix representing the nested for-loop over the descendants (see Fig. 3.9) which is simple and fast. We compute the *real* output size of this pair using the implicit index of each node, and the number of virtual nodes at that level (see e.g. Eq. (3.8)). Calculating the output size is useful when computing memory offsets, and filtering virtual nodes without explicit checks.

Handling Self-Intersection Tests

As pointed out by Wang *et al*. [154], an implicit race condition may arise during self-collision tests where a thread may produce duplicate workloads if no explicit ordering is specified between node-descriptors (e.g, checking 3-5 and 5-3 in Fig. 3.9). This race condition is generally benign but can affect performance due to work duplication. Ensuring that an explicit ordering is specified between node-descriptors is done by comparing the implicit indices of our nodes without extensive changes to the traversal algorithm thanks to the implicit layout.

Thus, determining the output size will amount to computing the number of edges in a fully connected graph by

$$\frac{N(N-1)}{2},$$

where N here denotes the number of real descendants of a node being tested against itself. [1]

3.7 Experiments and Results

In this section we present the results of our methods which are implemented using OpenCL with platform version "OpenCL 1.2 CUDA 9.1.84". Experiments are performed on a system with an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and an NVIDIA GeForce GTX 1080 @ 1733MHz equipped with 8GB of GDDR5X VRAM. We first evaluate the performance of our approach for fast BVH construction in Section Section 3.7.1. We then evaluate our method in collision detection scenarios and compare against the state-of-the-art in Section Section 3.7.2.

3.7.1 BVH Construction Performance and Comparison

We evaluate the performance of our construction algorithm where triangles are assumed to be already sorted. Where sorting is concerned (e.g. Section 3.7.2), we have used a publicly available bitonic-sort implementation [121] to sort Morton codes in our method: Each code is aliased as a 64-bit integer comprised of a 30-bit Morton code which occupies the 32 most significant bits, and its corresponding triangle ID.

Table 3.1 provides a breakdown of BVH construction time and compares against a well-known fast-construction method by Apetrei [6] which is implemented in CUDA.

[1] Note the importance of subtracting N from this fraction when the descendants are leaf nodes, since testing a leaf node against itself constitutes another redundant pair test.

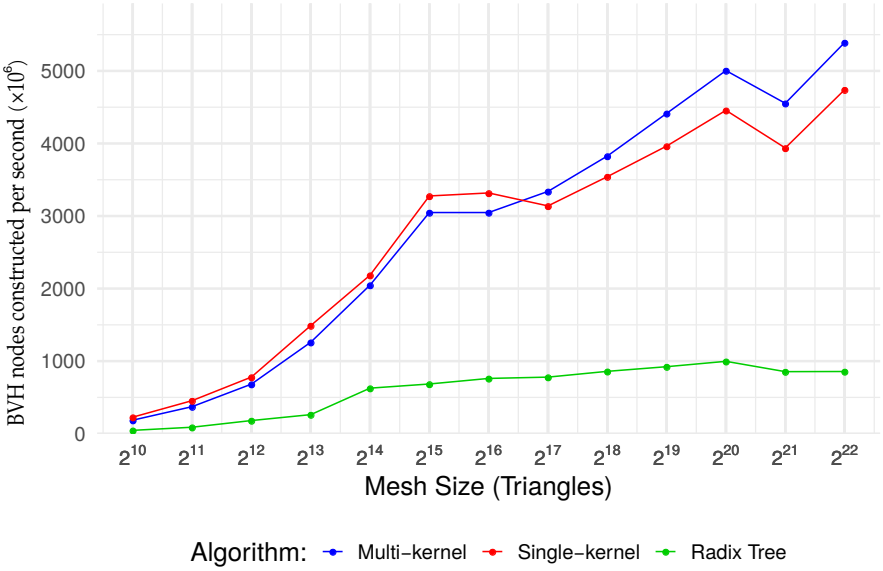


Figure 3.10: *BVH construction performance.*

	 Bunny 144k tris			 Erato 412k tris			 Dragon 873k tris			 Happy Buddha 1087k tris			 Hairball 2880k tris		
Builder	**Nodes**	**Padding** (%)	**Time** (ms)	**Nodes**	**Padding** (%)	**Time** (ms)	**Nodes**	**Padding** (%)	**Time** (ms)	**Nodes**	**Padding** (%)	**Time** (ms)	**Nodes**	**Padding** (%)	**Time** (ms)
Perfect Implicit BVH	524287	45.05	0.14	1048575	21.2	0.24	2097151	16.9	0.42	4194303	48.1	0.9	8388607	31.3	1.6
Apetrei [6]	288091	0	0.40	825337	0	1.10	1742611	0	2.29	2174947	0	2.92	5759999	0	8.50
OIBVH (Single-kernel)	288100	3.1e-3	0.074	825344	8.4e-4	0.25	1742621	5.7e-4	0.55	2174957	4.5e-4	0.83	5760004	8.6e-5	1.37
OIBVH (Multi-kernel)	288100	3.1e-3	0.077	825344	8.4e-4	0.23	1742621	5.7e-4	0.5	2174957	4.5e-4	0.7	5760004	8.6e-5	1.3
Speedup: Oi-BVH vs. Apetrei	-	-	5.4×	-	-	4.78×	-	-	4.58×	-	-	4.17×	-	-	6.53×

Table 3.1: *Results for BVH construction (time is in milliseconds), with a comparison between our two proposed implementations (single-kernel and multi-kernel) and the radix-tree BVH by Apetrei* [6]. *Speedup is calculated using our fastest algorithm over Apetrei* [6]*'s total time to compute the hierarchy topology and fit bounding boxes. Models are sourced from the McGuire Computer Graphics Archive* [92].

Our construction algorithms are faster than Apetrei [6]. At best, we achieve over 6.5× speedup over this state-of-the-art method, and averaging 5× across the evaluated datasets. At worst, we achieve 4.17× speedup on the Happy Buddha mesh dataset, which is significant given that this is our lowest score. Thus, our proposed method maps well to GPUs, offering a simpler and faster alternative for categorically fast BVH construction.

A comparison is also made against the naïve perfect implicit binary-tree BVH which has padded nodes. Although the performance is similar, our new layout saves up-to 48.1% of memory on the evaluated datasets which is significant because real-world meshes can require large BVHs where the size is just a few nodes off on the short side from being full. We conceivably occur some overhead during node index translation but it is reasonable to assume that this impact is negligible: Calculation requires only a few arithmetic instructions (plus e.g. `count_set_bits`), and without additional reads from a table. Thus, our approach is efficient by storing an optimal number of nodes and with minimal overhead.

Performance Scaling

The overall scaling of our BVH construction performance in terms of BVH nodes constructed per second is shown in Fig. 3.10. We analysed scalability by evaluating construction time using a gradually refined mesh, from 2^{10} to 2^{22} triangles. Our construction algorithms yield high throughputs, reaching a rate of at-least 4.7 billion BVH nodes per second. This experiment also reveals that ostensibly-implicit tree construction scales optimally and retains faster execution time than Apetrei [6] with 3.2–6.2× speedup (averaging 4.5×). Our speedup is highest with large meshes, where the number of threads is sufficient to saturate the hardware.

In relative terms, our construction algorithms yield competitive performance where the multi-kernel version is approximately 7% faster than the single-kernel implementation. Fig. 3.10 reveals that a crossing-point in throughput between the two algorithms is reached when using a mesh with approximately 2^{17} triangles. Our single-kernel algorithm is 1.15× faster below this threshold, but saturates the hardware with lower throughput due to the reliance on global atomics which increase with the number of triangles. The multi-kernel version is 1.22× faster above the threshold since local atomics amortise the cost of global barrier synchronisation.

3.7.2 Collision Detection Performance Comparison

In this section, we compare our method to two other techniques for handling collision detection - including broad-phase and narrow-phase. We show that our approach is faster across a number of benchmarks. Comparisons on worst-case runtime memory usage are also discussed. In all results shown, our BVHs are re-built from scratch at every frame.

Comparison against Wang *et al.* [154]

Table 3.2 summarises our collision detection performance using datasets from the UNC dynamics benchmarking suite [25], and compares with Wang *et al*. [154] (see also Fig. 3.12).

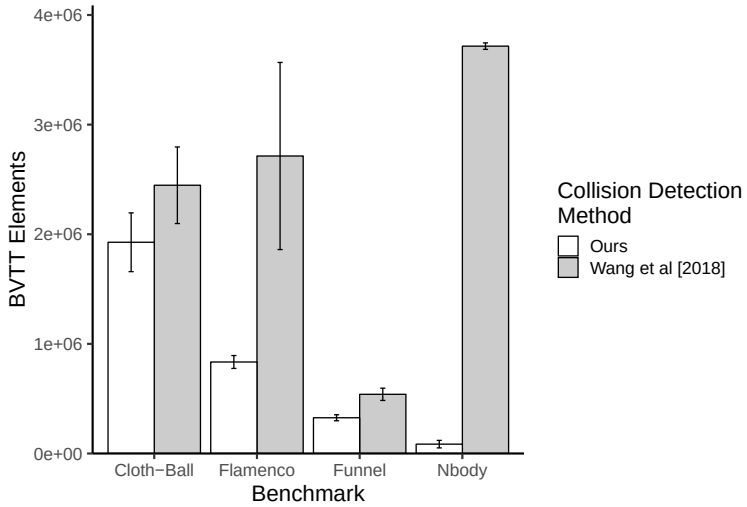


Figure 3.11: *Average BVTT size across frames.*

Speed: We compare against their speed using BVH refitting and front tracking. Our method is up to 30% faster which is significant since we reconstruct our BVHs from scratch at every frame (Note: our speedup is $4\times$ when they re-build BVHs every frame). At best the state-of-the-art BVH based techniques build the BVH once and simply refit at every frame, which degrades the overall efficiency over time. With our approach however, not only it is feasible to rebuild the BVH at every frame, but our technique is actually faster than refitting.

Memory: When performing broad-phase collision detection, front-tracking will explicitly cache BVH node pairs where traversal stops - managing such a scheme con-

				Wang *et al.* [154]		*Ours*		Comparison	
Benchmark	**Triangles**	**Objects**	**Frames**	**Time** (stdev)	**Runtime Memory** (mb)	**Time** (stdev)	**Runtime Memory** (mb)	**Performance Speedup**	**Memory Usage** (%)
Cloth-Ball	92k	2	94	2.3 (1.33)	145.46	1.9 (0.34)	85	1.2×	58.4
Funnel	19k	4	500	0.6 (0.16)	58.6	0.6 (0.06)	8.53	1×	14.5
N-Body	142k	305	75	4.3 (0.24)	296.85	3.62 (0.07)	15.1	1.18×	5
Flamenco	49k	10	705	2.59 (0.52)	130.37	2 (0.1)	24.2	1.29×	18.57

Table 3.2: *Performance comparison of our collision detection (broad + narrow phase) with Wang* et al. [154].

.

sumes a lot of memory, especially when interaction between the objects are intense. Fig. 3.11 shows a comparison of average BVTT size against Wang *et al*. [154] for the same benchmarks used in Table 3.2. Our approach can reduce the BVTT size by up-to 97.7% (see: N-Body benchmark), making our approach simpler, faster and more memory efficient.

Pipeline analysis

In general, BVH construction and traversal, which are the focus of this chapter, occupy most of the execution pipeline. Fig. 3.12 shows a breakdown of total collision detection time for the results presented in Table 3.2. We execute the full BVH construction pipeline on datasets with self-collisions (MC Eval, MC Sort and Build) which takes 28-40% of the total time. With the N-body dataset, BVH construction accounts for over 95% of the total execution time surmounting to 3.4ms. While the individual rigid-bodies are small (approximately 1024 triangles per mesh), the quantity leads to an overall degradation in performance because BVHs are constructed sequentially. Nonetheless, the total execution time for N-Body is below 4ms which is faster than Wang *et al*. [154]. BVH Traversal (broad-phase) accounts for approximately 48-52% of the execution time on the self-collision datasets leading to large workloads arising from the tested BVH node-pairs. For N-Body, traversal accounts for approximately 5% of the total time because it is a rigid body simulation (no self-collisions) and mesh sizes are small. In general, traversal is largely affected by mesh configurations in each frame relative to the density of the dataset under consideration. On the other hand, the cost of construction is largely dependent on the number of meshes.

Comparison against I-Cloth [142]

Table 3.3 presents the results of our method applied to larger datasets from the I-Cloth benchmark suit by Tang *et al*. [142] (see Fig. 3.13). Our method is on average 59.8× faster than I-Cloth, achieving up to 97.7× on the largest dataset which is Bridson-3, with 198k triangles. We compare against their publicly available CUDA source code with measured average timings per-frame. These measurements are obtained with `nvprof` and `nvidia-smi` tools, where we compare specifically against their collision detection kernels and without including the execution time for resolving collisions. Table 3.3 shows a comparison of runtime memory costs with I-Cloth. We compared against memory figures which are a 25% proportion of the values reported by the

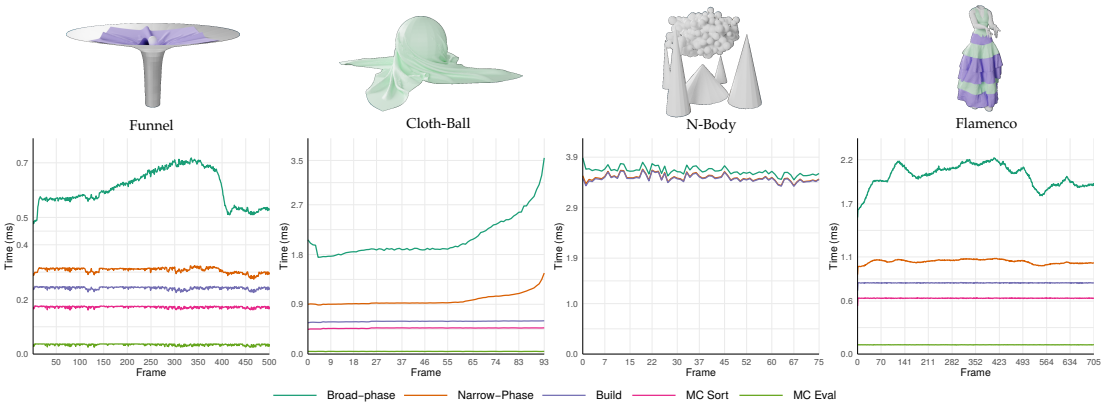
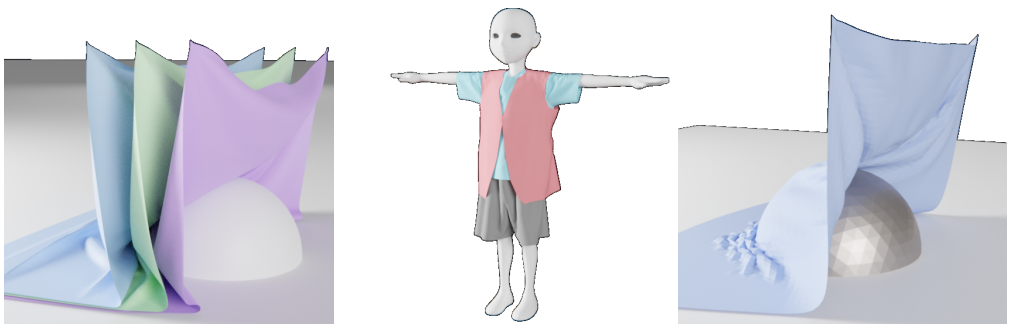


Figure 3.12: *Collision detection time using datasets from the UNC dynamics benchmarking suite* [25]. *Full BVH construction occurs every frame for the datasets with self-collisions (Funnel, Cloth-Ball and Flamenco) by including Morton-code evaluation (MC Eval) and sorting (MC Sort). We perform only refitting (Build) with N-body for all 305 objects because it is rigid-body dataset. We have used the same implementation as gProximity* [84] *for narrow-phase collision detection which is also adopted by Wang* et al. [154]. *The Broad-phase component highlights our simultaneous BVH traversal execution time to find the set of potentially colliding pairs which are forwarded onto the narrow-phase.*

`nvidia-smi` tool, making our measurements estimations due to source code access restrictions. Our method performs collision detection using less than 10% memory relative to I-Cloth (actual GPU RAM used). [2]



(a) Bridson-3 (b) Andy (c) Bridson

Figure 3.13: *I-Cloth benchmarks (source:* [142]*)*

[2]The I-Cloth source-code simply provides a `C++` interface to pre-compiled libraries.

				I-Cloth [142]		*Ours*		Comparison	
Benchmark	**Triangles**	**Objects**	**Frames**	**Time**	**Runtime Memory** (mb)	**Time** (stdev)	**Runtime Memory** (mb)	**Perf' Speedup**	**Mem' Reduction** (%)
Andy	127k	4	80	121.58	498.5	2.65 (0.02)	41.16	45.87×	91.7
Bridson	18k	2	867	17.93	456.25	0.5 (0.02)	6.62	35.86 ×	98.54
Bridson-3	198k	4	842	392.97	534.75	4.02 (0.17)	85.4	97.7×	84.02

Table 3.3: *I-Cloth* [142] *benchmark performance (see also Fig. 3.13).*

.

3.8 Discussion

The implicit tree is a technique used in computer graphics which is usually ideal when a BVH is perfectly balanced or the node payload size is relatively small. We have presented an adaptation of the implicit tree for collision detection in computer graphics. Our approach improves the generality, efficiency and scalability of classical implicit tree structures through a novel encoding of their structure using simple bitwise manipulations. With this adaptation, BVH traversal and construction are performed while assuming an implicitly defined structure, but we store nodes compactly in memory, and without post-processing (where the storage cost scales linearly with the number of objects). We demonstrated the advantages by comparing against the state-of-the-art for GPU-based collision detection. We also presented a fast construction algorithm which when combined with our simple collision detection pipeline enabled a decoupling of performance from BVTT front tracking and BVH refitting. Consequently, our pipeline is able to perform collision detection in a reasonably short time - with BVH construction occurring every frame.

The new layout will unfortunately require a small number of redundant nodes to retain the benefits of implicit trees. The layout requires 'support' nodes with only one child (e.g. node 12, Fig. 3.3) to maintain the implicit structure, which is a side-effect of moving virtual nodes to the right, and placing all leaves at the lowest level. However, in the worst-case (i.e. $t - 2^{\lfloor \log_2 t \rfloor} = 1$), storage costs are only approximately $N_r = N_c \times (.5 + \varepsilon)$, where

$$\varepsilon = \frac{\log_2 t}{N_c} \equiv \frac{\texttt{count\_set\_bits}(L_v)}{N_c} \tag{3.14}$$

accounts for the number of support nodes. Thus, our ostensibly-implicit layout is most efficient when t is just a few increments off on the short side from creating a full tree, reducing memory costs by up-to approximately 50% (classic padding will allocate space for N_c nodes). Moreover, the new layout guarantees that the storage costs scale linearly in t (*cf.* Fig. 3.14) to overcome the prohibitive storage costs of fully padded implicit trees.

3.8.1 Limitations & Further Work

The new layout, as currently defined, is constrained to labelling nodes in breadth-first search (BFS) order, meaning that alternative depth-first search (DFS) optimisations cannot be applied during traversal which we believe could be beneficial for potential

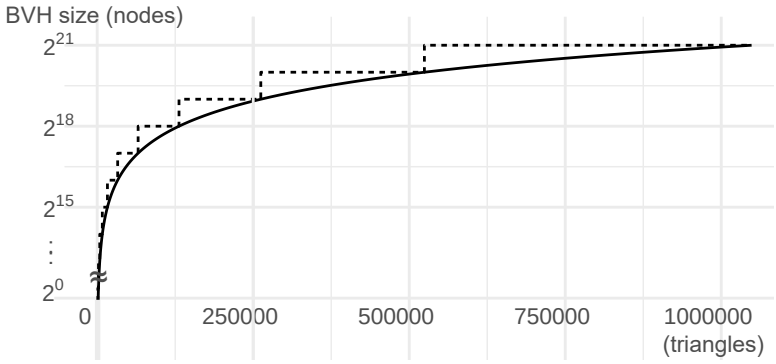


Figure 3.14: *A plot of the number of objects (triangles) vs the number of BVH nodes (N_r). The solid line represents the new layout, and the dashed line is a perfect-implicit layout (i.e. with padding).*

applications such as Ray-tracing. BFS is representative of parallel tasks with an irregular and data-dependent mode of execution, but it may not be suitable for all such applications. For this reason, an interesting future direction is to reformat/remap the current layout into DFS form.

As a categorically "fast-construction" approach, our technique emphasizes build performance and simplicity which can limit BVH quality by a considerable amount in some cases. We achieve performance by simply pairing nodes at each level and without looking at the actual radix bit values to construct the hierarchy. This is indeed fast for tree construction, but fails to account for adjacent pairs of triangles in a sorted list that are spatially far apart. The alternative LBVH [6] produces trees which consider this spatial adjacency, and thus the quality is better as presented in Table 3.4. We also conduct an experiment where we gradually move one triangle away from the original mesh (see Fig. 3.15) to examine how the quality of the BVHs by our method and those by LBVH varies with respect to the distance of the separate triangle (from 0 to 10 times the model diameter). The ratio of the SAH by the two methods are plotted in Fig. 3.16. As expected, the SAH cost of our method grows much faster compared to the LBVH, which reveals the weakness of our approach. Therefore, we trade BVH quality for simplicity and construction performance, which is fast but requires further consideration for building good quality trees.

Like many prior methods tackling the BVH construction problem, our algorithms do not currently handle simultaneous constructions - in particular if an application includes many mesh-BVHs. Thus, simultaneously building multiple BVHs may help this issue by using one generalised monolithic kernel.

In order to perform traversal, we repeatedly step through BVHs from the root to

Scene (#tris)	LBVH ([6])	Oi-BVH (ours)	SAH Cost Increase
Happy Buddha (1087k)	86	229.16	-2.66×
Hairball (2880k)	669.8	1122.34	-1.67×
Dragon (873k)	77.43	201.54	-2.6×

Table 3.4: *Comparison of surface area heuristic (SAH) with the LBVH* [6]. *We compute SAH using Eq. 1 in* [2].

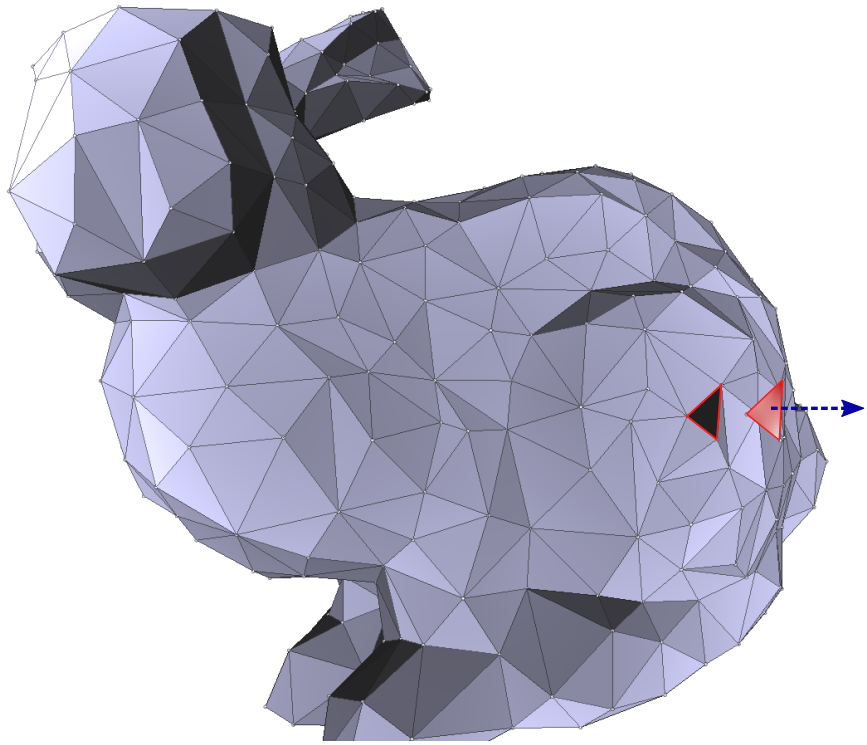


Figure 3.15: *Fig. 3.16 experiment setup: We move a single triangle away from the rest of a given mesh by a multiple of the bounding box diagonal in the normal direction.*

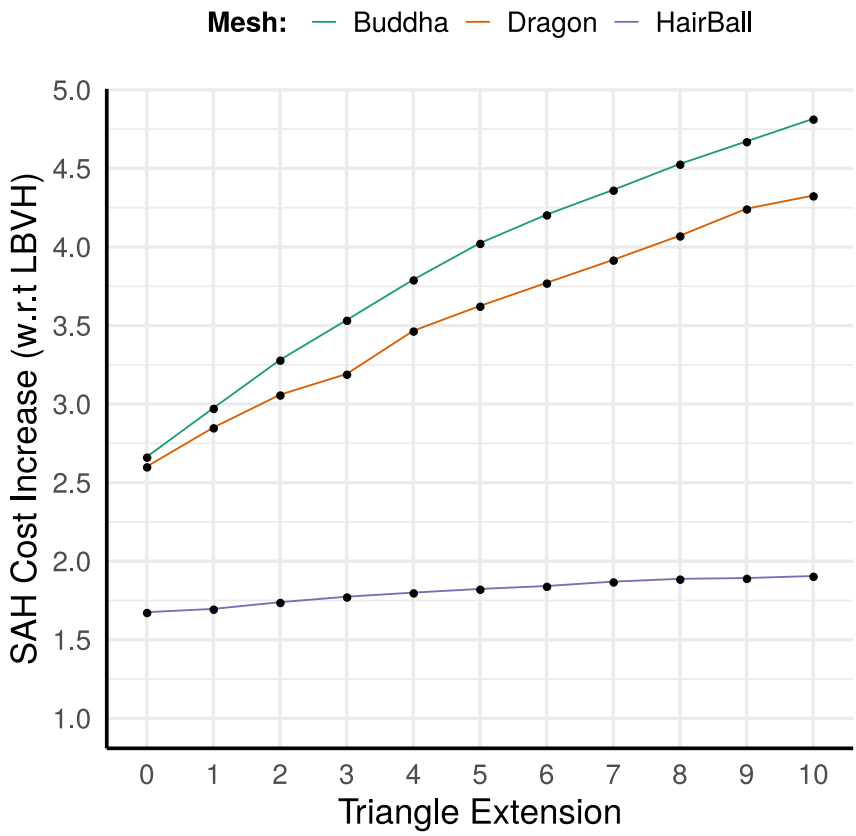


Figure 3.16: *A plot of the SAH cost increase for our technique which is calculated relative to the LBVH* [6] *(see also Table 3.4). Our setup is illustrated in Fig. 3.15, where the x-axis represents a triangle's offset multiple as it is moved away from the rest of the mesh.*

the leaves each frame, resulting in highly redundant intersection tests which can be mitigated by utilising temporal coherence. One simple solution is integrating classical front-tracking with deferred fronts to minimise the memory footprint [138, 141], while accounting for the depth-step and the traversal entry-level.

Also, current BVH traversal performance (e.g. with Cloth-Ball) is tightly coupled with our ability to re-build BVHs at every frame to minimise falsely-positive overlaps during node intersection tests. Optimisations such as parallel-scan and stream-compaction methods would facilitate retaining the complimentary benefits of irregular and/or timely but infrequent re-builds by reducing memory access overhead. This can then be accompanied by simpler refitting or local restructuring strategies to improve overall performance. Finally, normal cone tests [153] could also be used for reducing node-intersection tests but it is unknown whether a net-gain in performance is guaranteed.

3.9 Postscript

The framework described in this chapter can be used to perform large scale BVH traversal in collision detection and requires no tracking of radix-key ranges from Morton codes. A number of extensions are certainly possible in this work, including using trees with a larger arity (Appendix B), or more complex distributions of triangles inside the tree to account for quality and runtime costs.

While the data layout we developed in this chapter is lightweight, its design is particularly appropriate in collision detection environments because it is simple and very fast to build - there is no need for tracking radix key ranges. In addition, the memory costs are minimal - only node bounding volumes need to be stored in memory. The ostensibly-implicit layout opens avenues for further work on implicit trees which can bring benefits across the field of computer graphics - especially for other prominent areas like ray-tracing.

Chapter 4

Fast Indexing of Implicit Trees

This chapter will briefly present a set of formulas describing fast tree-indexing rules which are used in Chapter 2 and Chapter 3. The formulation is based on recursive geometric-sequencing which we use for enabling fast access to ancestor and descendant nodes in $O(1)$ time. The formulation is particularly useful during tree construction and traversal.

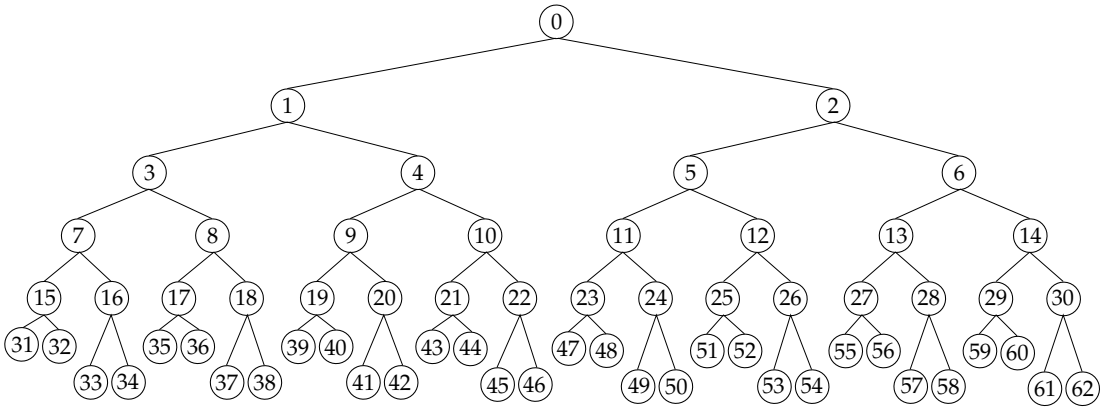


Figure 4.1: *A complete binary tree of n nodes which is stored into an array of N elements by mapping its nodes in a breadth-first level-by-level manner.*

We assume that a tree is stored into an array with nodes labelled in a breadth-first level-by-level (pre-order traversal) manner (*cf.* Fig. 4.1). Given this setup it is easy to show that, the immediate relatives of a node with an implicit index i, $(0 \leq i)$ are computed by

$$parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor \tag{4.1}$$

$$child(i,j) = 2i + j, \qquad 1 \leq j \leq 2, \tag{4.2}$$

which is also known as Eytzinger's method - a genealogical numbering system. Thus, it is possible to completely remove all pointers to store only the payload (data) of each node [35]. However, the formulae in Eq. (4.1) and Eq. (4.2) are limited to accessing only the neighbouring nodes which is restrictive if we wish 'jump' to any node(s) that we desire.

4.1 Descendants

We now describe the derivation of functions to find any descendant or ancestor of a node that is n levels from this node.

The n-th generation leftmost descendant of a node i in a complete κ-ary tree can be calculated using just index information (e.g. node 39 as the 2^{nd} generation descendant of node 9 in Fig. 4.1). To show this, we start with the implicit binary tree as presented in Fig. 4.1, where $\kappa = 2$. For this implicit binary tree, a function $f_n(i)$ is defined with the property that

$$f_0(i) = i \tag{4.3}$$

$$f_{n+1}(i) = 2f_n(i) + 1, \tag{4.4}$$

to determine the n-th generation leftmost descendant of a node i. The function $f_n(i)$ is given by the following conjecture

$$f_n(i) = 2^n i + 2^n - 1, \tag{4.5}$$

to describe the node-to-leftmost-descendant relationship in the implicit binary tree. This can be verified by using the recurrence relation in Eq. (4.4), to give

$$\begin{aligned} f_{n+1}(i) &= 2\left(2^n i + 2^n - 1\right) + 1, \\ &= 2^{n+1} i + 2^{n+1} - 2 + 1, \\ &= 2^{n+1} i + 2^{n+1} - 1, \end{aligned} \tag{4.6}$$

which is the solution used in Eq. (2.1) and the methods described in Chapter 3.

4.1.1 Beyond Binary Trees

Generalising Eq. (4.4) for $\kappa \geq 2$, we have

$$f_{n+1}(i) = \kappa f_n(i) + 1, \tag{4.7}$$

in order to account for any degree κ (e.g. ternary-, quad-, oc-tree etc.). The generalisation of $f_n(i)$ in Eq. (4.5) is then given by

$$f_n(i) = \kappa^n i + \alpha\kappa^n + \beta, \tag{4.8}$$

to describe the n-th generation leftmost descendant of node i in a κ-ary tree, where α and β are necessary coefficients to be determined. (Note that Eq. (4.8) is derived based on the intuition that it must follow a similar pattern as Eq. (4.5)). Thus, the next step is to determine the coefficients, which we find by using next recurrence of Eq. (4.8) (*cf.* Eq. (4.6))

$$\begin{aligned} f_{n+1}(i) &= \kappa(\kappa^n i + \alpha\kappa^n + \beta) + 1, &(4.9)\\ &= \kappa^{n+1} i + \alpha\kappa^{n+1} + \beta\kappa + 1. &(4.10) \end{aligned}$$

We find β and α from Eq. (4.10) using substitution: First, β is determined by

$$\begin{aligned} \beta &= \frac{1}{1-\kappa} \\ &\equiv -\frac{1}{\kappa - 1}, \end{aligned} \tag{4.11}$$

since Eq. (4.10) is the recurrence of Eq. (4.8), which implies

$$\beta\kappa + 1 = \beta. \tag{4.12}$$

Thus, we derive Eq. (4.11) from Eq. (4.12).

Proof.

$$\begin{aligned} \beta\kappa + 1 &= \beta \\ \beta\kappa - \beta &= -1 \\ \beta(\kappa - 1) &= -1 \\ \beta &= -\frac{1}{\kappa - 1}. \end{aligned} \tag{4.13}$$

□

Having found β, we then subtitute into Eq. (4.8) to arrive at

$$f_n(i) = \kappa^n i + \alpha\kappa^n - \frac{1}{\kappa - 1}, \tag{4.14}$$

which is our intermediate step to finding the full expression for $f_n(i)$ (Eq. (4.8)). With β, we determine α from Eq. (4.14) as

$$\alpha = \frac{1}{\kappa - 1}. \tag{4.15}$$

Proof. Using Eq. (4.14), let $i = 0$ and $n = 1$. Thus,

$$\begin{aligned} 1 &= \kappa^1 0 + \alpha\kappa^1 - \frac{1}{\kappa - 1} \\ 1 &= \alpha\kappa - \frac{1}{\kappa - 1} \\ 1 + \frac{1}{\kappa - 1} &= \alpha\kappa \\ \frac{\kappa - 1}{\kappa - 1} + \frac{1}{\kappa - 1} &= \alpha\kappa \\ \frac{\kappa}{\kappa - 1} &= \alpha\kappa \\ \frac{\kappa}{\kappa^2 - 1} &= \alpha \\ \frac{1}{\kappa - 1} &= \alpha. \end{aligned} \tag{4.16}$$

□

It follows then that the solution to compute the n-th generation leftmost descendant of a node i in a κ-ary tree (*cf.* Eq. (4.8)) is

$$f_n(i) = \kappa^n i + \frac{\kappa^n}{\kappa - 1} - \frac{1}{\kappa - 1}. \tag{4.17}$$

Once again, we verify Eq. (4.17) using the recurrence relation in Eq. (4.7) with $n = 1$ to give

$$f_{n+1}(i) = \kappa^{n+1} i + \frac{\kappa^{n+1}}{\kappa - 1} - \frac{\kappa}{\kappa - 1} + 1 \tag{4.18}$$

$$= \kappa^{n+1} i + \frac{\kappa^{n+1}}{\kappa - 1} - \frac{1}{\kappa - 1}, \tag{4.19}$$

which has equivalent form to Eq. (4.17).

Going back to Eq. (4.2), this can now be generalised using Eq. (4.17) to compute any n-th generation descendant of node i in a κ-ary tree by

$$descendant(i, k, n) = f_n(i) + k, \qquad 0 \leq k < \kappa^n \tag{4.20}$$

where k is the relative position of an n-th generation descendant of node i.

4.2 Ancestors

Notably, determining the n-th generation ancestor of a node is equally useful since it enables a generalised solution to back-stepping (node collapse) within implicit trees without loop constructs.

Let $j = f_n(i)$ as in Eq. (4.17), which is any n-th generation leftmost descendant ($k = 0$) of a node i. It is straightforward to determine i given j, which is done by conversion

$$\begin{aligned}
j &= \kappa^n i + \frac{\kappa^n}{\kappa-1} - \frac{1}{\kappa-1}, \\
j(\kappa-1) &= i\kappa^n(\kappa-1) + \kappa^n - 1, \\
j(\kappa-1)+1 &= \kappa^n\left[i(\kappa-1)+1\right], \\
\frac{j(\kappa-1)+1}{\kappa^n} &= i(\kappa-1)+1, \\
\frac{j(\kappa-1)+1}{\kappa^n} - 1 &= i(\kappa-1), \\
\frac{j(\kappa-1)+1}{\kappa^n} - \frac{\kappa^n}{\kappa^n} &= i(\kappa-1), \\
\frac{j(\kappa-1)+1}{\kappa^n(\kappa-1)} - \frac{\kappa^n}{\kappa^n(\kappa-1)} &= i, \\
\frac{j(\kappa-1)}{\kappa^n(\kappa-1)} + \frac{1}{\kappa^n(\kappa-1)} - \frac{1}{(\kappa-1)} &= i, \\
\frac{j}{\kappa^n} + \frac{1}{\kappa^n(\kappa-1)} - \frac{1}{(\kappa-1)} &= i.
\end{aligned} \tag{4.21}$$

In the common case of a binary tree ($\kappa = 2$), we have

$$\begin{aligned}
i &= \frac{j}{2^n} + \frac{1}{2^n} - 1 \\
&= \frac{1}{2^n}(j+1) - 1,
\end{aligned} \tag{4.22}$$

which is in fact the generalisation of Eq. (4.1).

Part II

Simulating Detected Physical Interactions

Chapter 5

Overview

This chapter will summarise the design choices made to produce a new method for simulating fracturing rigid bodies during physical interactions, and gives a brief overview for the remainder of this thesis.

More specifically, the next two chapters will present a so-called *remeshing-free* brittle fracture simulation method under the assumption of quasi-static linear elastic fracture mechanics (LEFM). Two algorithms are derived which are described in these two chapters. First, we develop an approximate volumetric simulation, based on the extended finite element method (XFEM), to initialise and propagate Lagrangian crack-fronts. The second algorithm defines a mesh cutting method, which is used for producing mesh fragments using the simulated fracture surface. Together, these algorithms comprise the brittle fracture simulation method which is described in the following two chapters.

We work with XFEM because it avoids volumetric remeshing (or refinement) procedures *when treating evolving cracks*, and it allows us to simulate on arbitrary domains which do not have particular constraints on the volume-to-surface-area ratio. A tetrahedral mesh is still used to represent the descrete domain because it makes for an easier implementation relative other mesh representations, and because there is now a mature set of mesh generation tools which are available for use. Moreover, since computer graphics applications traditionally use tetrahedral meshes to represent the physical domain, the required meshing techniques (i.e. by volume decomposition) are well known and thoroughly investigated.

In our implementation, we also choose an *explicit* XFEM as it allows us to immediately compute the enrichment data–particularly, Heaviside enrichments–without implicit representations for the locations of displacement discontinuities inside finite

elements. Thus, we do not use a level set method (or similar methods for representing material discontinuity): we instead aim to work directly on generated crack surface meshes and keep the resolution (polygons) of this mesh decoupled from the number of finite elements, where these level set methods become limited in practice. Consequently, we efficiently simulate crack propagation at a much higher resolution than the deformation, producing detailed and visually realistic fracture surfaces but without continuous re-meshing during crack propagation.

A simple Rankine criterion is also followed to model crack initiation, and with a Griffith criterion to model crack propagation for fracture simulation. We evaluate stress intensity factors (SIFs) by re-adapting a classic displacement correlation approach which is effective to determine both the speed and direction of crack growth. Thus, the number of degrees of freedom (DOF) is much lower than in standard XFEM because there is no requirement for crack tip enrichment which is efficient and further simplifies the method.

Finally, a quasi-static approach is used during fracture simulation, permitting larger time steps relative to fully dynamic methods: a rigid-body system handles large-scale dynamics. For simple scenes (e.g. fracture loading mode tests), we just run the fracture simulation with manually specified boundary conditions. For more complex scenes, we integrate our fracture method into the rigid-body system, such that a collision can cause an object to break, and the resulting fragments are added back into the rigid-body scene.

For our rendering meshes, we choose a high resolution surface representation, implemented via any standard surface mesh data structure. We store each fracture, as well as the object's rendering surface, on a separate mesh data structure during the fracture simulation. Afterwards, we intersect the rendering surface with the simulated crack surface mesh to determine how the object separates into individual fragments using our novel cutting algorithm.

For mesh cutting, we design an algorithm which operates directly on the half-edge data structures of two surface meshes. The method is general and thus enables for cutting arbitrary surface meshes including those of concave polyhedra and meshes with abutting concave polygons. The details of this cutting algorithm are described in Chapter 7

Chapter 6

Displacement-Correlated XFEM for Simulating Brittle Fracture

Figure 6.1: *Our method offers a high-resolution crack propagation scheme on an explicit surface mesh without the need for adaptively tetrahedralised input meshes. We achieve this by combining a novel adaptation of the extended finite element method (XFEM), and a polygon-based cutting algorithm to compute fragments which retain their characteristic ridge-like structures–and sharpness–due to cracks. The figure shows an impact between a statue of The Winged Victory of Samothrace and a wrecking ball.*

6.1 Preface

This chapter will present an efficient, scalable and controllable technique for physical simulation of the propagation of fracture in brittle objects. Specifically, we will describe an approximate volumetric simulation, based on the extended Finite Element Method (XFEM), to initialise and propagate *crack-fronts* within brittle objects. In this method, we model the geometry of fracture *explicitly* as a surface mesh, which allows us to generate high-resolution crack surfaces and to decouple the resolution of the fracture surface from the size of the linear system being solved.

6.2 Introduction

Realistic depiction of breaking objects is desirable across a range of computer graphics applications including special effects, computer animation and video games. The breakage of 3D models can be mimicked either manually via tedious artistic specification or using automatic algorithms. The latter can be classified into methods that exploit *heuristics* [101, 133] and those that compute physical simulations of the mechanics of fracture. Simulation methods typically have origins in engineering and are therefore designed to be accurate for targeted applications. Adapting them to suit computer graphics applications often requires overcoming challenges such as reducing computational cost and generalising to realistic boundary conditions [50] as well as providing controllability [18].

Despite the effort of computer graphics researchers, existing techniques still suffer from either scalability, stability or realism problems for simulating the propagation of crack surfaces on arbitrary shapes. Classic finite element method (FEM)-based approaches [107] require conforming the domain mesh to the crack surfaces by re-meshing, which poses several challenges when treating evolving fractures. To avoid the cost and instability caused by re-meshing, some methods operate on particle systems [116, 132, 162]. These meshless methods introduce difficulties with respect to enforcing essential boundary conditions, computational cost and overall rigidity of computed fragments. Boundary element method (BEM)-based methods use surface representations [49, 50, 166]. In contrast to other discretisation methods (e.g. FEM), these methods involve singular integrals which can be prohibitively expensive to solve and are restricted to materials characterised by large volumes. Also, they are not readily able to seed cracks on the interior due to their boundary integral formulation.

Extended finite element method (XFEM) [27, 30, 96] is proposed for decoupling the simulation mesh from the crack surface: however, prominent methods that represent the crack surfaces by level-sets require using high resolution simulation meshes for simulating detailed crack surfaces. Also, tracking the dynamic propagation of the crack surface by level sets is inherently difficult.

In this chapter, we present an efficient brittle fracture simulation method in high resolution and without any requirement on the simulation mesh. In this method, we combine XFEM with a high-resolution crack propagation scheme on an explicit surface mesh, resulting in an efficient framework for brittle fracture. Our method allows for handling crack propagation without re-meshing (changing the simulation mesh)

and crack-tip enrichment, which has several advantages including reducing the number of degrees of freedom (DOF). Using a volumetric setting, we can simulate brittle fracture on a broad range of domains and accommodate spatially varying material parameters to control fracture. To represent a crack we adopt an explicit approach which can simplify the procedure to propagating the crack surface within the volumetric domain.

We simulate results showing detailed and realistic fracture of multiple brittle objects colliding and breaking into small pieces (see Fig. 6.1, and Section 6.8). Our method also allows animators to control the breakage by biasing the toughness within the domain, and to control the distinctive look sought from real-world materials in a simple fashion.

6.2.1 Chapter Contributions

Fig. 6.2 provides a visual overview and comparison with existing approaches. The contribution of this chapter is a novel quasi-static brittle fracture simulation method on the basis of XFEM but without crack-tip enrichment. This method leverages basis enrichment to efficiently simulate the propagation of cracks without re-meshing the simulation mesh. Propagation is handled using displacement correlation by relating the computed nodal displacements with asymptotic near-field displacement-equations to determine fracture loading parameters. Thus, only one type of enrichment is required to capture strong discontinuities in the displacement fields, which is Heaviside enrichment. The result is an efficient crack propagation scheme using the computed loading parameters to grow fracture surfaces in a volumetric setting.

Organisation: The rest of the chapter proceeds as follows: After reviewing related work in Section 6.3, we proceed to describe the derivation of XFEM on the basis of FEM in Section 6.4. An overview of our method is then given Section 6.5 which is followed by the details of our XFEM system in Section 6.6. We describe the process by which we interface our XFEM system with a rigid body solver to compute tractions forces from collision impulses in Section 6.7, and present our experimental results in Section 6.8. Finally, we conclude the chapter in Section 6.9.

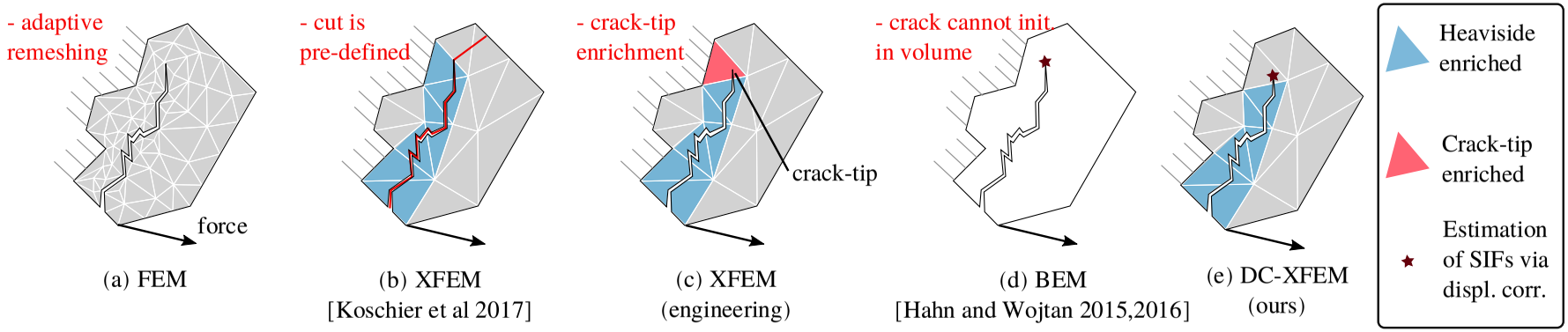


Figure 6.2: *An illustrated comparison of relevant work.*

6.3 Related Work

In this section, we review methods for simulating fracture and re-meshing in computer graphics. We then briefly review XFEM as presented in engineering literature.

6.3.1 Fracture Simulation in Computer Graphics

Physically-based fracture is a well studied problem in computer graphics, stemming from the seminal work by Terzopoulos *et al*. [146]. Early approaches proposed mass-spring systems to model brittle fracture with stress-based yield thresholds [5, 105]. However, visual artefacts were common due to spring removal and representing crack surfaces was non-trivial. Approaches based on FEM have had wider success [7, 79, 102, 103, 106, 107, 113, 128] (*cf.* Fig. 6.2 (a)). The earliest of these used nodal stress analysis to perform planar fracture for brittle and ductile material setting [106, 107]. O'Brien and Hodgins [107] introduced brittle fracture with FEM which was later extended by others [7, 79, 102]. For example, Bao *et al*. [7] also present a method for simulating both brittle and ductile (denting) fracture. A real-time method for brittle fracture is also presented Parker and O'Brien [113] which is based on O'Brien and Hodgins [107]' method but with refinement procedures to ensure that the meshes stayed self-consistent. Glondu *et al*. [44] also present a real-time approach with FEM, using modal analysis to handle crack initiation and an energy-driven algorithm for fracture propagation on implicit surfaces.

Re-meshing is frequently used in FEM simulation to handle high stress distributions accurately; align tetrahedral meshes with cracks; or to simply overcome fracture resolution constraints. Wick *et al*. [160] use dynamic local mesh refinement (and coarsening) to repair degraded tetrahedra, while Chen *et al*. [18] handle re-meshing based on gradient descent flow to enhance fracture resolution and detail. Koschier *et al*. [79] also present an adaptive subdivision scheme to facilitate cutting during fine breakage on the basis of the virtual node algorithm (VNA) [97].

The challenges of handling topological discontinuities via re-meshing can be addressed through methods such as Discontinuous Galerkin FEM (DGFEM) [71] which requires moving-least squares interpolation. The material point method (MPM) has also been used with level sets to simulate brittle and ductile fracture [54, 162]. Though impressive, MPM may preclude the ability to represent of infinitely sharp features, and with difficulties handling rigid shatter effects. XFEM embedding [70] improves accuracy but imposes limits on the fracture geometry. It scales poorly with the resolution of

the fracture surface. Richardson *et al*. [120] present a crack propagation scheme which uses a finer mesh for integration purposes and a geometric mesh cutting tool [127] for full XFEM enrichment but in 2D. An extension to 3D is described by Koschier *et al*. [78] (see also Jeřábková and Kuhlen [64]) but using only Heaviside enrichment to simulate the dynamics of meshes with cuts, since specifying of crack-tip enrichment is non-trivial in 3D. Their work also does not address the generation or propagation of fracture, so the cuts are pre-specified.

Some methods simulate fracture with surface meshes. For example, Pfaff *et al*. [118] capture the tearing of thin sheets by solving the elasto-plastic equations on a triangulated finite element mesh. Zhu *et al*. [166] perform brittle fracture based on a boundary integral formulation of elasticity combined with mesh evolution and a rigid body solver from which contact forces are extracted as Neumann boundary conditions. Hahn and Wojtan [49] (*cf.* Fig. 6.2 (d)) adapted BEM for computing stress on surfaces with spatially varying fracture parameters to produce interesting effects. They estimate stress intensity factors along crack fronts and use these for crack propagation. Their method simulates fracture on a coarse crack surface accompanied by an implicit surface to address the poor scaling properties of BEM due to singular integrals. The singularity comes from restrictions of functions from the domain to its boundary, which must be handled carefully.

6.3.2 XFEM in Engineering

Classical FEM requires adaptive meshing to handle evolving discontinuities in the simulation domain, such as in the case of fracture. XFEM (*cf.* Fig. 6.2 (c)) was first introduced by Moës *et al*. [96] to address this limitation by introducing discontinuities in the interpolation functions, thereby allowing the simulation mesh to remain unchanged (see also Belytschko and Black [10] and others [27, 30]). Mousavi *et al*. [99] present a method to handle multiple intersecting cracks using generalised harmonic enrichment functions but in 2D, which is based on the work of Kaufmann *et al*. [70]. XFEM has also been used to model fatigue crack propagation in 3D using planer cracks [134, 135] and curved surfaces [117].

Cracks in XFEM were previously modelled purely with *implicit* level set functions [130]. However, these introduced a tight coupling between the resolution of the simulation mesh and the crack surface, precluding a general ability to incorporate fine details. Alternative *explicit* mesh-based representations of the crack surface

have been explored [117, 134], along with adaptive refinement to accommodate varying propagation speeds along the crack front [43]. These explicit methods are able to model fracture surfaces more realistically and, during crack propagation, it is simpler to update a crack-surface mesh than to update level sets [43, 119]. Notably, explicit methods also require complicated computational geometric operations between the crack surface and the simulation mesh, which we address (Chapter 7). On this surface, an approach akin to Fries and Baydoun [37] is used: Fries and Baydoun propose a hybrid approach that combines the benefits of explicit and implicit representations by inducing the level function using the explicit crack mesh to enrich elements. Section 6.4 presents a review of FEM and XFEM. We refer readers to a textbook [73] for a more thorough treatment.

6.4 Elasto-static Equations with Fracture

In this section, we briefly describe the derivation of XFEM on the basis of classical FEM as described in various textbooks. A notation which is similar to Khoei [73] and Zienkiewicz and Taylor [167] is used, to which we refer interested readers for further details.

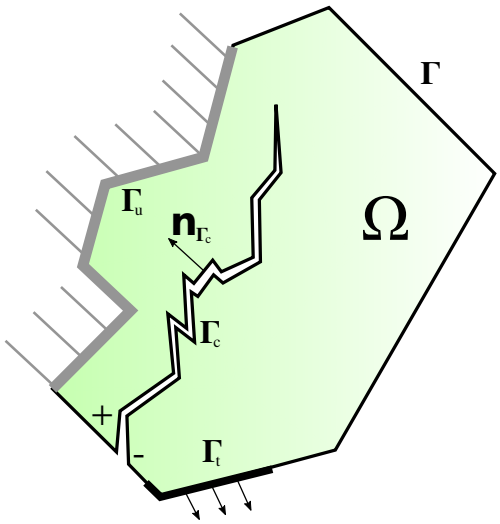


Figure 6.3: *Geometry configuration of fractured object with a crack* Γ_c.

6.4.1 Governing Equations

Consider a body of *linear* elastic material occupying a domain $\Omega \subset \mathbb{R}^3$, with boundary Γ and subjected to body loading b as shown in Fig. 6.3. To describe the deformation

behaviour, our problem is to satisfy the elasto-static equation

$$\nabla \cdot \sigma = -b, \tag{6.1}$$

by finding a displacement function $\mathbf{u}(\mathbf{x}), \mathbf{x} \in \Omega$ under static equilibrium[1]. Eq. (6.1) is the *strong form* of a steady-state problem, where $\nabla\cdot$ denotes the divergence operator $\frac{\partial\sigma}{\partial\mathbf{x}}$, and σ is cauchy stress which is a function of $\mathbf{u}(\mathbf{x})$. It is so-called *strong* because the partial differential equation (PDE) must hold at every point in Ω[2].

The boundary Γ consists of the subsets Γ_c, Γ_t and Γ_u. The region Γ_c is the interior crack surface which is assumed to be traction free. Prescribed displacements and boundary forces (tractions) are specified along Γ_u and Γ_t respectively [3]

$$\mathbf{u} = \bar{\mathbf{u}} \text{ on } \Gamma_u \quad \text{and} \quad \sigma \cdot \mathbf{n} = \bar{\mathbf{t}} \text{ on } \Gamma_t \tag{6.2}$$

where $\bar{\mathbf{u}}$ and $\bar{\mathbf{t}}$ are the prescribed displacements and tractions, and $\mathbf{n}$ is the outward normal to Ω.

6.4.2 FEM Approximation of Linear Elastic Mechanics

FEM reformulates the above strong form of the problem as a set of algebraic equations via a *weak form* that can be solved numerically. The key difference is that it focuses on finding an unknown vector of displacements $\mathbf{u}$ rather than a continuous function. The function $\mathbf{u}(\mathbf{x})$ is then obtained from the vector using relevant interpolation functions known as *shape functions*. The choice of shape functions depends on the discretisation of Ω into *elements*, and results in a system of algebraic equations.

Shape Functions

As we work with the *linear* tetrahedron for representing an element e, it will be specified by four nodes (vertices) in $\mathbb{R}^3$ and their displacements $\mathbf{u}_{i=1\ldots4} \in \mathbb{R}^3$. Thus, the

[1] Elastic forces are often expressed as a function of displacements, so that zero displacement results in zero force, but can also be written as a function of position.

[2] Another reason is that there are two derivatives on the the field of interest $\mathbf{u}(\mathbf{x})$ which is a strong condition ('smoothness'). Thus, for Eq. (6.1) to make sense $\mathbf{u}(\mathbf{x})$ should be a function of a type that allows second-order derivatives.

[3] In various literature, Γ_u and Γ_t are also referred to as the Dirichlet and Neumann boundary conditions.

interpolated displacement at a given point $\mathbf{x} \in e$ is

$$\mathbf{u}(\mathbf{x}) = \sum_{i=1}^{4} \mathbf{N}_i(\mathbf{x})\mathbf{u}_i \tag{6.3}$$

$$= [\mathbf{IN}_1, \mathbf{IN}_2, \mathbf{IN}_3, \mathbf{IN}_4]\,\mathbf{u}_e \tag{6.4}$$

$$= \mathbf{N}\mathbf{u}_e, \tag{6.5}$$

where $\mathbf{I}$ is the 3×3 identity matrix, and $\mathbf{N}_i$ are shape functions associated with the nodes of element e. The element is referred to as linear since its shape functions are linear polynomials due to having only four nodes (see e.g. Eq 6.8 in [167]).

Strain and Stress

Elastic deformation is measured by assuming infinitesimally small displacements, which admits a linearisation of the constitutive model relating strain to stress: $\sigma = \mathbf{D}\varepsilon$. The strain ε at any point within the element is determined by six components which contribute to internal work

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{bmatrix} = \frac{\partial \mathbf{N}}{\partial \mathbf{x}}\mathbf{u}. \tag{6.6}$$

Thus, strain–a descriptor for the severity of deformation–and thus, stress are constant in an element e:

$$\varepsilon_e(\mathbf{x}) = \mathbf{B}_e\mathbf{u}_e; \qquad \sigma_e(\mathbf{x}) = \mathbf{D}\,\varepsilon_e(\mathbf{x}). \tag{6.7}$$

Here $\mathbf{B}_e = \left[\mathbf{B}_e^1, \mathbf{B}_e^2, \mathbf{B}_e^3, \mathbf{B}_e^4\right] \equiv \frac{\partial \mathbf{N}}{\partial \mathbf{x}}$ is the constant 6×12 discretisation gradient matrix containing the partial derivatives of the shape functions. For a node i, we get $\mathbf{B}_e^i$ as

$$\mathbf{B}_e^i = \begin{bmatrix} \frac{\partial \mathbf{N}_i}{\partial x} & 0 & 0 \\ 0 & \frac{\partial \mathbf{N}_i}{\partial y} & 0 \\ 0 & 0 & \frac{\partial \mathbf{N}_i}{\partial z} \\ \frac{\partial \mathbf{N}_i}{\partial y} & \frac{\partial \mathbf{N}_i}{\partial x} & 0 \\ 0 & \frac{\partial \mathbf{N}_i}{\partial z} & \frac{\partial \mathbf{N}_i}{\partial y} \\ \frac{\partial \mathbf{N}_i}{\partial z} & 0 & \frac{\partial \mathbf{N}_i}{\partial x} \end{bmatrix}, \tag{6.8}$$

which is the corresponding block entry in $\mathbf{B}_e$. Finally, $\mathbf{D}$ is the elasticity matrix which encodes (isotropic) material properties in terms of young modulus E and Poisson's ratio ν which are the elastic constants

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ & 1-\nu & \nu & 0 & 0 & 0 \\ & & 1-\nu & 0 & 0 & 0 \\ & & & \frac{(1-2\nu)}{2} & 0 & 0 \\ & \text{Sym.} & & & \frac{(1-2\nu)}{2} & 0 \\ & & & & & \frac{(1-2\nu)}{2} \end{bmatrix} \tag{6.9}$$

As we assume isotropic material i.e. strain-stress relation has no directional dependence, it is worth mentioning that the stress can be further simplified to a form often seen in computer graphics literature. In this case, the stress-strain model which uses the Lamé coefficients λ and μ is also

$$\sigma(\mathbf{u}_e) = 2\mu\varepsilon(\mathbf{u}_e) + \lambda tr(\varepsilon(\mathbf{u}_e))\mathbf{I}, \tag{6.10}$$

since $\mathbf{D}$ has only 2 independent components [48]. It is also common that the strain ε is defined as a function of the deformation gradient (see e.g. Teran *et al*. [144] and Sifakis [126] for details). We maintain using E and ν (as in Eq. (6.7)) for consistency, which are related to the Lamé parameters by

$$E = \frac{\mu(3\lambda+2\mu)}{\lambda+\mu} \quad \text{and} \quad \mu = \frac{\lambda}{2(\lambda+\mu)}.$$

The Stiffness Matrix

By the principle of virtual work, variations in internal work must equal variations in external due to boundary constraints (*cf.* Eq. 2.13a in [167]) which gives the following linear system within each element

$$\mathbf{K}_e\mathbf{u}_e = \mathbf{f}_e \quad \text{where,} \quad \mathbf{K}_e \equiv \int_e \mathbf{B}_e^\intercal \mathbf{D}_e \mathbf{B}_e \, \mathrm{d}\mathbf{x}, \quad \mathbf{f}_e \equiv \int_e \mathbf{N}^\intercal \bar{\mathbf{t}} \, \mathrm{d}\mathbf{x}. \tag{6.11}$$

Thus, $\mathbf{K}_e$ is the 12×12 stiffness matrix obtained from material properties and $\mathbf{f}_e$ is obtained from external forces $\mathbf{t}_e$ (boundary conditions) acting on the element. For simulating a domain with many elements, the element-wise $\mathbf{K}_e$ from all elements are carefully assembled into a global sparse linear system $\mathbf{K}$ to solve for all unknown nodal displacements $\mathbf{u}$.

In the presence of fracture, displacements are non-smooth within elements e intersected by the crack. FEM addresses this problem by aligning the elements with the

discontinuity (crack surface). This requires re-meshing which needs special care but also results in a larger $\mathbf{K}$ for high-resolution crack surfaces (more degrees of freedom) as shown in Fig. 6.2 (a).

6.4.3 XFEM for Simulating Fracture

XFEM copes with cracks running through elements by enriching the approximation space independent of the elements. (Here we consider only *extrinsic* enrichment where more shape functions and unknowns result in the approximation, while operating on the same domain elements). An element that is cut by a fracture surface may either be completely split or contain the crack-tip within its volume. The former results in a strong discontinuity while the latter leads to singularities in the near-field displacements.

Let $\mathbf{x} \in \mathbb{R}^3$ be a point which lies inside of an enriched element e. The displacement approximation (interpolation)

$$\mathbf{u}_e^{enr}(\mathbf{x}) = \sum_{i=1}^{m} \mathbf{N}_i(\mathbf{x})\, \mathbf{u}_e^i + \Upsilon_{\text{hev}}(\mathbf{x}) + \Upsilon_{\text{tip}}(\mathbf{x}) \text{ where} \tag{6.12}$$

$$\Upsilon_{\text{hev}} = \sum_{i=1}^{m} \mathbf{N}_i^{enr}(\mathbf{x})\, (\Psi_{\text{hev}}(\mathbf{x}) - \Psi_{\text{hev}}(\mathbf{x}_i))\, \mathbf{a}_i \text{ and} \tag{6.13}$$

$$\Upsilon_{\text{tip}} = \sum_{i=1}^{m} \sum_{k=1}^{4} \mathbf{N}_i^{enr}(\mathbf{x})\, (\Psi_{\text{tip}}^k(\mathbf{x}) - \Psi_{\text{tip}}^k(\mathbf{x}_i))\, \mathbf{b}_i^k, \tag{6.14}$$

sums over all m nodes of the element[4]. For a tetrahedron element, $m = 4$ (we assume that $\mathbf{N}_i \equiv \mathbf{N}_i^{enr}$ without no loss of generality). In this element, $\Psi_{\text{hev}}(\mathbf{x})$ is the Heaviside (jump) function, and $\Psi_{\text{tip}}^k(\mathbf{x})$ are the four asymptotic crack-tip functions to capture singularities[5]. The variables $\mathbf{a}_i \in \mathbb{R}^3$ and $\mathbf{b}_i^k \in \mathbb{R}^3$ are vectors of added degrees of freedom (DOF) due to the enrichment.

The Heaviside function

$$\Psi_{\text{hev}}(\mathbf{x}) = \begin{cases} +1 & sign(\xi(\mathbf{x})) > 0 \quad \text{above} \\ -1 & sign(\xi(\mathbf{x})) < 0 \quad \text{below} \end{cases}, \tag{6.15}$$

describes the location of $\mathbf{x}$ with respect to the crack, where

$$\xi(\mathbf{x}) = \min ||\mathbf{n}_{\Gamma_c} \cdot (\mathbf{x} - \mathbf{x}_{\Gamma_c})||, \tag{6.16}$$

[4] A shifted function e.g. $\Psi_{\text{hev}}(\mathbf{x}) - \Psi_{\text{hev}}(\mathbf{x}_i)$, is used since generally approximations of the form Eq. (6.12) lack the Kronecker-delta property necessary to ensure that $\mathbf{u}(\mathbf{x}_i) = \mathbf{u}_i$

[5] Specific definitions of the crack-tip enrichment functions are omitted here because they are not necessary in the formulation of the method described herein this chapter, but readers are referred to Khoei [73] for details

is the signed distance to the closest point on the crack surface $\mathbf{x}_{\Gamma_c}$ with normal $\mathbf{n}_{\Gamma_c}$ (*cf.* Fig. 6.3).

Extended Strain

Thus, the enriched approximation of Eq. (6.5) becomes

$$\mathbf{u}_e(\mathbf{x}) = \mathbf{N}\mathbf{u}_e + \mathbf{N}^{enr}\mathbf{a} \quad \text{or} \quad \mathbf{u}_e(\mathbf{x}) = \mathbf{N}\mathbf{u}_e + \mathbf{N}^{enr}\mathbf{b}, \tag{6.17}$$

depending on whether the nodes are Heaviside or crack-tip enriched respectively. Likewise, the extended strain (*cf.* Eq. (6.7)) corresponding to the approximate displacement field is

$$\boldsymbol{\varepsilon}_e(\mathbf{x}) = \mathbf{B}_e\mathbf{u}_e + \mathbf{B}_e^{hev}\mathbf{a} \quad \text{or} \quad \boldsymbol{\varepsilon}_e(\mathbf{x}) = \mathbf{B}_e\mathbf{u}_e + \mathbf{B}_e^{tip}\mathbf{b}. \tag{6.18}$$

The terms

$$\mathbf{B}_e^{hev} = \frac{\partial}{\partial \mathbf{x}}\mathbf{N}^{enr}(\Psi_{\text{hev}}(\mathbf{x}) - \Psi_{\text{hev}}(\mathbf{x}_i)) \quad \text{and} \quad \mathbf{B}_e^{tip} = \frac{\partial}{\partial \mathbf{x}}\mathbf{N}^{enr}(\Psi_{\text{tip}}(\mathbf{x}) - \Psi_{\text{tip}}(\mathbf{x}_i))$$

are the spatial derivatives of the enrichment shape functions which have the same structure as in Eq. (6.8). These derivatives are obtained using the product rule by

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{N}_i^{enr}(\Psi_*(\mathbf{x}) - \Psi_*(\mathbf{x}_i)) = \frac{\partial}{\partial \mathbf{x}}\mathbf{N}_i^{enr}(\Psi_*(\mathbf{x}) - \Psi_*(\mathbf{x}_i)) + \mathbf{N}_i^{enr}\frac{\partial \Psi_*(\mathbf{x})}{\partial \mathbf{x}}, \tag{6.19}$$

to define the corresponding matrix entries, where Ψ_* is a placeholder for Ψ_{hev} or Ψ_{tip}^k. Thus, the enhanced discretized gradient matrix containing enrichment terms is defined for each element as

$$\mathbf{B}_e^{enr} = [\mathbf{B}_e, \mathbf{B}_e^*], \tag{6.20}$$

where $\mathbf{B}_e^*$ is a placeholder for $\mathbf{B}_e^{hev}$ or $\mathbf{B}_e^{tip}$ depending on the type of enrichment affecting element e. In the case of Heaviside enrichments $\mathbf{B}_e^{enr}$ is a 6×24 matrix, and 6×60 for crack-tip enrichments. This is because three DOFs are added per node, for each enrichment function used in e.

Extended Stiffness Matrix

Enrichment means that the structure of the global stiffness matrix, and thus the number of DOFs, will change and increase in size. At the element level, rather than solving for $\mathbf{u}_e$ (as in FEM), the new system $\mathbf{K}_e^{\text{enr}}\mathbf{u}_e^{\text{enr}} = \mathbf{f}_e^{\text{enr}}$ has additional unknowns:

$$\mathbf{u}_e^{\text{enr}} = [\mathbf{u}^\intercal\ \mathbf{a}^\intercal]^\intercal \quad \text{or} \quad \mathbf{u}_e^{\text{enr}} = \left[\mathbf{u}^\intercal\ \mathbf{b}_1^\intercal\ \mathbf{b}_2^\intercal\ \mathbf{b}_3^\intercal\ \mathbf{b}_4^\intercal\right]^\intercal, \tag{6.21}$$

which–once again–depends on whether the nodes are Heaviside enriched or crack-tip enriched respectively. For the global stiffness matrix $\mathbf{K}$, appropriately ordering DOFs gives rise to the following block structure

$$\begin{bmatrix} \mathbf{K}_{uu} & \mathbf{K}_{ua} & \mathbf{K}_{ub} \\ \mathbf{K}_{au} & \mathbf{K}_{aa} & \mathbf{K}_{ab} \\ \mathbf{K}_{bu} & \mathbf{K}_{ba} & \mathbf{K}_{bb} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{a} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{f_u} \\ \mathbf{f_a} \\ \mathbf{f_b} \end{bmatrix}, \tag{6.22}$$

where $\mathbf{K}_{uu}$ is the traditional FEM stiffness matrix; $\mathbf{K}_{aa}$ and $\mathbf{K}_{bb}$ are the enrichment stiffness matrices; and the remaining blocks are coupling matrices between e.g. the traditional and enriched stiffness components etc. These block entries have the form

$$\mathbf{K} = \begin{bmatrix} \int_\Omega \left(\mathbf{B}^{\text{std}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{std}} & \int_\Omega \left(\mathbf{B}^{\text{std}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{hev}} & \int_\Omega \left(\mathbf{B}^{\text{std}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{tip}} \\ \int_\Omega \left(\mathbf{B}^{\text{hev}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{std}} & \int_\Omega \left(\mathbf{B}^{\text{hev}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{hev}} & \int_\Omega \left(\mathbf{B}^{\text{hev}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{tip}} \\ \int_\Omega \left(\mathbf{B}^{\text{tip}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{std}} & \int_\Omega \left(\mathbf{B}^{\text{tip}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{hev}} & \int_\Omega \left(\mathbf{B}^{\text{tip}}\right)^\intercal \mathbf{D}\mathbf{B}^{\text{tip}} \end{bmatrix}, \tag{6.23}$$

where the superscript `std` is used to emphasis that the respective terms are constructed as in standard/classical FEM assemblage. The extended vector of the applied forces in Eq. (6.22) is

$$\mathbf{f} = \begin{bmatrix} \int_{\Gamma_t} \left(\mathbf{N}^{\text{std}}\right)^\intercal \mathbf{t} \\ \int_{\Gamma_t} \left(\mathbf{N}^{\text{hev}}\right)^\intercal \mathbf{t} \\ \int_{\Gamma_t} \left(\mathbf{N}^{\text{tip}}\right)^\intercal \mathbf{t} \end{bmatrix}, \tag{6.24}$$

where $\mathbf{N}_i^{\text{std}} = \mathbf{N}_i(\mathbf{x})\mathbf{I}$, $\mathbf{N}_i^{\text{hev}} = \mathbf{N}_i(\mathbf{x})(\Psi_{\text{hev}}(\mathbf{x}) - \Psi_{\text{hev}}(\mathbf{x}_i))\mathbf{I}$, and $\mathbf{N}_i^{\text{tip}} = \mathbf{N}_i(\mathbf{x})(\Psi_{\text{tip}}(\mathbf{x}) - \Psi_{\text{tip}}(\mathbf{x}_i))\mathbf{I}$, with $\mathbf{I}$ being a 3×3 identity matrix.

The number of rows in the updated global stiffness matrix increases from $3n$ to $3(n+n_h)$ in the case of Heaviside enrichment and to $3(n+n_h+4n_t)$ in the case of Heaviside and crack-tip enrichment. Here n is the total number of nodes in the discretised domain (e.g. tetrahedral mesh), n_h is the number of nodes on elements with a complete crack passing through them and n_t is the number of nodes on elements containing crack front[6]. Thus, enrichment also means that all the new blocks (i.e. those with superscripts a and/or b in Eq. (6.22)) and the vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{f}$ will grow with each new element that is intersected by the crack, but the size (and structure) of $\mathbf{K}_{uu}$ and $\mathbf{u}$ remains unchanged.

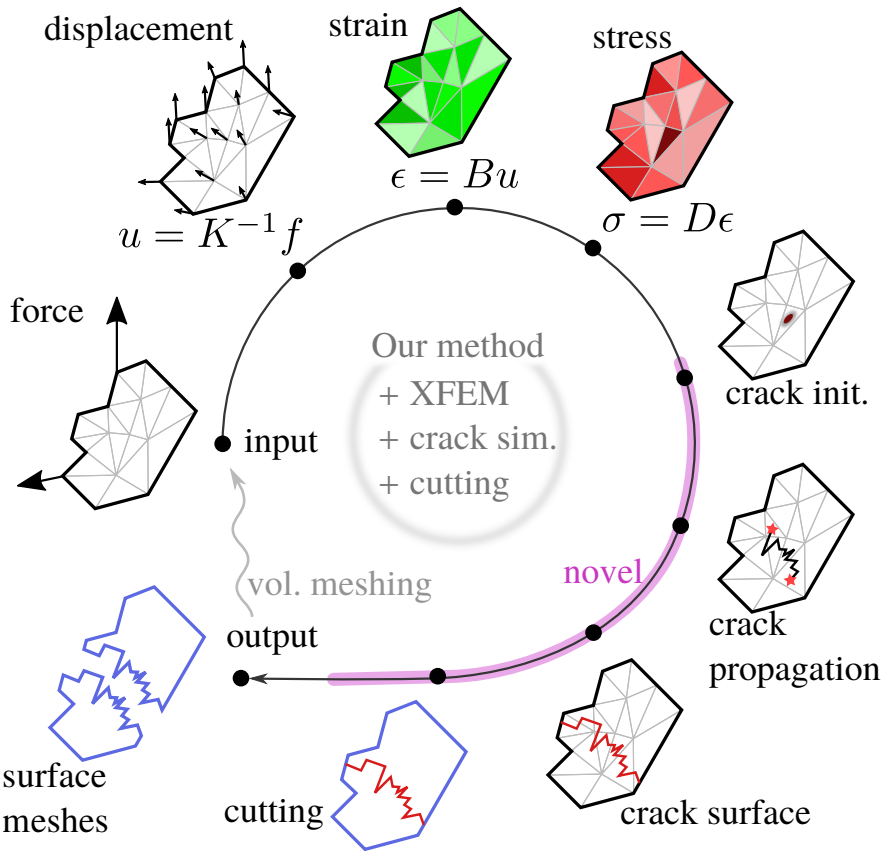


Figure 6.4: *Illustrative summary of the different stages of our method.*

6.5 Method Overview

Having briefly described XFEM on the basis of classical FEM in previous sections, we now provide an overview of our method to simulate brittle fracture which is also illustrated in Fig. 6.4.

Starting from a (possibly high resolution and detailed) surface mesh, we first convert it to a lower-resolution mesh to accelerate simulation and collision detection (see Section 6.8) and construct a finite element (tetrahedral) mesh for XFEM. With the tetrahedral mesh, we perform simulation by applying given boundary conditions and computing displacements and stress as in standard XFEM. We simulate crack growth by emulating the existence of singular "crack-tip stress", correlating finite element displacements with crack-tip displacement equations in elements containing the crack front (Section 6.6.2). Thus, crack initiation and propagation can be handled separately (Section 6.6.3) as in linear elastic fracture mechanics (LEFM). Since we operate directly on an explicit surface mesh, our crack propagation algorithm proceeds as shown in Fig. 6.8, extending crack front vertices according to the strain energy release rate. We make use of an induced signed distance function by intersecting elements with our explicit crack surface mesh to evaluate enrichment functions–which we do after each propagation step. Finally, we compute the disconnected mesh fragments using the generated explicit crack surface and the high detail mesh for visualisation. (The details of this cutting procedure are described in Chapter 7).

6.6 Displacement-Correlated XFEM

Assuming an unfractured object, boundary conditions, and a finite element mesh such that we can assemble global matrix $\mathbf{K}$ and vector $\mathbf{u}$, we describe in Section 6.6.1 how to add a crack to this system while using only Heaviside enrichment (Eq. (6.13)). The dynamics of fracture in a quasi-static and volumetric setting are then presented in Section 6.6.2, which we use to calculate of fracture-mechanical loading parameters. We then describe our crack mesh representation and the steps to initiate a crack, and compute crack-front motion using the computed loading parameters in Section 6.6.3.

[6] In general, $n_h + n_t \ll n$ since it is only elements intersected by the crack surface that are enriched.

6.6.1 System Equations

As we saw in Section 6.4.3, a crack in XFEM is represented by two types of enrichments, Υ_{step} and Υ_{tip}, which capture strong discontinuities in the displacement field and stress singularities respectively. Creating additional DOFs is the common property of these types of enrichments. However, treating these enrichments simultaneously when formulating XFEM is challenging: In addition to being cumbersome to implement, the accuracy of crack-tip enrichment is limited in 3D as existing methods (which rely on 2D crack-tip parameters) require the stiffness matrix and force vector to be expanded arbitrarily to account for the extra unknowns. Accuracy further suffers from the lack of a clear definition of 'normal to the crack front' for points further away from the crack front. Another drawback is the introduction four times the DOFs compared to Heaviside enrichment (48 compared to 12) while performing extrinsic enrichment.

One possible way to circumvent this issue is *intrinsic* XFEM [38] which replaces the element shape functions by special ones with no additional unknowns to capture non-smooth solutions. This method, however, requires careful treatment of enriched moving least-squares functions near discontinuities as well as special weighting functions.

We propose a simpler *displacement correlated* XFEM on the basis of the extrinsic formulation but we reduce the system to only requiring Heaviside enrichment. We capture strong discontinuity as per Heaviside enrichment, but correlate analytical expressions for crack-tip displacement with numerically obtained smooth solutions to estimate stress intensities. A fracture can still be represented by a single sheet of triangles. Using this surface, the weak form of the linear elasticity equations (*cf.* Eq. 2.58 in [73]) is applied after dropping the (zero) surface traction term. The reduced global system is (*cf.* Eq. (6.22))

$$\begin{bmatrix} \mathbf{K}_{uu} & \mathbf{K}_{ua} \\ \mathbf{K}_{au} & \mathbf{K}_{aa} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{f_u} \\ \mathbf{f_a} \end{bmatrix}, \tag{6.25}$$

which we solve using the method of conjugate gradients.

To resolve the issue of estimating stress intensities without crack-tip enrichment (and hence, crack opening displacements), we simply require that crack-front vertices have a reference element in which they reside (see Section 6.6.2). Thus, there are no unknowns introduced at the crack front but we add DOFs according the elements completely cut by the crack.

Quadrature Integration inside Enriched Elements

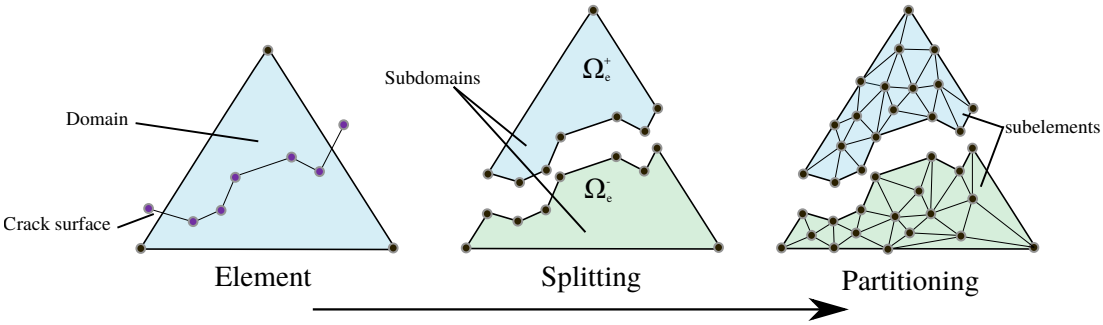


Figure 6.5: *Partitioning an enriched element into sub-elements for integration purposes.*

For standard elements, the Gauss integration rule can be directly used to evaluate the integral of the stiffness matrix (*cf.* Eq. (6.11)). However, this integration rule is insufficient for the elements intersected by the crack surface due to the non-smoothness of the interpolated displacements. To overcome this insufficiency, we adopt a common approach [39, 73] which splits and partitions each intersected element into sub-elements (*cf.* Fig. 6.5). These sub-elements align with the crack intersection and allow for retaining the ability to use standard quadrature when computing the enriched stiffness matrix. (Note that splitting and partitioning is performed strictly for integration purposes and it does not affect system DOFs).

Thus, considering an element e that is intersected into two parts Ω_e^+ and Ω_e^-, the enriched element stiffness matrix is constructed by

$$\mathbf{K}_{ij}^{\alpha\beta} = \int_{\Omega_e} (\mathbf{B}_i^\alpha)^\intercal \mathbf{D} \mathbf{B}_j^\beta \, d\Omega_e \quad \alpha,\beta = \texttt{std},\texttt{hev} \tag{6.26}$$

$$= \int_{\Omega_e^+} (\mathbf{B}_i^\alpha)^\intercal \mathbf{D} \mathbf{B}_j^\beta \, d\Omega_e^+ + \int_{\Omega_e^-} (\mathbf{B}_i^\alpha)^\intercal \mathbf{D} \mathbf{B}_j^\beta \, d\Omega_e^- \tag{6.27}$$

$$= \sum_{l=1}^{N_{sub}^+} \left(\sum_{k=1}^{N_{GP}^+} \left(\mathbf{B}_{i(k)}^\alpha \right)^\intercal \mathbf{D} \mathbf{B}_{j(k)}^\beta w_k \right)_l + \sum_{l=1}^{N_{sub}^-} \left(\sum_{k=1}^{N_{GP}^-} \left(\mathbf{B}_{i(k)}^\alpha \right)^\intercal \mathbf{D} \mathbf{B}_{j(k)}^\beta w_k \right)_l , \tag{6.28}$$

which is a 24×24 matrix with a block structure similar to the global system in Eq. (6.25). The subscripts i and j are nodal indices, where $1 \leq i, j \leq 4$. N_{sub}^+ and N_{sub}^- are the number of sub-elements in Ω_e^+ and Ω_e^- respectively. N_{GP}^* is the number of gauss quadrature points used for integration where each has a weight w_k (we assume $N_{GP}^* = 1$ since our sub-elements are tetrahedra).

Although special integration rules are available for arbitrary polyhedra e.g. [78, 100], tetrahedral decomposition is easier with ready-made tools for construction which are sufficiently robust for our use-case. We split enriched elements using the mesh

cutting algorithm which is described in Chapter 7 (see Section 6.8 for details on partitioning).

6.6.2 Fracture Dynamics

We now describe how finite element displacements (from Eq. (6.25)) near the crack front are used to emulate singular stress fields as in linear elastic fracture mechanics (LEFM). This formulation allows us to propagate the crack using the strain energy release rate [4], which is ideal for treating crack initiation and propagation separately (*cf.* Section 6.6.3).

Stress Intensities Near the Crack Front

Using the displacement $\mathbf{u}$ computed by solving Eq. (6.25), we can calculate stress and thus stress intensities near the crack front for propagation. We adopt the displacement method [17] to estimate stress intensities by correlating computed displacements with known crack-front displacement equations. These equations are evaluated at a location $\mathbf{x}_{\Gamma_{cp}}(r,\theta)$ which is on the crack face. This location is called the *correlation point* [61], where r and θ are polar coordinates which are defined relative to the crack front as shown in Fig. 6.6.

The analytical displacement equations near the crack front, given $\mathbf{x}_{\Gamma_{cp}}(r,\theta)$, are defined by

$$\begin{aligned} u_2^* &= \frac{K_I}{2\mu}\sqrt{\frac{r}{2\pi}}g_2^I(\theta)+\frac{K_{II}}{2\mu}\sqrt{\frac{r}{2\pi}}g_2^{II}(\theta) \\ u_1^* &= \frac{K_I}{2\mu}\sqrt{\frac{r}{2\pi}}g_1^I(\theta)-\frac{K_{II}}{2\mu}\sqrt{\frac{r}{2\pi}}g_1^{II}(\theta) \\ u_3^* &= \frac{2K_{III}}{\mu}\sqrt{\frac{r}{2\pi}}g_3^{III}(\theta), \end{aligned} \tag{6.29}$$

where μ is the shear modulus. $u_{i=1,2,3}^*$ are the displacement components at $\mathbf{x}_{\Gamma_{cp}}(r,\theta)$ which correspond to the crack loading modes which are illustrated in Fig. 6.7. $K_{L=I,II,III}$ are the *stress intensity factors* (SIFs) which characterise the stress singularity near the tip of the crack. $g_i^L(r,\theta)$ are the angular functions

$$g_1^I = \cos\frac{\theta}{2}(\kappa-\cos\theta), \quad g_2^I = \sin\frac{\theta}{2}(\kappa-\cos\theta) \tag{6.30}$$

$$g_1^{II} = \sin\frac{\theta}{2}(\kappa+\cos\theta+2), \quad g_2^{II} = \cos\frac{\theta}{2}(\kappa+\cos\theta-2) \tag{6.31}$$

$$g_3^{III} = \sin\frac{\theta}{2}, \tag{6.32}$$

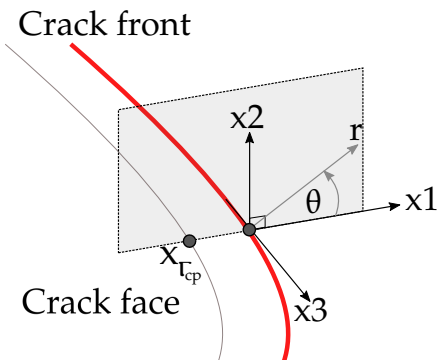


Figure 6.6: *Local coordinate system perpendicular to crack front.*

where $\kappa = 3 - 4\nu$ is the elastic constant for plane strain. These describe stress change along the circumference direction near the crack tip (*cf.* Eq. 3.16, 3.25 and 3.31 in [80]).

Eq. (6.29) can be further simplified - from which corresponding expressions for $K_{L=I,II,III}$ are determined. This is because the SIFs are evaluated on the crack face and relative to the crack front, which is the location of crack opening where $\theta = \pi$ (*cf.* Fig. 6.6). Thus, for crack geometries with pure mode-I loading, crack front displacements reduce to

$$u_1^* = \frac{1}{2\mu}\sqrt{\frac{r}{2\pi}} K_I g_1^I(\theta). \tag{6.33}$$

The corresponding SIF is determined via conversion to give

$$K_I = 2\mu\sqrt{\frac{2\pi}{r}}\frac{u_1^*}{g_1^I(\theta)}, \tag{6.34}$$

which is the simplest method to determine K_I. Extending to pure mode-II and mode-III loading for K_{II} and K_{III}, we have

$$[K_I \;\; K_{II} \;\; K_{III}] = \mu\sqrt{\frac{\pi}{2r}}\left[\frac{u_2^*}{(2-2\nu)} \;\; \frac{u_1^*}{(2-2\nu)} \;\; u_3^*\right], \tag{6.35}$$

which describes how we compute the SIFs that we use for crack propagation. The expressions for K_{II} and K_{III} in Eq. (6.35) are derived by applying similar interpretations as K_I but using g_i^{II} with $u_{i=1,2}^*$ and g_i^{III} with u_3^* in Eq. (6.33) and Eq. (6.34) respectively. The solution is simple, fast and sufficiently accurate for our purposes even though more accurate methods exist (e.g. J-integral method [61]) - each SIF is associated exactly with the displacement component of a crack opening mode.

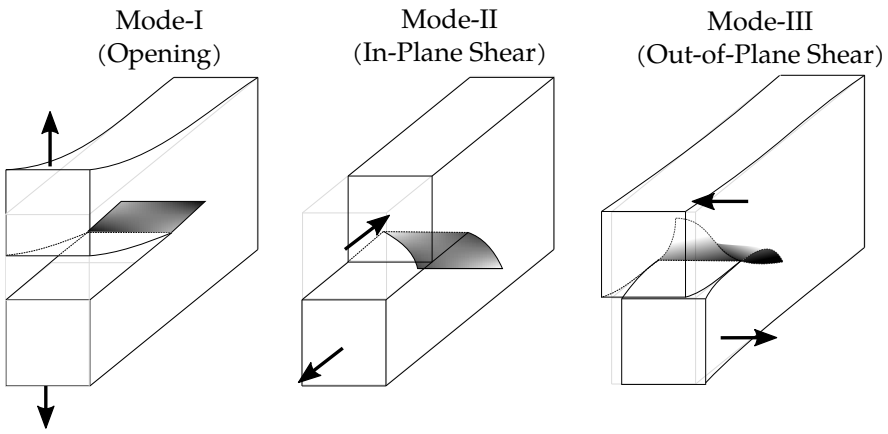


Figure 6.7: *Arrows: Loading modes; Shaded: expected propagation behaviour (*[62]*). Mode-I opens the crack perpendicular to the crack plane and makes it propagate forward due to the applied tensile loading. In Mode-II, the crack faces are displaced on their plane, normal to the crack front. For Mode-III, the crack is displaced on its 'plane', parallel to the crack front.*

Virtual Crack Opening Displacements

In practice, evaluating Eq. (6.35) is dependent on the relation between u_i^* and the solution $\mathbf{u}$, which we now describe.

We can–for a moment–assume that the crack has already been initiated (as described in Section 6.6.3), since Eq. (6.35) is given on the basis of the existence of a crack. Thus, given $\mathbf{x}_{\Gamma_{cp}}$, we compute u_i^* from $\mathbf{u}$ by exploiting polynomial approximations on the element e containing $\mathbf{x}_{\Gamma_{cp}}$ as interpolated nodal displacements $\mathbf{u}_e$, which we get after solving the linear system in Eq. (6.25). The location $\mathbf{x}_{\Gamma_{cp}}$ is given near the boundary of the crack mesh (e.g. a point along interior dashed blue line in Fig. 6.8, right). So we evaluate u_i^* at $\mathbf{x}_{\Gamma_{cp}}$ using Eq. (6.3) and then projecting onto the local orthonormal basis of the crack-front (*cf.* Fig. 6.6). A disadvantage of this approach is the assumption that the interpolated displacements are representative of the crack opening displacements which leads to a loss of symmetry properties when all three modes I, II, and III are superimposed. However, we believe that the simplicity of this formulation outweighs the drawbacks since it results in fewer DOFs due to enrichment while still retaining the advantages of explicit XFEM.

6.6.3 Crack Initiation and Propagation

Having described how we compute fracture loading parameters from finite element displacements, we now provide the details of how a crack is initiated, and propagated

using the loading parameters from Section 6.6.2.

Crack Mesh Initiation

During XFEM, a crack is initiated according to a fracture criterion which is based on the traditional Rankine condition: brittle material fails if the maximum principal stress exceeds material strength

$$\sigma_{\text{max}} \geq \sigma_{\text{critical}}.$$

Thus, crack initiation and its location are determined as follows: First, we calculate the nodal displacements followed by stresses in all elements. Then, eigen analysis is computed on each element's stress tensor to find direction of maximum tensile (or compressive) stress by comparing based on the largest magnitude: The maximal and minimal principal stresses are compared to the tensile and compressive strength respectively. If the Rankine condition is satisfied, we then create an initial mesh at the element centre.

Detail: Principal stress directions are found by the solving eigenvalue problem: $\sigma \mathbf{n}_i = \lambda_i \mathbf{n}_i$. There are three real eigenvalues λ_i which represent the principal (normal) stresses. These eigenvalues are associated with three eigenvectors $\mathbf{n}_i$ that are each also associated with the principal stress directions. The planes orthogonal to the principal stress directions are also called the *principal planes* [48].

The three-dimensional crack surface is represented by an explicit mesh which we initially create as a single sheet of triangles spanning a circular disk [7]. We create and align this mesh to be orthogonal to the principal stress direction (tensile or compressive) such that it is inline with the principal stress plane and therefore the expected crack propagation direction. Disk diameter is scaled according to element size, but may be set manually e.g. as a fraction of the domain's bounding box diagonal to control resolution. Since all the triangles of this disk mesh are vertex adjacent, each forms part of the initial crack front which will be propagated.

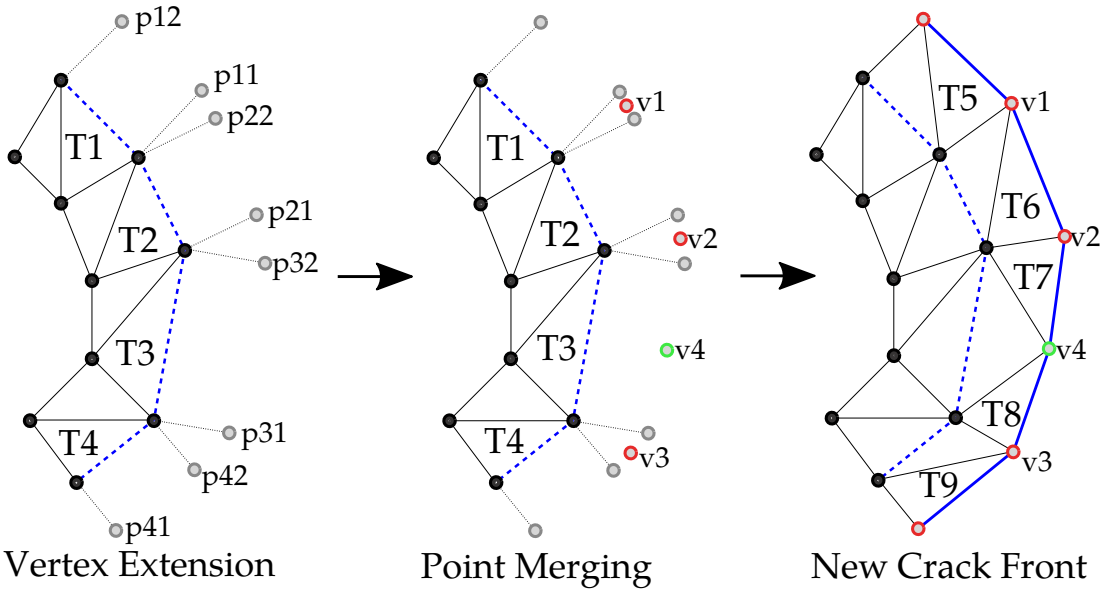


Figure 6.8: *Crack mesh extension: Triangles T1-T4 form the current crack front (dashed blue line, left and middle). We extend two vertices for each triangle along the crack front, and then combine the extended points (red) or insert new ones e.g. v4 (green) according to the new edge length. Finally, we build triangles forming new crack front T5–T9 (solid blue line, right).*

Crack Mesh Propagation

Propagation of the crack front occurs according to the strain energy release rate [4] where the crack propagates iff $K_* > K_t$, where

$$K_* = \sqrt{\frac{K_I^2 + K_{II}^2 + K_{III}^2}{(1-\nu)}} \text{ and, } K_t = \sqrt{\frac{2\gamma E}{(1-\nu^2)}}$$

are the effective stress intensity and material toughness. The new parameters are as follows: γ is the surface energy which is chosen experimentally, E is Young's modulus and ν is Poisson's ratio of an element incident to the crack-front. We calculate the propagation speed and direction at crack front vertices similarly to Hahn and Woltan [49] (see also [114]), but in a volumetric setting to propagate the fracture surface mesh. Propagation speed $s = c_R(1 - K_*/K_t)$ is the linear approximation of the upper limit of the Rayleigh wave speed $c_R \approx 0.57\sqrt{E/\rho}$ [80], where ρ is the material density which we set. We compute the propagation direction (*cf.* Eq. 10 in [49], and Eq. 43 in [114]) by

$$\theta = 2\,\text{atan}\left[\left(K_{I,III} - \left(K_{I,III}^2 - 8k_{II}^2\right)^{\frac{1}{2}}\right)/4K_{II}\right],$$

[7] other types of meshes are also possible e.g. rectangular mesh, which was in Fig. 6.9

using $K^2_{I,III} \equiv K^2_I + K^2_{III}/(1-\nu)$.

Starting from the crack front triangles, we extend the crack mesh during propagation as follows (*cf.* Fig. 6.8, left): Given a triangle, e.g. $T2$, we copy and extend its two vertices which belong to the current crack front - creating new *temporary* points e.g. $p21$ and $p22$. For convenience, vertices are extended on a triangle-by-triangle basis, so each crack front vertex will have two temporary copies which we later merge (see below). Crack front vertices are extended according to the propagation speed and direction.

To compute the connectivity defining the new crack front, we first merge temporary points, which ensures well-shaped triangles in the generated mesh. We merge pairs of temporary points that are copies of the same vertex which is on the current crack front e.g. $p11$ and $p22$ (*cf.* Fig. 6.8, left). Merging produces new vertices like $v1$ which we compute simply as a mid-point (*cf.* Fig. 6.8, middle). After merging all paired points, we then add edges connecting adjacent mid-point vertices to form the new crack front geometry. Extra vertices may be added in this step to split the new edges if their length exceeds a given threshold, e.g. twice the crack front edge size in the initial disk mesh. Finally, we construct triangles to extend the crack surface by connecting the new crack front geometry (*cf.* Fig. 6.8, right, $T5$–$T9$).

6.7 Rigid-body Contact and Traction Forces

Our XFEM system is integrated with a rigid body solver which is where we get forces from collisions. These forces are essential to the construction of the fracture system. So in this section we describe how rigid body collisions result in traction forces $\mathbf{f}$ which parameterise our linear system in Eq. (6.25).

Once contacts are detected and resolved in the rigid body system, we extract the computed impulses from which we estimate the contact duration and traction forces. To do this we have followed a similar approach as existing methods [44, 49, 79] which use the Hertzian contact model. First, we build a traction field from the collision impulses of the rigid body system, mapping an object's collision points to the closest elements in the XFEM mesh, and add the nodal tractions to each element. As in [49], each collision point from the rigid body system contributes a $I\mathbf{n}_\Gamma/(t_c A_e)$ traction to the input force vector corresponding to the element e. We specify A_e as the outward-facing triangle area of e which has unit normal $\mathbf{n}_\Gamma$ in the object's local coordinate space. Finally, I is the impulse from the rigid body impact, and t_c is the contact duration.

Contact duration t_c for each collision point must be estimated since the impulse I is independent of rigid body time-step. The Hertzian contact model describes the collision of two elastic spheres to estimate this duration. A sphere is described by an effective elastic modulus E, radius R, and mass m

$$E = \left[\frac{(1-\nu_1^2)}{E_1} + \frac{(1-\nu_2^2)}{E_2}\right]^{-1}, \quad R = (R_1^{-1} + R_2^{-1})^{-1}, \quad m = (m_1^{-1} + m_2^{-1})^{-1}.$$

The contact duration is then computed by

$$t_c = 2.87 \left(\frac{m^2}{E^2 R v}\right)^{\frac{1}{5}}, \tag{6.36}$$

where v is the relative velocity between the two objects.

These variables are specified either manually or they may come from the rigid body system. Elasticity parameters and material densities are given as user inputs for each object. Contact point velocities and transferred impulse are extracted from the rigid-body simulation, and the effective radii are computed as the distance between the contact point and the object's centre of mass as in [79]. The volume of each object is computed using its high resolution surface mesh, which is required to compute its mass.

6.8 Results and Discussion

Demonstrative results of brittle fracture simulation are presented in this section - showing complex animations, and comparing with related work. As we focus on the aspect of simulating fracture, additional mesh cutting results are deferred to Chapter 7.

Setup and Implementation

We visualise broken fragments using high detailed meshes but use lower resolution (i.e. simplified) meshes to speed up simulation, tetrahedralisation and mesh preprocessing for collision detection. We do this since simulating (and meshing) on high resolution meshes will generally offer diminishing returns in visible quality with regard to computation time. The lower resolution meshes are created using standard edge-collapse [89] which is implemented with CGAL [15]. Tetrahedralisation is done with TetWild [58] and the high-resolution fracture surfaces are simulated in the generated tetrahedral mesh. We enrich elements and cut the high-resolution meshes with our

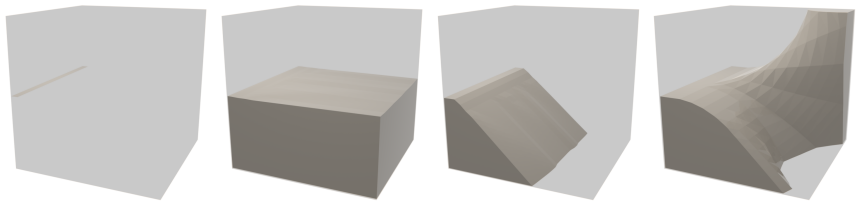
cutting algorithm which is implemented in `C++` using the `Surface_mesh` data structure [125] (refer to Chapter 7 for details).

All results are rendered using Blender [21]. Rigid body dynamics are implemented with Bullet [24], using VHACD [91] for convex decompositions to improve the robustness of collision detection. Our Dirichlet boundary conditions are specified similarly to Müller *et al*. [103]. Impact forces are computed at discrete collision events, where we treat objects as if they are anchored and proceed to compute their static equilibrium response using XFEM.

6.8.1 Fracture Simulation

We now show our fracture simulation results in this section. First, we show results for reproducing standard fracture benchmarks and evaluate against expected crack propagation behaviour according to LEFM. Next, we will show simple adaptations to emulate spatially varying material grain structure, which we use in our complex examples for adding detailed fractures comprised of rigid-body animations.

Benchmark Tests for Crack-front Motion Estimation



(a) Initial crack. (b) Mode-I. (c) Mode-II. (d) Mode-III.

Figure 6.9: *a) planar edge-crack under mode I, II, and III loading. Results: b) mode-I, c) mode-II, d) mode-III.*

Fig. 6.9 shows propagation behaviour according to Irwin's crack loading modes [62] (see also Fig. 6.7). In Mode-I (opening mode), the crack opens perpendicular to the crack plane due to tensile loading. Mode-II causes in-plane sliding. Here the crack faces are displaced on their plane and normal to the crack front, which correlates to a transversal shearing load (*cf.* Fig. 6.7). Finally, Mode-III is the out-of-plane tearing mode where the crack faces are displaced on their plane and parallel to the crack front. Our results are inline with theoretical predictions of LEFM by producing a crack which propagates according to the applied loading.

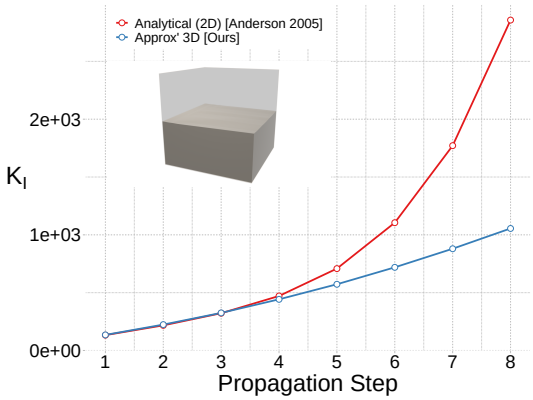
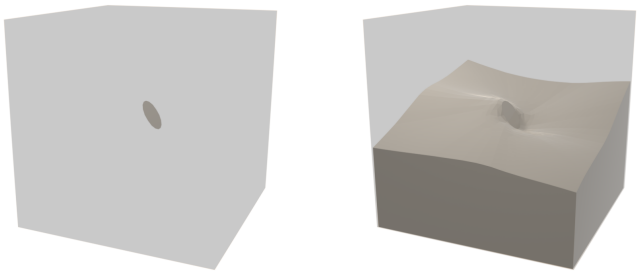


Figure 6.10: *Comparison: our approximation of SIFs vs analytical solution* [4] *for the single edge-notched tension test (Mode-I).*

To assess the impact of our approximation quantitatively, we also compared our stress intensity (in the cross-section) to an analytical solution in 2D for the Mode-I configuration ('edge-notched tension test' by Anderson [4]). Our experimental setup used a cube with edge length 4m consisting of 9987 elements (2048 nodes) and an initial crack surface with 60 faces. The material parameters used were: Loading=100N, Young's modulus=2GPa, Poisson's ratio=0.3, and density=2800 (kg/m^3). Fig. 6.10 plots our estimate (blue) and the analytical solution (red) as the crack propagates (horizontal axis). The result is that our method provides a linear approximation of a quadratic function, which is sufficient for low propagation steps but underestimates the SIF further away from the initial crack. For applications which do not require numerical fidelity, the advantage of our approximation is that it avoids crack-tip enrichment. Crack-tip enrichment, achieves better accuracy at the cost of extra design complexity, increased computation and potential instability arising from the extra degrees of freedom.



(a) Initial crack. (b) Propagation result.

Figure 6.11: *Crack propagation of the inclined crack under vertical loading*

Fig. 6.11 shows our result for the "inclined penny crack" test, which is a crack propagation test often used in engineering to evaluate the validity of the simulation method. A circular surface crack is placed inside a cube with a 45° inclination relative to the axis of principal stress. Our method maintains qualitatively correct crack orientation in all cases. The fracture surface smoothly re-orients itself to be orthogonal to the direction of principal stress which is vertical.

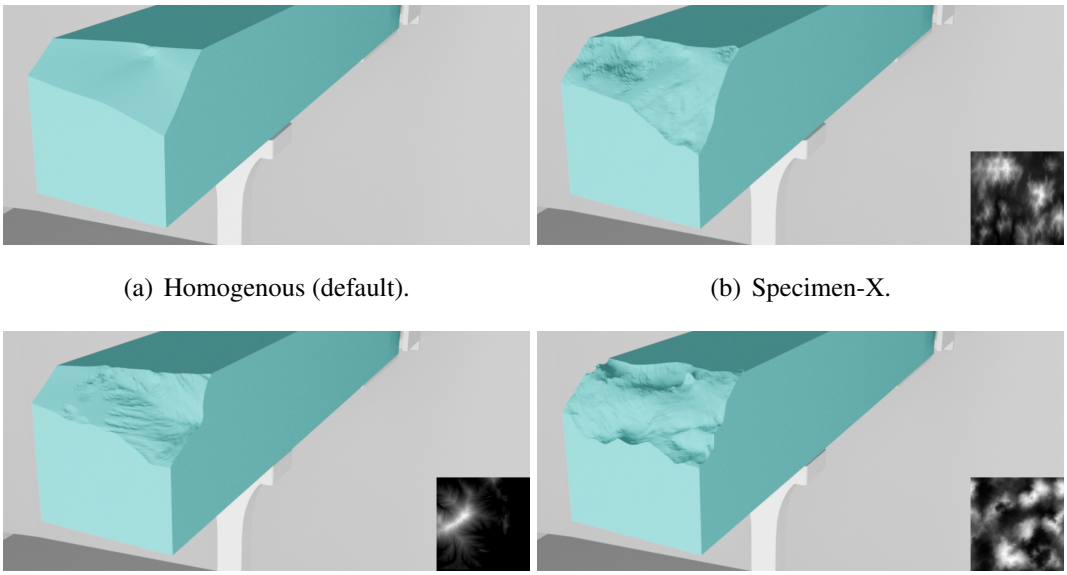
Emulating Inhomogeneous Material

Smooth surfaces in homogeneous material are important for brittle fracture simulation but they may lack the quality observed in real-world materials. So, as our model assumes that the material is homogenous, the produced fracture surfaces are smooth by default which we seek to avoid (*cf.* Fig. 6.11 and Fig. 6.9). To resolve this, we adopt a simple solution by exploiting the fact that toughness fields in material change similarly to a height field. Thus, we emulate spatially varying toughness fields using texture height maps by combining our already high-resolution crack surface meshes with surface-texture parametrisation [28]. Vertices along the crack mesh are then simply displaced using sampled texture values to obtain the distinctive quality sought in a simple fashion. The great benefit of this method is that any rich set of conventional 2D texture height maps can be applied to achieve the fidelity sought from real-world material. An example of our results is shown in Fig. 6.12 (see also accompanying videos for further reference).

In addition, our method inherently supports spatial variations in the elasticity parameters, enabling numerous avenues for biasing crack initiation. Consequently, a spatially varying strength field can also be used to automatically bias crack initiation which can also be used as a tool to control the simulation, as shown in Fig. 6.13. This capability is an inherent strength of finite elements which is in contrast to methods such as BEM [49] where variations in the elasticity parameters require adjusting the fundamental solution.

Comparison against BEM [50]

We performed a thin material breakage test where we compared our XFEM based method with the well known BEM. In the scenario shown in Fig. 6.14 we drop a wine glass on the floor. Because of the ground contact force, the glass breaks into multiple fragments. With our simulation (Fig. 6.14(b)), the entire glass shatters into multiple



(a) Homogenous (default). (b) Specimen-X.

(c) Specimen-Y. (d) Specimen-Z.

Figure 6.12: *Texture-based material grain structure which are used to emulate physically based toughness models. Here we show simple examples where a small fragment is chipped away from a block*

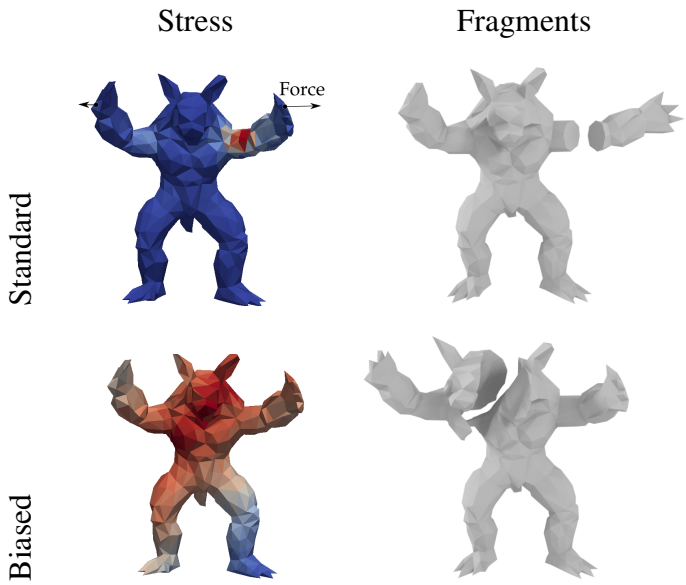


Figure 6.13: *Using material modulation to guide fracture. Considering homogeneous elasticity, high stress is within the inner left elbow (top-left) causing it to crack as shown in (top-right). By using a simple voxel embedding with stress-biasing coefficients, we weaken the middle of the armadillo relative to the rest of its body. This causes the highest stress to be in the chest area (bottom-left) and the crack is initiated this weak region (bottom-right).*

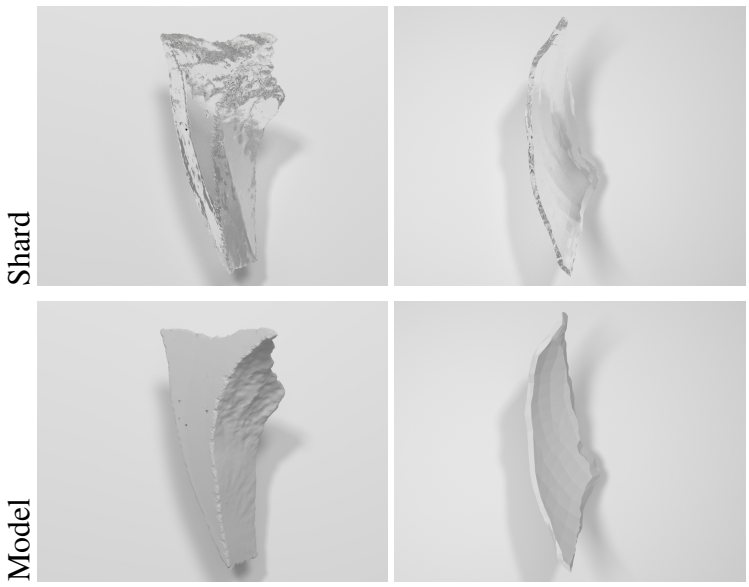
shards. In contrast, for the BEM simulation the bowl is noticeably unbroken resulting in a physically incorrect and even implausible state as cracks are under-resolved (Fig. 6.14(a)).

(a) BEM (b) XFEM (ours)

Figure 6.14: *BEM* [50] *vs our method: Simulating brittle fracture in objects with thin features. Our method breaks the entire wine glass including the bowl and rim which are the thinest parts unlike BEM which breaks only the stem and base.*

BEM can simulate impressive results in objects where the volume-to-surface-area ratio is sufficiently large. However, on objects with relatively larger surface areas the method exhibits limitations in fracture behaviour on thin parts (and always dependent on the meshing details). This limitation has a number of causes including low-order integration, mesh resolution, and related numerical issues. In effect, BEM fractures tend to get 'stuck' and fail to cut off fragments properly (*cf.* Fig. 6.15). Also, we observed a particular susceptibility to a loss of intricate sharp features during cutting (*cf.* Fig. 6.16) due to a dependence on meshing operations with implicit surfaces which can be memory intensive to resolve.

Figure 6.15: *close-up model from Fig. 6.14(a).*



(a) BEM (b) XFEM (ours)

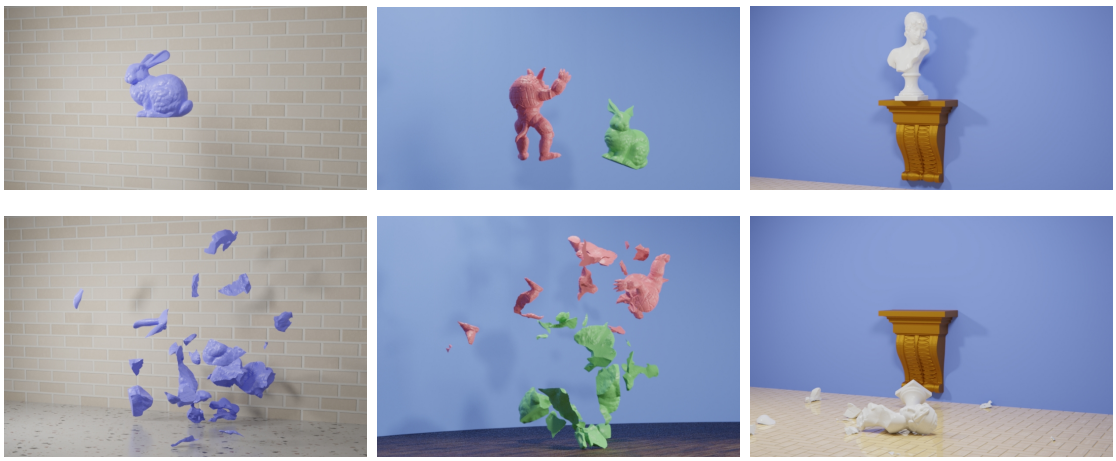
Figure 6.16: *Comparison of shard thickness and sharpness (cf. Fig. 6.14): The wine glass has thin parts which when broken produce shards with sharp edges. Here we show the difference between two example pieces from our simulation and BEM* [50]. *The piece from BEM is comes from the foot of the wine glass since it exhibits the finest breakage, whereas ours is from the bowl which is the thinnest part of the glass.*

Complex Animations of Brittle Fracture

In addition to the comparisons and benchmark examples, we performed four simulations with multiple complex fractures. In the first experiment (*cf.* Fig. 6.17, left), the Stanford bunny is thrown at a wall. Although the object has complex concavities in its shape, the bunny is broken into many pieces. In the second example, we simulated a head-on impact collision between the Stanford armadillo and bunny (*cf.* Fig. 6.17, middle). Here we demonstrate that the rigid body coupling is able to handle fast paced interactions and/or robust collisions with many fragments. The third example (*cf.* Fig. 6.17, right) breaks a heavy marble statue fragmented into multiple pieces which retain their characteristic sharp features.

In the scenario illustrated in Fig. 6.1, we collide a wrecking ball with a statue which is then broken into many pieces. The result is another example of our ability to handle finely structured cuts with sharp edges on our fragments. Moreover, it shows that our method yields realistic results even on a relatively coarse mesh. The tetrahedral mesh is approximately 7k elements while the visualised/cutting mesh has 40k triangles.

Our performance results are shown in Table 6.1 which provides a breakdown of



(a) Bunny (b) ArmBunny (c) Statue

Figure 6.17: *Examples of brittle fracture simulated with our method.*

how long individual components took in each benchmark shown. [8]

Example	M_t	t_{cd}	t_{tet}	t_{sim}	t_{cut}	t_{tot}
Fig. 6.1	40k	121	313	491	802	1539
Fig. 6.12	.3k	27	7	30	135	197
Fig. 6.14(b)	13k	1733	855	1411	329	3721
Fig. 6.17 (a)	8k	863	514	415	795	2471
Fig. 6.17 (b)	18k	788	529	203	365	1843
Fig. 6.17 (c)	16k	493	258	179	552	1437

Table 6.1: *Performance overview: (M_t) triangles defining input meshes (total); (t_{cd}) convex decomposition time; (t_{tet}) tetrahedralisation time; (t_{sim}) fracture simulation; (t_{cut}) mesh cutting time; (t_{tot}) total computation time of the entire simulation (includes I/O, mesh simplification etc.). Timings are given in seconds and measured on an Intel® Core™ i9-7920X CPU @ 2.90GHz CPU.*

6.9 Conclusion

In summary XFEM is a simulation technique used in engineering which is usually tailored by domain experts for each application. We have presented an adaptation of XFEM applicable to brittle fracture simulation in computer graphics. Our algorithm improves the generality, efficiency, scalability and controllability of classical FEM by

[8] These timings are based on our unoptimised and single-threaded implementation.

sacrificing accuracy. After introducing how we add cracks to the linear system, an approach to estimate stress intensity factors to compute crack propagation for XFEM was described. Moreover, an algorithm was presented that keeps the linear system close to the original size. We demonstrated the advantages of using an explicit representation of the crack surface. Further, we showed that our method is able to realistically simulate detailed cracks – even with coarse tetrahedral discretisations.

Limitations: A limitation of our method is that we assume simple topologies for the propagation of the crack front. We do not explicitly handle crack bifurcation ('branching' as in e.g. [27, 99]), and self-intersections which can occur. Despite this, our method can still be used to simulate complex fracture patterns on flat (wine glass) and volumetric (statue) shapes.

Controlling the condition number of the stiffness matrix is an open challenge in XFEM discretisation, which we did not address. Simulation failure can occur if the stiffness matrix is not kept regular, and/or when nodes whose enrichment functions have only small supports in the cut element are not removed (see Fries and Belytschko [39] for a brief discussion).

Finally, as mesh fragments often exhibit complex geometries, robust tetrahedral meshing tools are required to mitigate unpredictable failures during tetrahedralisation (and even simulation). Thus, the stability of simulation is also dependent on robust meshing tools (e.g. TetWild [58]) to cater for arbitrary shapes with intricate geometries like those produced by our cutter (Chapter 7).

6.10 Postscript

In this chapter, we presented a technique for simulating brittle fracture by tracking crack fronts in the deformation mesh. We also include a number of useful details for this procedure to be possible including how we perform enriched stiffness matrix construction, rigid body coupling, collision detection, and tetrahedral meshing. Our simulation technique builds on the displacement correlation technique originating from computational mechanics to estimate stress intensity factors from the computed displacement field. This method is useful for many physically based animations because it offers the benefits of XFEM but without nuisance of dealing with crack tip enrichment functions. In this way, it provides a basis for further work to mitigate the drawbacks of FEM with counterparts which have less requirement for re-meshing during

crack propagation and are not constrained to specific volumetric domains.

In Chapter 7, we follow up to describe the cutting algorithm which was used during fracture simulation. More specifically we show how the simulated crack surface is then used to cut the domain mesh for producing fragments.

Chapter 7

Cutting Polygon Meshes

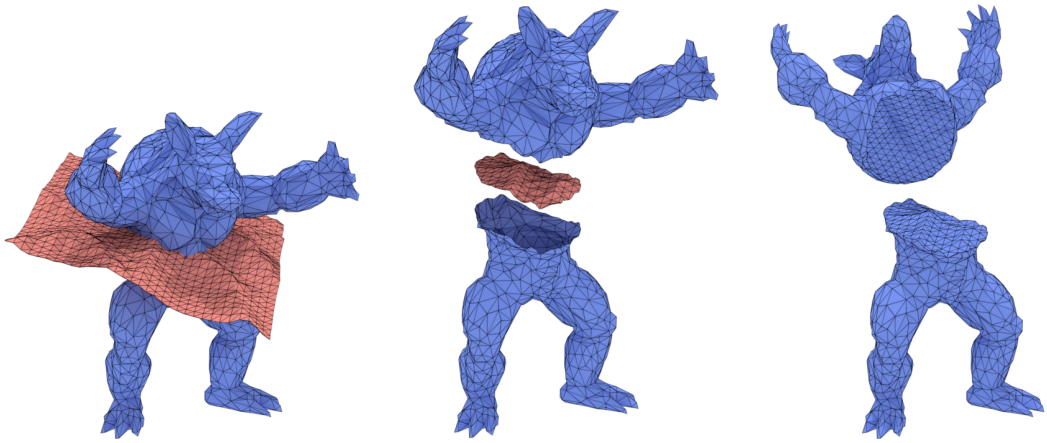


Figure 7.1: *Mesh cutting example: An armadillo mesh is cut into two pieces. The resulting meshes (right) are identical everywhere except at the edges introduced by the cut.*

7.1 Preface

This chapter will describe the second contribution of this part of the thesis which was used during simulation of the results shown in Chapter 6. Thus, we will describe about our novel cutting algorithm that can be applied to arbitrary planar polygons for representing surface meshes to alleviate requirements for explicit triangulations or volumetric decompositions when cutting. The algorithm is a practical solution to the problem of resolving complex intersections between meshes, which is a nuisance to implement. The method is particularly attractive as it also generalises to a number of topological settings (e.g. two-dimensional cuts and even boolean operations) enabling

use-cases that are within–and even beyond–the scope of immediate application. In practice, we use this method to cut finite element geometry for enrichment purposes, as well as cutting arbitrarily-shaped surface meshes which define the domain to obtain new fragments for simulation, collision detection, and rendering.

7.2 Introduction

Mesh cutting is a fundamental geometric problem which appears across a wide set of applications. The task of cutting is to partition a given volumetric domain or surface, described by its boundary into a set of disjoint connected components or fragments. The resulting fragments are typically employed for further model design and/or simulation, such as virtual surgery, computer aided design, and fracture simulation. Many such geometric tasks also exploit volumetric subdivisions, such as tetrahedral decomposition [156], uniform voxel grids [29] and implicit representations [104, 108] to lessen the difficulty of the task in terms of implementation and robustness.

Implicit surfaces, particularly the level set method (LSM) [108], are a popular choice in computer graphics [54, 104] as well as engineering [130, 131] applications. While the LSM provides an elegant space for boolean operations which facilitate cutting, scan conversion which transforms the polygon mesh to the level-set equivalent has several disadvantages: It may result in a loss of sharp features and resolution, while incurring significant memory costs due to voxel-based discretisation. In addition, the inverse task of mesh extraction (say via dual contouring) modifies the cut mesh globally – even in regions of the mesh that should remain unaffected by the cut.

As an equally popular strategy, explicitly modelled surfaces using e.g. triangulated geometry [127], offer a number of benefits, including a guaranteed preservation of volume and geometric detail on the cut model without incurring significant memory costs. Thus, there are no imposed restrictions on the geometry of the cuts. These methods operate by intersecting and clipping polygons of an input boundary mesh against the cutting surface. The effective goal is to ensure that all intersecting input polygons are resolved into an intersection-free polygon mesh containing the connected components that retain geometric detail. However, a new problem of numerical robustness arises whereby restrictions on floating point precision during geometrical operations may lead to unexpected runtime failures.

In general, there exists no single answer to the question of which strategy is best for all applications. Choice depends rather on the specific application and its requirements

- which in our case happens to be the explicit method. This is because our fracture simulation pipeline (Chapter 6) is heavily dependent on explicit meshes at different stages - e.g. for crack propagation and enrichment.

The general design of an *explicit* cutting algorithm is fairly simple to describe but has essential criteria which must be satisfied for practical use. As we focus efforts on averting runtime failures, the algorithm should be suitably *robust* to degenerate scenarios arising from geometries "in the wild" and those resulting from simulations as seen in Chapter 6. In addition, it also ought to be *general* enough to support: 1) meshes with open boundaries; 2) extensions boolean to operations; and 3) arbitrary polygonal subdivisions which include concave polygons. The latter point is particularly important as previous methods are underpinned by the assumption that the meshes under consideration are triangulated. Such an assumption causes to severe degradation of meshes when applying subsequent cuts. Triangulation is not unique and may thus be found to be suboptimal with the introduction of incremental cuts. In effect, one would be forced to undo previous triangulations in order to avoid degeneracies due to newly introduced cuts, thus hindering the incremental nature of a robust cutting algorithm. One must of-course triangulate for rendering, collision detection etc. but the resulting bad-shaped triangles are not used in further cutting. The difficulties of meeting these constraints while achieving good performance has lead to a number of methods, which are discussed in Section 7.3

7.2.1 Contribution

Our contribution is an explicit mesh cutting tool to partition a given mesh into one or more components. The algorithm is general and robust. It is general because it permits any manifold cutting surface defined using arbitrary polygonal subdivisions - likewise for the mesh to be cut as well[1]. Further, it is robust by deferring numerical floating point operations only to the task of computing intersection points between polygons. The remaining parts of the algorithm can then rely purely on the combinatorial structure of the intersecting meshes to find the final connected components. In effect, the connectivity of the resulting fragments is identical to the (uncut) input mesh except at edges introduced by the cut as shown in Fig. 7.1.

Organisation: The rest of this chapter proceeds as follows: We first review related

[1]This generality also implies that e.g. client applications (as in Chapter 6) are not restricted to triangular meshes. Thus, cracks may be constructed with any polygons desired, including concave polygons.

methods in Section 7.3. An overview of our method is then provided Section 6.5. Section 7.5 then proceeds to describe how we resolve and store polygon intersections and how new edges connecting intersection points are computed. In Section 7.6, we describe the details of clipping intersecting polygons using the created edges. Section 7.7 then follows to describe the steps to partition the cut mesh using the newly traced polygons after clipping, and also how we seal fragments to produce watertight meshes. Our results are presented in Section 7.8 and we conclude the chapter in Section 7.9.

7.3 Related Methods

Attempts to satisfy objectives stated in Section 7.2 (in addition to performance) has lead to a number of algorithms. In this section we will briefly discuss some these related methods for cutting meshes. We also refer the interested reader to Wu *et al*. [164] for a survey on mesh cutting in computer graphics.

7.3.1 Piecewise Linear Cuts

One class of cutting algorithms applies piecewise-linear cuts [79, 97, 115, 145, 156]. These algorithms use a volumetric (e.g. tetrahedral) decomposition and duplicate vertices along *predefined element faces* that are most aligned with the cutting surface. While relatively simple, this approach can be computationally costly for cut surfaces with geometrically complex patterns due to necessary mesh refinements [79, 115]. In addition, there is potential for a lack of preservation in volume and mesh-scale [156] since they operate using volume-refinement. (Similar issues exist for methods based on regular grids [29, 68]). The recent work of Wang *et al*. [156] is worth mentioning: Wang *et al*. have redeveloped the virtual node algorithm (VNA) [97] to allow for cuts passing through vertices or even those lying on edges and faces of tetrahedra. A great strength of their approach is that it permits multiple cuts through a tetrahedron - unlike the original VNA. Thus, their method offers a solution which is competitive due to robustness and with lower complexity than e.g. Sifakis *et al*. [127]. Unfortunately, this method–like its predecessors–is dependant on a volumetric decomposition.

7.3.2 Explicit Boundary Mesh Methods

Mesh evolution algorithms can overcome piece-wise linear cutting limitations given their ability to preserve volume, mesh-scale detail, and sharp features [26, 161]. How-

ever, these are not directly applicable to our problems of interest such as fracture. Da *et al*. [26] present a multi-material triangle mesh-based surface tracking scheme for evolving fluid interfaces which has been used for fracture simulation by Zhu *et al*. [166] but with significant modifications for tracking the crack surface. Sifakis *et al*. [127] present an influential method to cut tetrahedralised meshes with arbitrary incisions, along with novel edge placement rules for reconnecting topology. The method works well but assumes a tetrahedral representation and is thus only applicable to triangulated meshes. We share broad similarities with Sifakis *et al*. [127] but do not assume triangulated cut geometry in a tetrahedralised domain ("embedded meshing tool"). In addition, they omit critical details of the boundary tracking procedure necessary to produce fragments which we explore in detail, and their method also assumes that cuts are introduced in triangle form.

Averting arbitrary failures which arise due to numerical error when cutting is a common challenge as seen in constructive solid geometry (CSG). Zhou *et al*. [165] present a technique for boolean operations using mesh arrangements which is robust and preserves original mesh geometry. Their method uses exact arithmetic and extends the space-partition view of boolean operations [12] which has also been used within meshing tools like TetWild [58]. Wang *et al*. [156] also describe a practical technique for robustly computing the intersection of triangle meshes. Their approach offers a way to robustly define the intersections of a cutting triangulated surface against a tetrahedron mesh by checking for intersections in the order of most degenerate to least.

7.3.3 Implicit Methods Using Voxel Grids

A third class of approaches converts polygonal meshes to an intermediate sparse voxel representation (e.g. OpenVDB [104]) for cutting before re-triangulation, which is useful for avoiding the numerical errors of explicit methods. These require multiple representations of the cutting surface and are useful when the resolution of the cutting surface affects simulation performance as seen in BEM [49]. Voxel representations are also used when rendering meshed particles but may require trade-offs between surface smoothness and sharpness [162].

It is also worth mentioning cut-cell mesh generation tools which, although intended for mesh generation, may have use for cutting tasks. The recent Mandoline tool by Tao *et al*. [143] is one such example wherein mesh generation occurs by embedding a triangle surface mesh into an adaptive voxel grid to produce a Cartesian cut-cell

mesh. However, while their method gives details about reconnecting mesh topology, these operations are dependant on floating point arithmetic as in previous engineering literature [67, 88], which we avoid. In contrast, our method is fundamentally motivated by the desire to partition meshes using infinitely thin cut surfaces i.e. the cut surface can be an open mesh. Conversely, Mandoline [143]'s fundamental task (as a cutting problem) is the boolean operation. Thus, emulating a "cut" with Mandoline amounts to cutting away a thin strip of material from the input mesh which leads to slight volume loss similarly to what is seen in [49] when using OpenVDB [104] (*cf.* Fig. 6.15).

7.3.4 Methods in Engineering

Lastly, engineering solutions, which are similar to Sifakis *et al*. [127], also discuss the problem of polyhedral rock-block identification using simplicial homology for identifying meshes (blocks) created by intersections of planar surfaces [34, 60, 88]. These methods systematically identify blocks by defining and collecting the corresponding geometry data where blocks are regarded as "oriented complexes" [67]. However, identification (polygon clipping) depends directly on geometry (floating point) operations like Mandoline [143] to determine inner and outer facing polygons since intersections are resolved simultaneously with many cutting surfaces considered. Further, these methods can be complex, often requiring additional post-processing for filtering vertex, edge and face data as shown by Jing [66] and Jing and Stephansson [65].

7.4 Method Overview

Having described some related methods in Section 7.3, we will now introduce our algorithm - starting with an overview in this section.

The input to our cutting algorithm is a pair of mesh data structures for an object $\mathcal{M}$ and a cut surface $\mathcal{C}$. The output is a set of mutually exclusive fragment meshes which are a result of the cut. We make the following simplifying assumptions about each of these meshes: 1) that there are no self-intersections; 2) that a mesh is composed of simple polygons (which can be concave); 3) that improbable cases such as the intersection of two edges or a vertex intersecting a plane are extremely rare[2]; 4) that meshes are manifold (edges are incident to at-most two faces). An illustration of the pipeline

[2]Note: we did not encounter this problem in examples shown in Chapter 6, which are geometrically complex cases due to fragment shapes and sizes.

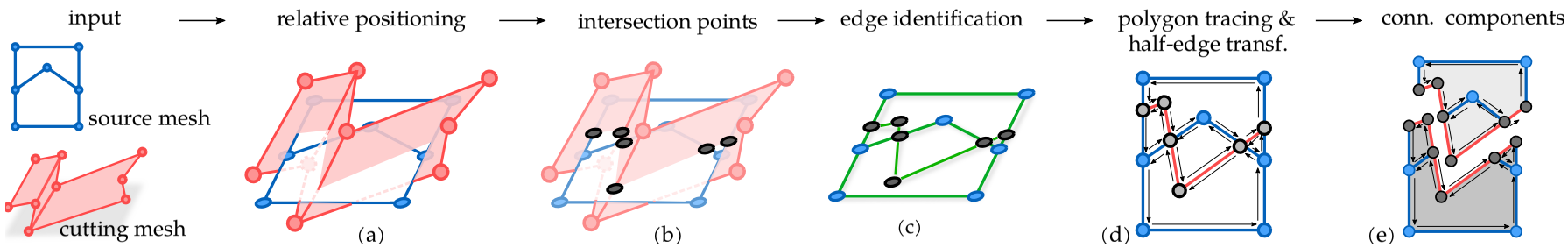


Figure 7.2: *Our cutting algorithm consists of four different stages (b-e). We test all pairs of intersecting polygons (using a bounding volume hierarchy) and insert intersection points, as well as edges between these points, into a half-edge data structure. Then, we traverse all edges in the data structure, duplicate nodes when required and update edges to these nodes.*

in 2D is shown in Fig. 7.2: we use a 2D illustration since it is difficult to visualise the effect of operations on 3D meshes in a static figure.

The algorithm can be implemented using any standard manifold mesh data structure, (e.g. vertex-face adjacency list) but the halfedge mesh is most convenient since it supports maintaining incidence information of vertices, (half)edges and faces.

7.5 Polygon Intersections

In this section, we describe how we resolve intersections between an input mesh and the cut surface, which is the initial phase of our algorithm. First, we describe how we compute and store intersections points between polygons in Section 7.5.1. Section 7.5.2 then describes how we create the set of edges which connect intersection points for later - to clip intersecting polygons as described in Section 7.6.

7.5.1 Calculating Intersection Points

We now describe the first step in our algorithm, which is to compute new points that result from intersecting polygons.

In order to compute these *intersection points*, we resolve all intersections using one polygon soup $\mathcal{P} = \mathcal{M} \cup \mathcal{C}$ for convenience (*cf.* Fig. 7.2 (a)). Thus, intersection points are computed by testing the halfedges of each polygon in $\mathcal{M}$ against each polygon in $\mathcal{C} \in \mathcal{P}$ and using standard point-in-polygon tests [51] (*cf.* Fig. 7.2 (b)). Our mesh vertex coordinates are assumed to be rational coordinates, thus intersection points are computed exactly [165]. In general, we impose no restrictions on the numerical representation of intersection points, so static filtered floating-point predicates can also be applied using the binary space-partition view as in [12] (see also [123]).

In practice, we speed up this process with a bounding volume hierarchy (BVH), with each leaf node containing one polygon. In our implementation we used the binary ostensibly-implicit tree which is described in Chapter 3.

Intersection Registries

To solve the boundary tracking problem of clipping intersecting polygons *without* numerical operations, we will need to keep track of the set of polygons which meet at each point of intersection (*cf.* Fig. 7.3). Thus, we create an *intersection registry*, wherein

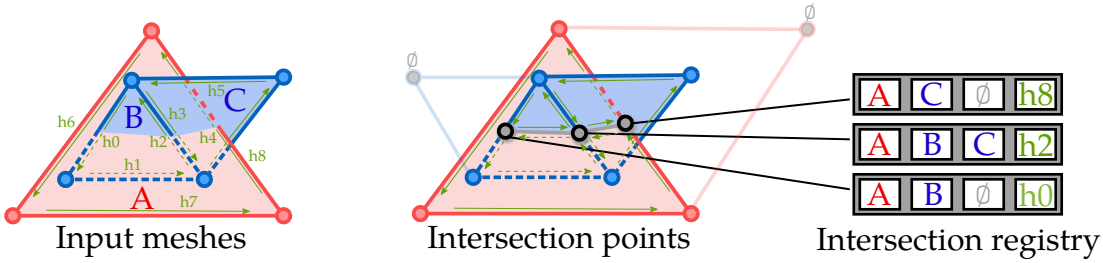


Figure 7.3: *The intersection of multiple polygons represented using a registry for uniquely identifying each intersection point.*

we store an entry for each intersection point. A registry entry is the set of polygons (from $\mathcal{M}$ and $\mathcal{C} \in \mathcal{P}$) that the intersection point lies on.

We adopt the method of Sifakis *et al.* [127], but extended to also include the halfedge which intersected a face to give the intersection point: The intersection between an edge pq and a polygon xyz is registered as the intersection of polygons pqr, pqs and xyz. We also introduce a virtual polygon $pq\emptyset$ if pqr is the only polygon incident on pq, $\emptyset$ refers to an auxiliary virtual point which is unspecified. Thus the same intersection point is encoded as the intersection of pqr, $pq\emptyset$, and xyz. Finally, the tested halfedge belonging to pq is also stored as part of the registry in addition to the intersecting faces. This halfedge is used during various sub-stages, including edge creation and more.

Storage: At this stage, we can assume–without loss of generality–that a new mesh data structure $\mathcal{P}'$ is created which will store copies of all *non-intersecting* polygons in $\mathcal{P}$ from $\mathcal{M}$ and $\mathcal{C}$, as well as the new ones after clipping. Thus, all new vertices computed as intersection points are stored in $\mathcal{P}'$. Then, the registries of intersection points can be used to determine whether the intersection of three given polygons has already been registered (possibly as the intersection of three different polygons) to prevent duplicates. This means that we store an intersection point into $\mathcal{P}'$ only if its registry entry is unique. [3]

7.5.2 Edge Identification

Now that the points of intersection between the polygons of $\mathcal{M}$ and $\mathcal{C}$ have been determined, the next step is to create new edges that will be used to clip the intersecting polygons.

In general, edges are created by connecting the intersection points of each pair of

[3] Keep in mind that that two opposing half-edges will be tested against the same polygon twice because intersections are fundamentally done on a polygon-by-polygon basis.

intersecting polygons between $\mathcal{M}$ and $\mathcal{C}$, using the intersection registry (*cf.* Fig. 7.2 (c)). Also, new edges are added between an intersection point and, an original vertex from an edge which intersected a face to split this intersecting edge. These new edges are stored in $\mathcal{P}'$.

Detail: The details of edge creation are as follows: we add an edge between two intersection points if their registry entries match by *at-least* two polygons. If there are two points on the pair of intersecting polygons then we simply connect them, otherwise we sort and connect according to the order of the sorting. Since the intersection points are guaranteed to be collinear (the polygons are planar), we add edges consecutively in the order of the computed connectivity. Finally, we identify edges between points which lie on the same edge from $\mathcal{P}$ (i.e. in the original mesh of $\mathcal{M}$ or $\mathcal{C}$) while ensuring that we create a minimal set of non-overlapping edges as described above.

7.6 Polygon Clipping

In this section, we describe how new polygons (i.e. 'child' polygons) are created by clipping the intersecting polygons in $\mathcal{P}$ (*cf.* Fig. 7.2 (c)). The process of clipping a polygon is performed in two steps: 1) gathering the minimal set of halfedges necessary to trace child polygons (Section 7.6.1), and 2) tracing the child polygons (Section 7.6.2).

Keep in mind that our overarching goal is to recast the problem of determining halfedge connectivity as a combinatorial one - using only topological data (halfedges) to trace child polygons. Thus, polygon tracing will be preceded by a number of filtering steps to exclude unnecessary halfedges that are coincident to a clipped polygon. Filtering is needed to confine the problem of tracing child polygons which, in effect, avoids ambiguities when selecting halfedges during tracing (see below).

7.6.1 Gathering Halfedges

We now describe how halfedges incident to an intersecting polygon are gathered in this section. The gathered subset of halfedges will be used for tracing which is described in Section 7.6.2.

For each intersecting polygon in $\mathcal{P}$: we first gather all edges in $\mathcal{P}'$ whose defining vertices coincide on this polygon. We gather edges by associating them with polygons, using the intersection registry described in Section 7.5.1: An edge connects two points

which–if they are intersection points–will have a registry entry that specifies on which polygons they lie.

These gathered edges are then classified as either *interior* or *exterior* for filtering purposes. An interior edge is one which connects two intersection points whose registry entries match by at least two polygons, but on a different edge which is determined from the halfedge in the registry entry. An example of interior edges is shown in Fig. 7.3 (the grey edges connecting new points). The remaining edges in the gathered set are–by definition–exterior (i.e. those which lie on the polygon boundary).

After classification, we collect all the halfedges from the gathered edges by querying $\mathcal{P}'$ [4], and then we eliminate a subset of these halfedges which cannot be used to trace child polygons. This eliminated subset is comprised of 1) halfedges incident to exterior edges, whose winding order (direction) is opposite to that of the clipped polygon, and 2) halfedges incident to interior edges that are guaranteed to lay outside of the clipped polygon as shown in Fig. 7.4 ("filtered interior edge") where polygon 'A' (blue) is to be clipped .

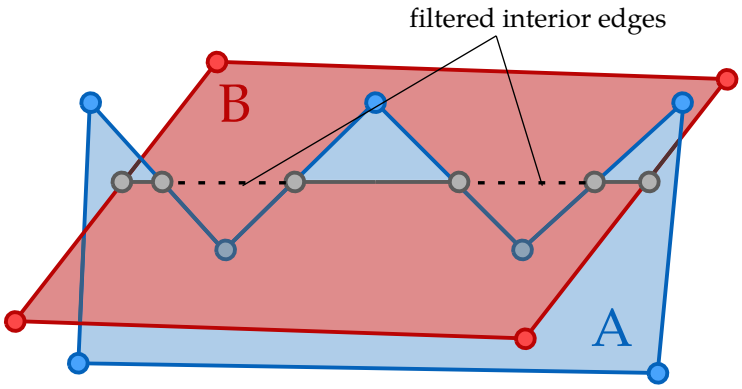


Figure 7.4: *Filtering interior edges from a gathered set which lie on a clipped polygon. For polygon A, the dashed edges are eliminated from the gathered set since they lie outside the polygon. For polygon B, all interior edges (including those dashed) are used for tracing.*

Filtered interior edges: The latter, i.e. filtered interior edges, are a little more complex to identify and so justify further explanation: These arise due to intersections with concave polygons which are on the border (at-least one edge is incident to only one face). Furthermore, these edges are *generally* identified as every-second edge from a sorted sequence of edges whose vertices are intersection points which have registry

[4] An edge in a halfedge-mesh data structure is associated with two opposing halfedges.

entries that match by at-least two polygons. The notion of sorting here implies matching adjacent edges according the vertices they connect (see Fig. 7.4, polygon A, blue).

These interior-edge sequences are constructed incrementally: We start by explicitly mapping each intersection point with interior edges that connect to it, which can be either one or two edges. This mapping will also identify two points that are associated with exactly one interior edge, which are the start and end points of the sequence. We pick any one of these two points as the first vertex–and thus, edge–from which to start constructing the sequence passing through the polygon. Then, we incrementally find the next interior edge by matching vertices using the mappings we created.

Finally, we partition each sequence into sub-sequences. A sub-sequence contains edges connecting intersection points that share one polygon in their registry entries, where the shared polygon (real or auxilliary) is not the one being clipped. If all points share the same polygon then the final sub-sequence is same to the partitioned sequence. Otherwise a sub-sequence will have one or more edge.

For a given sub-sequence, all its edges will be used to clip the current polygon if 1) it has one edge, or 2) the registry entries of at-most two referenced intersection points contain a halfedge on the border. This means that the sub-sequence forms a concrete set of edges along which the clipped polygon will be split. Otherwise, we filter by removing every-second edge from the sub-sequence. This is shown more clearly in Fig. 7.4, polygon 'A' (blue), where two edges are removed. Notice also that if we assume that polygon 'B' (red) is the source-mesh, then we can see that it can never be split because the cut-mesh (polygon 'A') does not fully cut it in two.

Once the unnecessary (half)edges have been eliminated, we proceed to tracing child polygons as described next (Section 7.6.2).

7.6.2 Tracing

Having described how we gather our minimal set of halfedges in Section 7.6.1, we now describe how we trace child polygons to clip the intersecting polygon.

Our tracing procedure is analogous to the boundary tracking problem (e.g. the directed-body concept [59]) but we can now use only the gathered halfedges to trace each new polygon. The steps of clipping a polygon are provided in Algorithm (4): We start with any halfedge in the gathered set as the 'current halfedge', and then iteratively search for the next, until a valid loop-sequence of halfedges is constructed.

To find the next halfedge in a sequence, we first search for a list of candidates as

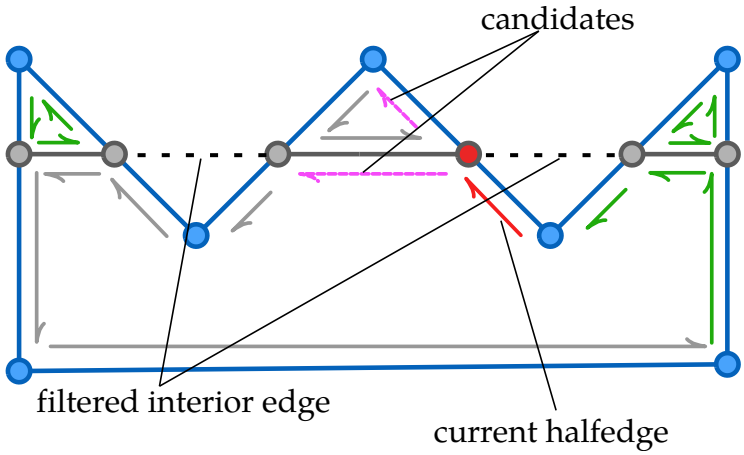


Figure 7.5: *Selecting the next halfedge during polygon tracing. The blue (outer) edges outline the polygon being clipped.*

shown in Fig. 7.5. Candidates are halfedges that have not yet been used to construct a loop sequence (valid or invalid), and whose source vertex is the same as the target of the current halfedge. There can be at most two candidates - thanks to our previous filtering steps. Thus, we initially select any candidate as the prime so long as there is at least one to pick from. When there are two candidates, we then proceed to select whichever is marked as 'interior' if the current halfedge is marked as 'exterior'; otherwise, we pick whichever of the two candidates that is not the opposite of the current halfedge as the next one. In general, the constructed loop sequence is considered valid if the source vertex of the first halfedge is the same as the target of the last halfedge. Our tracing solution thus incorporates both Sutherland-Hodgman [136] and Weiler-Atherton [157] polygon clipping under a single representation and in a three-dimensional setting.

7.7 Mesh Separation and Stitching

So far, we have described how to calculate intersection points (Section 7.5.1) and connect them (Section 7.5.2), so that we could then clip the intersecting polygons as described in Section 7.6. In this section, we proceed to describe steps for how we separate the input mesh along the cut surface in Section 7.7.1, and then how we seal the holes which arise within resulting fragments in Section 7.7.2.

Algorithm 4: Halfedge polygon tracing

```
Function get_candidates(halfedge_list, vertex):
    candidates ← ∅
    for halfedge in halfedge_list do
        if source_vertex(halfedge) == vertex then
            candidates.add(halfedge)
    return candidates

Function trace_polygons(...):
    // Gathered set of halfedges after filtering (input)
    halfedges ← ...
    // Traced polygons (output)
    polygons ← ∅
    // Trace all polygons
    do
        poly ← ∅
        curr ← NULL
        next ← halfedges.any()
        valid ← false
        // Trace one polygon
        do
            curr ← next
            poly.add(curr)
            next ← NULL
            halfedges.remove(curr)
            if poly.number_of_halfedges() ≥ 3 then
                if poly.first().source() == poly.last().target() then
                    cond0 ←curr ≠ poly.first().opposite()
                    cond1 ← curr == poly.first().opposite() AND halfedges.empty()
                    if cond0 OR cond1 then
                        valid ⟵ true
                        break
            candidates ← get_candidates(halfedges, curr.target())
            prime ← NULL
            if candidates.size() > 0 then
                prime ← candidates.first_halfedge()
            if candidates.size() == 2 then
                if halfedge_is_exterior(curr) then
                    prime ← select_interior_halfedge(candidates)
                else
                    if curr.opposite() == prime then
                        prime ← candidates.second_halfedge()
            next ← prime
        while next ≠ NULL
        if valid then
            polygons.add(poly)
    while halfedges.size() > 0
```

7.7.1 Mesh Partitioning Using Halfedge Transformation

Given the polygons of $\mathcal{M}$ and the new edges along its cut surface, we identify which intersection points need to be duplicated so that they appear on exactly two sides to partition resulting fragments (refer to Fig. 7.6).

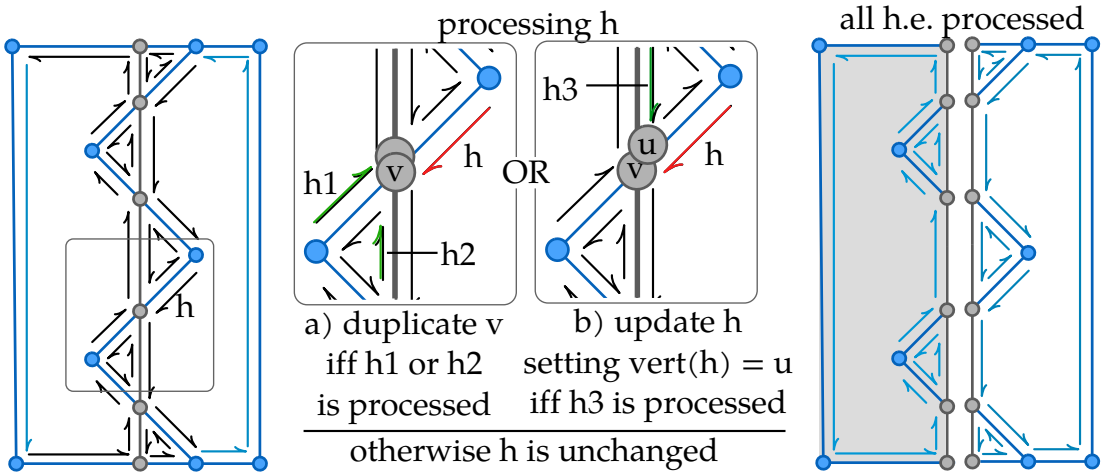


Figure 7.6: *The intersection points (grey) due to a vertical cut (grey line) of two concave polygons (blue), are shown along with all half-edges (arrows). All new half-edges (black) are traversed sequentially to determine whether duplicate vertices are required.*

For this task, we iterate through all half-edges h of $\mathcal{M} \in \mathcal{P}'$ which are from the newly inserted edges, and process them individually using the following three conditions: 1) If h is incident on a vertex v and another halfedge on the same side as h (across the cut surface) and incident on v has been 'processed', then we update the connection of h to the correct instance of v so that it matches the other halfedge. 2) If this is not the case, we check if another halfedge connected to v on the opposite side of h (across the cut surface) has already been 'processed', in order to duplicate v. 3) If neither of the first two conditions are satisfied, then we leave h as is. Finally, we mark h as 'processed' and repeat the process on next halfedge of h. In practice, we traverse through halfedges one intersecting polygon $\in \mathcal{P}$ at a time.

7.7.2 Polyhedron Fragment Sealing

Since the cutting is performed on a (hollow) surface mesh, the fragments of $\mathcal{M}$ will not be closed just yet. For example, Fig. 7.7 (b) shows the bottom fragment of a cube cut by an elliptical mesh. The resulting fragment is a cuboid without the top face. To fix this, we construct polygon patches (similarly to Zhou *et al.* [165], and Mei and Tipper [94]) which are used to seal the fragments of $\mathcal{M}$. A patch is constructed as a

set of adjacent polygons from $\mathcal{C}$, which is bounded by a closed sequence of halfedges forming an oriented loop and passing through intersection-points.

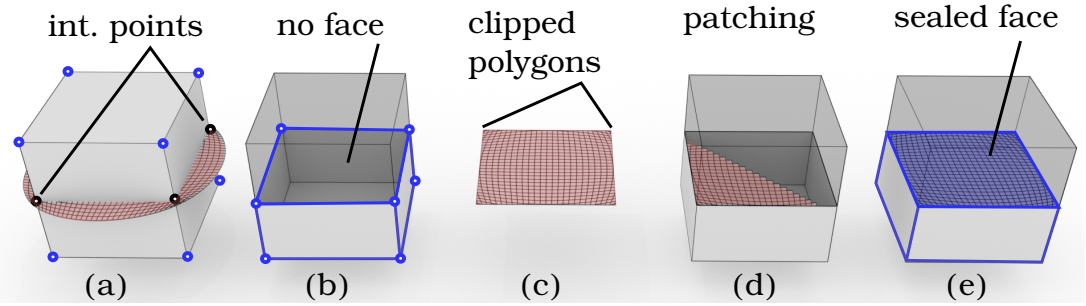


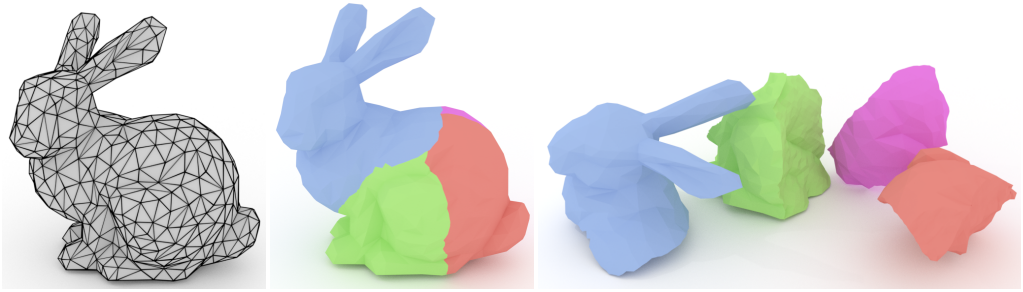
Figure 7.7: *Stitching the polygons of a cut-surface patch to a connected component of the input-mesh (cube).*

We start from any *new polygon* from $\mathcal{C} \in \mathcal{P}'$ which is coincident to at-least two intersection points (adjacent to cut-path) and use breadth-first search (BFS) to identify all patch-polygons. During BFS, we build a patch iteratively, advancing an edge front that grows until all polygons of the patch are identified and added. Advancing amounts to finding polygons which are adjacent to the current front and updating the front with each newly inserted polygon. Two polygons are adjacent if one of them contains a halfedge whose opposite is in the other polygon.

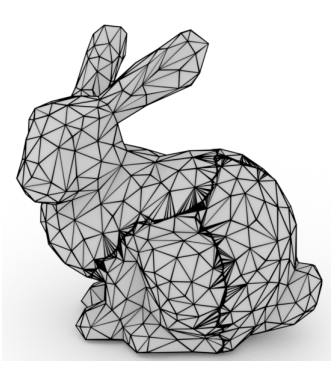
Once constructed, a patch is then duplicated to create a copy whose polygon winding order is reversed (e.g. clockwise) and it is then stitched to a fragment e.g. in the top fragment of the cube in Fig. 7.7. We stitch each patch by matching its bounding halfedges with the border halfedges of each fragment of $\mathcal{M}$ – inserting patch polygons one-at-time as shown in Fig. 7.7 (d). The final polygon-soup contains multiple connected components (*cf.* Fig. 7.2 (e)) which we determine using a standard algorithm [23].

7.8 Results

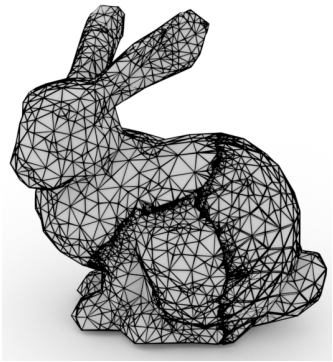
We present the results of our algorithm in this section, which are an addendum to what is shown in Chapter 6 since the same algorithm was used then. The algorithm is implemented in `C++` using the `Surface_mesh` as our halfedge mesh data structure [125] which is provided with CGAL [15]. We first compare against similar mesh-based approaches and then show examples highlighting the generality of our approach.



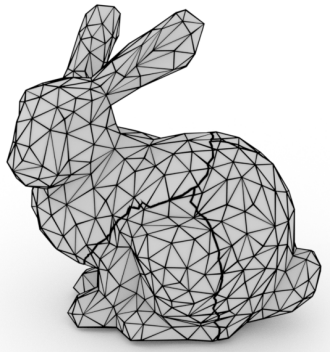
(a) Setup



(b) Sifakis *et al*. [127]



(c) Wang *et al*. [156]



(d) Ours

Figure 7.8: *Mesh cutting evaluation scene where a bunny is cut into four pieces. See also Table 7.1.*

Cutting Method	*Repr.*	*#Vertices*	*#Edges*	*#SurfacePolys*
Sifakis *et al*. [127]	triangles	3420	10236	6824
Wang *et al*. [156]	tetrahedra	25581	22300	84692
Ours	N-gons	3384	8226	4850

Table 7.1: *Surface mesh cutting comparison: Using the scene shown in Fig. 7.8, we show the size (total) of mesh data (geometry and topology) which is to stored in order to cut the fragment meshes for each method.*

7.8.1 Our Method *vs.* Sifakis *et al.* [127] and Wang *et al.* [156]

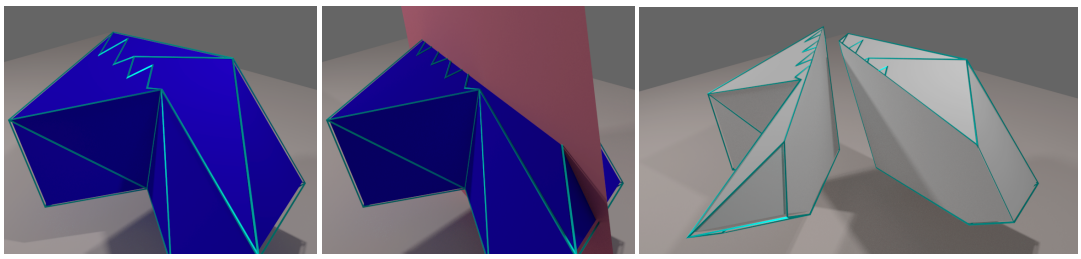
We compared our cutting results with state of the art [127, 156], with respect to the number of triangles and connectivity. For this, we cut a Stanford bunny mesh using four different cutting surfaces (*cf.* Fig. 7.8). Table 7.1 summarises the results. Compared to Sifakis *et al*., our method reduces the number of mesh edges and cutting elements (boundary facets) by 20% and 30% respectively. Our algorithm also produces fewer polygons and edges compared to Wang *et al*. [156], and requires at least 17× fewer cutting elements (tetrahedra in their case) than theirs since we use a boundary representation of the cutting mesh. The visual quality of the meshes corresponding to the results in the Table 7.1 can be seen in Fig. 7.8. Our code can be upto several orders of magnitude slower when compared to e.g. Wang *et al*., but we emphasis that performance is not the focal point of this work and it is reserved for further work. The source is complex and very large with over 7000 lines of code, and without any optimisations or multi-threading considered.

7.8.2 Concave Polygons and Polyhedra

We demonstrate our cutting algorithm's ability to cut concave polyhedra (*cf.* Fig. 7.9) and polygons by cutting a remodelled pentagonal frustum (blue) with a large quad (red). The pentagonal faces are modified so that there is a concavity, rotated so that they are not parallel to each other and divided into polygons with many concavities. The whole model is composed of only one volume element (all edges are on the surface). Our algorithm produces the correct surface meshes for the fragments (white), and does not modify the connectivity on each except where intersected with the cutting surface.

7.8.3 General Examples

In addition, we show our cutting algorithm's ability to handle more general examples including partial cuts, 2D cuts and 3D boolean operations while still operating directly on the halfedge data structure. In Fig. 7.10, we show a 3D partial cut where the input mesh is a cube with convex faces (no triangulation), and the cutting surface is composed of two triangles. Our algorithm correctly computes one output component with an incision-cut along three faces where the interior is sealed to form a water tight mesh (right). We show 2D cutting in Fig. 7.11. In this example, our algorithm correctly cuts a surface mesh into two components with abutting convex and concave polygons.



(a) input mesh (b) cutting plane (c) output

Figure 7.9: *An extreme example. The input mesh (blue) is a pentagonal frustum where the pentagons (top and bottom faces) have been made concave (and are not parallel to each other). Each pentagon is composed of polygons with several concavities. Our cutting algorithm correctly produces two fragments (white) whose surface connectivity is preserved everywhere except along the edges introduced by the cutting plane (red). The edges are shown in cyan.*

Finally, we show a generalisation to constructive solid geometry (CSG) in Fig. 7.12, where we correctly handle a boolean operation between the Stanford bunny and a cube. Our algorithm produces the correct results for the classic set of operations (union, intersection and subtraction), and does not require additional information aside from the halfedge connectivity of the input surfaces. Further results are also shown in Fig. 7.13, Fig. 7.14, Fig. 7.15, Fig. 7.16, and Fig. 7.17.

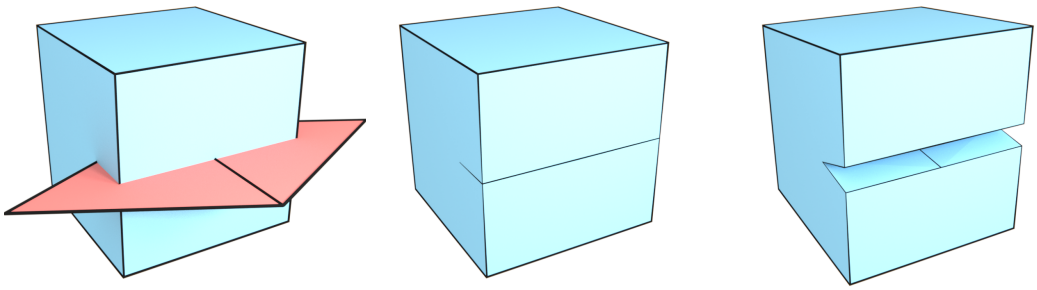


Figure 7.10: *Incision (partial) cuts.*

7.9 Conclusion

Partitioning meshes using arbitrary cut surface geometry has many useful applications in fracture simulations and, more broadly, in computer graphics. We presented an approach appropriate for cutting a surface mesh, and present a number of operations that can be performed to avoid numerical operations during polygon boundary tracking.

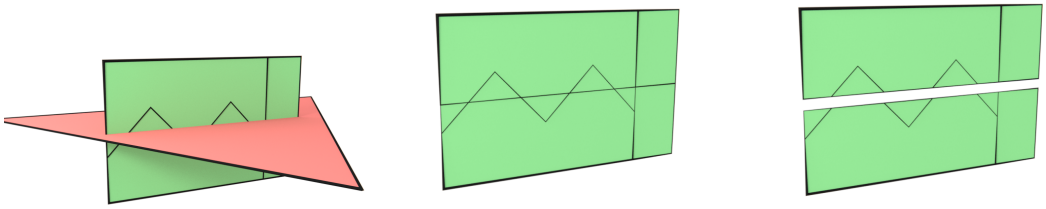
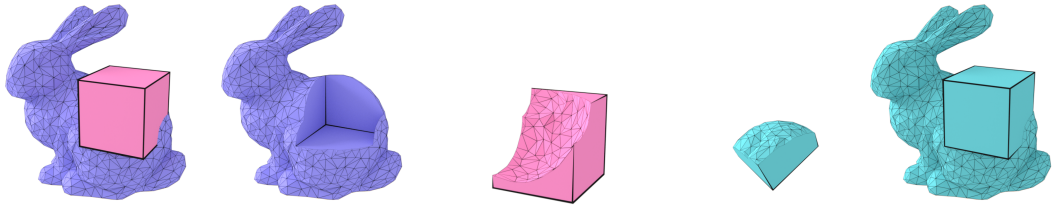


Figure 7.11: *2D (surface) cutting*



(a) Input (b) Subtract cube (c) Subtract bunny (d) Intersection (e) Union

Figure 7.12: *A boolean operation result using our mesh cutter*

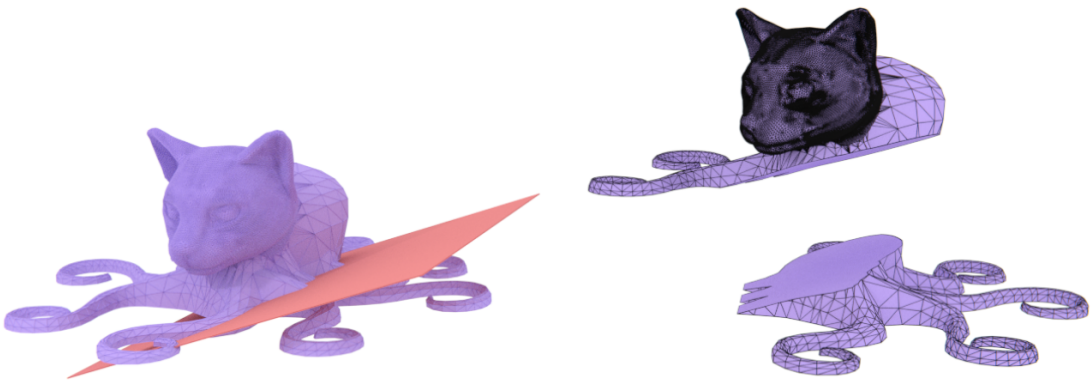


Figure 7.13: *Octocat cut with a quad plane*

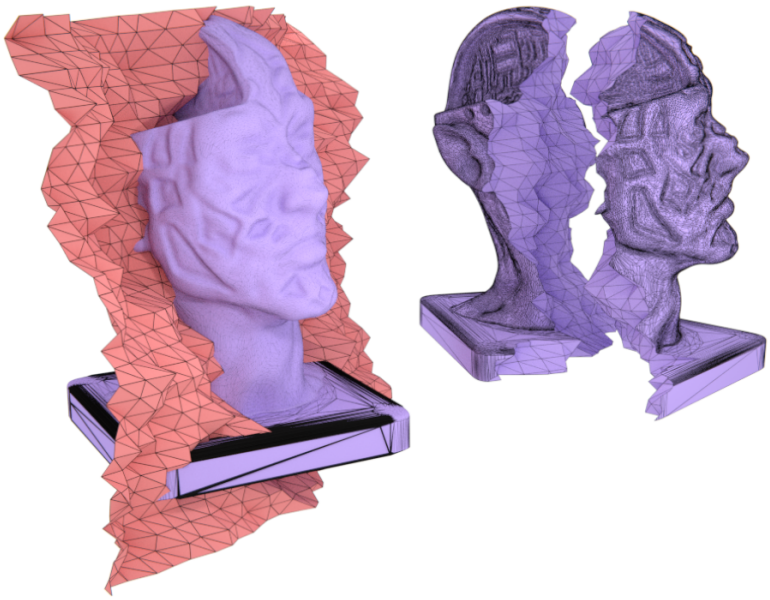


Figure 7.14: *Zombie head cut with a noisy surface.*

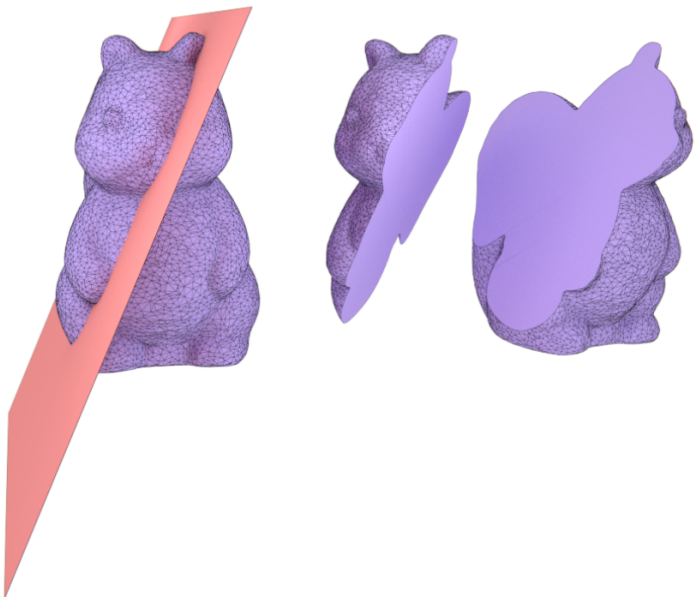


Figure 7.15: *Cute squirrel cut with quad plane.*

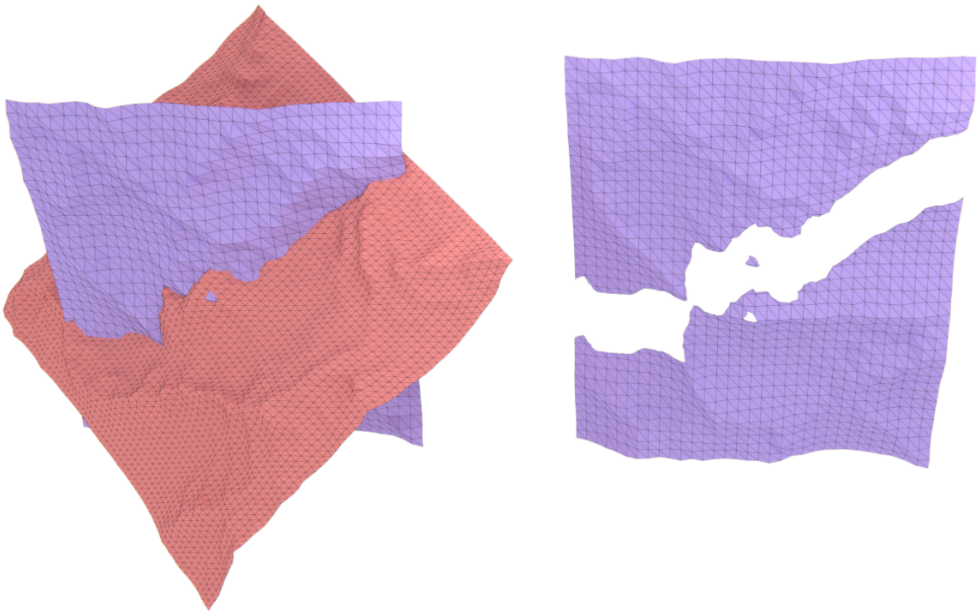


Figure 7.16: *Surface to surface cutting.*

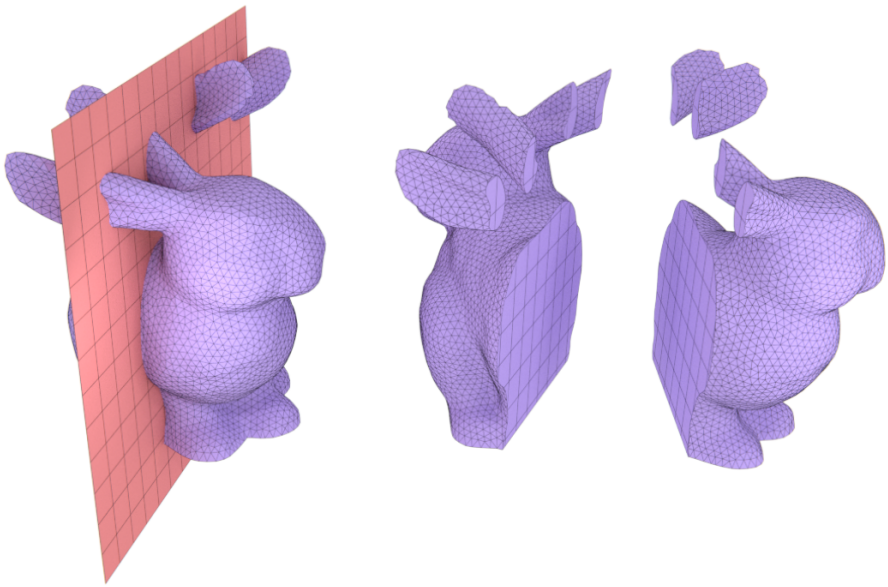


Figure 7.17: *Conjoined bunnies cut into multiple connected components.*

The strength of our approach is in the explicit and general formulation of the topological domain of the cutting problem. Floating point operations do not have to carried out throughout the cutting pipeline - but can be reduced only to the task of computing intersection points. This makes our algorithm powerful and more robust than other approaches, which e.g. first decompose the volume defined by a surface mesh and then manipulate it to approximate cut surface geometry.

Limitations: Our cutting algorithm successively consigns the use of floating-point operations to the task of calculating intersection points but it is not provably robust against rare degeneracies. Mesh intersections therefore need to be sufficiently well positioned to prevent unpredictable failures during intersection tests. Despite this fact, the likelihood of failure due to numerical error is relatively low: the algorithm is reasonably stable, even with our complex fragment geometries (see Chapter 6). One solution to providing robustness guarantees is incorporating contingency measures to detect and resolve degeneracies by using tolerances in a hierarchical manner and specifying bounds on floating point error [156]. Alternatively, geometric predicates may also be applied e.g. using the binary space-partition view for representing intersection points [12], but these require further consideration when handling concave polygon intersections.

7.10 Postscript

By providing a general, exact, and suitably robust approach to cutting arbitrary surface meshes, we have allowed almost any existing framework which uses explicit mesh representations to be seamlessly integrated with our algorithm with very little requirement on the meshes.

This presents a key step towards general surface mesh cutting algorithms, allowing for both the use of current tools, and the development of new applications with the hope that they require mesh cutting. This work is the first work specifically targetting this issue without volumetric decomposition or voxel grids, and we hope it will lead to future improvements and a wider adoption of surface mesh based cutting techniques in practical research and applications.

Part III

Conclusion

Chapter 8

Conclusion

We summarise and conclude the thesis in this section, discussing the two major objectives in its development.

The first objective was concerned with the problem of collision detection and to understand why existing parallel BVH traversal methods on GPUs had not considered implicit BVH representations, and to develop new methods to address this issue. For this objective, we developed a number of methods which can avert parallel algorithm design constraints in previous solutions - allowing faster algorithms with real-time performance. Our proposed traversal technique provides an improved solution but it is also seen as a step toward more incremental improvements in parallel BVH-based collision detection.

We also showed how the binary implicit tree could be used to advance the state of the art in parallel BVH construction. Influenced by previous methods, we developed a new tree structure called the binary ostensibly-implicit tree which has several features particularly suitable for collision detection - low memory footprint, it can be constructed very fast, and it behaves exactly like an implicit tree.

Together we believe that these techniques have opened an avenue for further use of implicit trees in the field of collision detection. By devising a new encoding of the implicit tree using only the number of objects, we have contributed to alleviating dependence on heuristic optimisations (e.g. collision-front tracking) while enabling fast constructions required in the development of applications like physics simulators and even games.

The second objective shifted from the subdomain of detecting collisions and toward simulating the behaviour of fracturing objects *after* collision. We used a method for simulating continuum solids with material discontinuities to simulate brittle frac-

ture. In this direction we first developed a simplified formulation to estimate stress magnitudes at the crack tip - estimations which characterise the singular stress field to propagate a crack. This itself is a powerful technique, providing opportunity for simplifying and speeding up crack propagation problems in volumetric domains. Additionally, this presented a foundation upon which further work could be developed and new stress based crack generation schemes explored.

Alongside the fracture simulation method we also developed an algorithm for mesh cutting and fragment generation. This algorithm can be used for a wide variety of common mesh partitioning problems, does not need any intermediate mesh representations, and fits well into the fracture simulation pipeline. This represents a significant practical contribution to the design of algorithms for mesh manipulation techniques in fracture simulation.

On a final note, in the last decade, there has been a lot of research on efficient BVH traversal on GPUs in the context of ray tracing, but collision detection has received significantly less attention. In general, the methods for broad-phase collision detection are lagging behind, which offers a positive outlook about research attempting to advance the field. However, it remains as important as ever that alternative data structures other than the BVH are researched as well to improve the field of collision detection. It is not certain that BVTT traversal is (always) the right approach in the first place. Thus, many difficult problems remain open and as important as ever.

In XFEM, we see a lot of theoretical benefits discussed in literature, including the acclaimed advantages of enrichment. However, one quickly finds that the method can be quite complex to implement with numerous intricate details to cater for - e.g. handling variable DOFs; accounting for large volume ratios in partitioned elements; numerical instability due to sub-optimal sub-element generation etc. Also, while estimating the stress field within the volume, we perform cutting using surface meshes. The fragments therefore need to be tetrahedralised for recursive breakage to be possible. This introduces an extra computational cost which can be hinder productivity as simulations take a long time to complete. Since our method requires tetrahedralisation for each crack surface that generated fragments, using specialised sign enrichments for multiple branched cracks (e.g. [27, 99]) is suggested for future work.

Bibliography

[1] Timo Aila and Tero Karras, *Architecture considerations for tracing incoherent rays*, Proc. of the conf. on high performance graphics, 2010, pp. 113–122.

[2] Timo Aila, Tero Karras, and Samuli Laine, *On quality metrics of bounding volume hierarchies*, Proceedings of the 5th high-performance graphics conference, 2013, pp. 101–107.

[3] Timo Aila and Samuli Laine, *Understanding the efficiency of ray traversal on gpus*, Proceedings of the conference on high performance graphics 2009, 2009, pp. 145–149.

[4] Ted Anderson, *Fracture mechanics: Fundamentals and applications, third edition*, CRC Press, 2005.

[5] Kimiya Aoki, Ngo Hai Dong, Toyohisa Kaneko, and Shigeru Kuriyama, *Physically based simulation of cracks on drying 3d solids*, Proceedings of the computer graphics international, 2004, pp. 357–364.

[6] Ciprian Apetrei, *Fast and Simple Agglomerative LBVH Construction*, Computer graphics and visual computing (cgvc), 2014.

[7] Zhaosheng Bao, Jeong-Mo Hong, Joseph Teran, and Ronald Fedkiw, *Fracturing rigid materials*, IEEE Transactions on Visualization and Computer Graphics **13** (March 2007), no. 2, 370–378.

[8] Rasmus Barringer and Tomas Akenine-Möller, *Dynamic stackless binary tree traversal* **2** (2013), no. 1, 38–49.

[9] Pablo Bauszat, Martin Eisemann, and Marcus Magnor, *The Minimal Bounding Volume Hierarchy*, Vision, modeling, and visualization (2010), 2010.

[10] Ted Belytschko and Thomas Black, *Elastic crack growth in finite elements with minimal remeshing*, International Journal for Numerical Methods in Engineering **45** (1999), no. 5, 601–620.

[11] Gino van den Bergen, *Efficient collision detection of complex deformable models using aabb trees*, Journal of Graphics Tools **2** (1997), no. 4, 1–13.

[12] Gilbert Bernstein and Don Fussell, *Fast, exact, linear booleans*, Proceedings of the symposium on geometry processing, 2009, pp. 1269–1278.

[13] Robert Bridson, Ronald Fedkiw, and John Anderson, *Robust Treatment of Collisions, Contact and Friction for Cloth Animation*, Proc. of the 29th Annual Conference on Computer Graphics and Interactive Techniques, 2002, pp. 594–603.

[14] Tyson Brochu, Essex Edwards, and Robert Bridson, *Efficient geometrically exact continuous collision detection*, ACM Trans. Graph. **31** (July 2012), no. 4, 96:1–96:7.

[15] Fernando Cacciola, *Triangulated surface mesh simplification*, CGAL user and reference manual, 2019.

[16] Daniel Cederman and Philippas Tsigas, *On dynamic load balancing on graphics processors*, Proc. of the 23rd acm siggraph/eurographics symposium on graphics hardware, 2008, pp. 57–64.

[17] S.K. Chan, Stephen Tuba, and William Wilson, *On the finite element method in linear fracture mechanics*, Engineering Fracture Mechanics **2** (1970), no. 1, 1 –17.

[18] Zhili Chen, Miaojun Yao, Renguo Feng, and Huamin Wang, *Physics-inspired adaptive fracture refinement*, ACM Trans. Graph. **33** (July 2014), no. 4, 113:1–113:7.

[19] Floyd M. Chitalu, Christophe Dubach, and Taku Komura, *Bulk-synchronous parallel simultaneous bvh traversal for collision detection on gpus*, Proc of i3d, 2018, pp. 4:1–4:9.

[20] David Cline, Kevin Steele, and Parris Egbert, *Lightweight bounding volumes for ray tracing*, Journal of Graphics Tools **11** (2006), no. 4, 61–71.

[21] Blender Online Community, *Blender - a 3d modelling and rendering package*, Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[22] Shane Cook, *Cuda programming: A developer's guide to parallel computing with gpus*, 1st ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms, third edition*, 3rd ed., The MIT Press, 2009.

[24] Erwin Coumans, *Bullet physics simulation*, Acm siggraph 2015 courses, 2015.

[25] Sean Curtis, Naga Govindaraju, Ilknur Kabul, Stephane Redon, and Simon Pabst, *Unc dynamic scene benchmarks*, 2009.

[26] Fang Da, Christopher Batty, and Eitan Grinspun, *Multimaterial mesh-based surface tracking*, ACM Trans. Graph. **33** (July 2014), no. 4, 112:1–112:11.

[27] Christophe Daux, Nicolas Moës, John Dolbow, Natarajan Sukumar, and Ted Belytschko, *Arbitrary branched and intersecting cracks with the extended finite element method*, International Journal for Numerical Methods in Engineering **48** (2000), no. 12, 1741–1760.

[28] Mathieu Desbrun, Mark Meyer, and Pierre Alliez, *Intrinsic parameterizations of surface meshes*, Computer Graphics Forum **21** (2002), no. 3, 209–218.

[29] C. Dick, J. Georgii, and R. Westermann, *A Hexahedral Multigrid Approach for Simulating Cuts in Deformable Objects*, IEEE Transactions on Visualization and Computer Graphics **17** (November 2011), no. 11, 1663–1675.

[30] John Everett Dolbow, *An extended finite element method with discontinuous enrichment for applied mechanics*, Ph.D. Thesis, 1999.

[31] Peng Du, Elvis S. Liu, and Toyotaro Suzumura, *Parallel continuous collision detection for high-performance gpu cluster*, Proceedings of the 21st acm siggraph symposium on interactive 3d graphics and games, 2017, pp. 4:1–4:7.

[32] Peng Du, Jie-Yi Zhao, Wan-Bin Pan, and Yi-Gang Wang, *Gpu accelerated real-time collision handling in virtual disassembly*, Journal of Computer Science and Technology **30** (2015), no. 3, 511–518.

[33] M. Eisemann, P. Bauszat, S. Guthe, and M. Magnor, *Geometry presorting for implicit object space partitioning*, Comput. Graph. Forum **31** (June 2012), no. 4, 1445–1454.

[34] Marc Elmouttie, Grégoire Krähenbühl, and George Poropat, *Robust algorithms for polyhedral modelling of fractured rock mass structure*, Computers and Geotechnics **53** (September 2013), 83–94.

[35] Christer Ericson, *Real-time collision detection*, CRC Press, Inc., 2004.

[36] Naznin Fauzia, Louis-Noël Pouchet, and P. Sadayappan, *Characterizing and enhancing global memory data coalescing on gpus*, Proc. of the 13th annual ieee/acm international symposium on code generation and optimization, 2015, pp. 12–22.

[37] Thomas-Peter Fries and Malak Baydoun, *Crack propagation with the extended finite element method and a hybrid explicit-implicit crack description*, International Journal for Numerical Methods in Engineering **89** (2011), no. 12, 1527–1558.

[38] Thomas-Peter Fries and Ted Belytschko, *The intrinsic xfem: a method for arbitrary discontinuities without additional unknowns*, International Journal for Numerical Methods in Engineering **68** (2006), no. 13, 1358–1385.

[39] Thomas-Peter Fries and Ted Belytschko, *The extended/generalized finite element method: An overview of the method and its applications*, International Journal for Numerical Methods in Engineering **84** (2010), no. 3, 253–304.

[40] K. Garanzha, *Efficient clustered bvh update algorithm for highly-dynamic models*, 2008 ieee symp on interactive ray tracing, 2008Aug, pp. 123–130.

[41] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister, *Simpler and faster hlbvh with work queues*, Eurographics/ acm siggraph symposium on high performance graphics, 2011.

[42] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister, *Simpler and faster hlbvh with work queues*, Proc of hpg, 2011, pp. 59–64.

[43] J. Garzon, P. O'Hara, C. A. Duarte, and W. G. Buttlar, *Improvements of explicit crack surface representation and update within the generalized finite element method with application to three-dimensional crack coalescence*, International Journal for Numerical Methods in Engineering **97** (2013), no. 4, 231–273.

[44] L. Glondu, M. Marchal, and G. Dumont, *Real-time simulation of brittle fracture using modal analysis*, IEEE Transactions on Visualization and Computer Graphics **19** (2013Feb), no. 2, 201–209.

[45] J. Goldsmith and J. Salmon, *Automatic creation of object hierarchies for ray tracing*, IEEE Computer Graphics and Applications **7** (1987May), no. 5, 14–20.

[46] Stefan Gottschalk, Ming C Lin, and Dinesh Manocha, *Obbtree: A hierarchical structure for rapid interference detection*, Proc. of the 23rd annual conf. on computer graphics and interactive techniques, 1996, pp. 171–180.

[47] Stefan Aric Gottschalk, *Collision queries using oriented bounding boxes*, Ph.D. Thesis, 2000.

[48] David Hahn, *Brittle fracture simulation with boundary elements for computer graphics*, Ph.D. Thesis, 2017.

[49] David Hahn and Chris Wojtan, *High-resolution brittle fracture simulation with boundary elements*, ACM Trans. Graph. **34** (July 2015), no. 4, 151:1–151:12.

[50] David Hahn and Chris Wojtan, *Fast approximations for boundary element based brittle fracture simulation*, ACM Trans. Graph. **35** (July 2016), no. 4, 104:1–104:11.

[51] Eric Haines, *Graphics gems iv*, 1994, pp. 24–46.

[52] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek, *Efficient stack-less bvh traversal for ray tracing*, Proceedings of the 27th spring conference on computer graphics, 2011, pp. 7–12.

[53] Pawan Harish and P. J. Narayanan, *Accelerating large graph algorithms on the gpu using cuda*, Proc. of the 14th international conf. on high performance computing, 2007, pp. 197–208.

[54] Jan Hegemann, Chenfanfu Jiang, Craig Schroeder, and Joseph M. Teran, *A level set method for ductile fracture*, Proceedings of the 12th acm siggraph/eurographics symposium on computer animation, 2013, pp. 193–201.

[55] John L. Hennessy and David A. Patterson, *Computer architecture, sixth edition: A quantitative approach*, 6th ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017.

[56] A. Hermann, S. Klemm, Z. Xue, A. Roennau, and R. Dillmann, *Gpu-based real-time collision detection for motion execution in mobile manipulation planning*, 2013 16th international conference on advanced robotics (icar), 2013Nov, pp. 1–7.

[57] Qiming Hou, Kun Zhou, and Baining Guo, *Bsgp: Bulk-synchronous gpu programming*, Acm siggraph 2008 papers, 2008, pp. 19:1–19:12.

[58] Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo, *Tetrahedral meshing in the wild*, ACM Trans. Graph. **37** (July 2018), no. 4, 60:1–60:14.

[59] Y. Ikegawa and J.A. Hudson, *A novel automatic identification system for three-dimensional multi-block systems*, Engineering Computations **9** (1992), no. 2, 169–179.

[60] Y. Ikegawa and J.A. Hudson, *A Novel Automatic Identification System for Three-Dimensional Multi-Block Systems*, Engineering Computations **9** (February 1992), no. 2, 169–179.

[61] A.R. Ingraffea and P.A. Wawrzynek, *Finite Element Methods for Linear Elastic Fracture Mechanics*, Comprehensive Structural Integrity, 2003, pp. 1–88.

[62] George R Irwin, *Analysis of stresses and strains near the end of a crack traversing a plate*, J. appl. Mech. (1957).

[63] Damkjær Jesper, *A comparison of acceleration structures for gpu assisted ray tracing*, Department of Computer Science, University of Copenhagen, 2005.

[64] L. Jeřábková and T. Kuhlen, *Stable cutting of deformable objects in virtual environments using xfem*, IEEE Computer Graphics and Applications **29** (2009March), no. 2, 61–71.

[65] L. Jing and O. Stephansson, *Topological identification of block assemblages for jointed rock masses*, International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts **31** (1994), no. 2, 163 –172.

[66] Lanru Jing, *Block system construction for three-dimensional discrete element models of fractured rocks*, International Journal of Rock Mechanics and Mining Sciences **37** (2000), no. 4, 645 –659.

[67] Lanru Jing and Ove Stephansson, *6 - The Basics of Combinatorial Topology for Block System Representation*, Developments in Geotechnical Engineering, January 2007, pp. 179–197.

[68] Wu Jun, Westermann Rüdiger, and Dick Christian, *Real-Time Haptic Cutting of High-Resolution Soft Tissues*, Studies in Health Technology and Informatics (2014), 469–475.

[69] Tero Karras, *Maximizing parallelism in the construction of bvhs, octrees, and k-d trees*, Proc of hpg, 2012, pp. 33–37.

[70] Peter Kaufmann, Sebastian Martin, Mario Botsch, Eitan Grinspun, and Markus Gross, *Enrichment textures for detailed cutting of shells*, ACM Trans. Graph. **28** (July 2009), no. 3, 50:1–50:10.

[71] Peter Kaufmann, Sebastian Martin, Mario Botsch, and Markus Gross, *Flexible simulation of deformable models using discontinuous galerkin fem*, Proceedings of the 2008 acm siggraph/eurographics symposium on computer animation, 2008, pp. 105–115.

[72] A. Kensler, *Tree rotations for improving bounding volume hierarchies*, 2008 ieee symp on interactive ray tracing, 2008Aug, pp. 73–76.

[73] Amir R. Khoei, *Extended finite element method: theory and applications*, Wiley series in computational mechanics, John Wiley & Sons, Ltd, 2015.

[74] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan, *Cusha: Vertex-centric graph processing on gpus*, Proc. of the 23rd international symposium on high-performance parallel and distributed computing, 2014, pp. 239–252.

[75] James T Klosowski, Martin Held, Joseph SB Mitchell, Henry Sowizral, and Karel Zikan, *Efficient collision detection using bounding volume hierarchies of k-dops*, IEEE transactions on Visualization and Computer Graphics **4** (1998), no. 1, 21–36.

[76] Dave Knott and Dinesh K. Pai, *CInDeR: Collision and interference detection in real-time using graphics hardware*, Proc. of the graphics interface 2003 conference, june 11-13, 2003, halifax, nova scotia, canada, 2003June, pp. 73–80.

[77] Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler, *Fast, effective bvh updates for animated scenes*, Proc of i3d, 2012, pp. 197–204.

[78] Dan Koschier, Jan Bender, and Nils Thuerey, *Robust extended finite elements for complex cutting of deformables*, ACM Trans. Graph. **36** (July 2017), no. 4, 55:1–55:13.

[79] Dan Koschier, Sebastian Lipponer, and Jan Bender, *Adaptive tetrahedral meshes for brittle fracture simulation*, Proceedings of the acm siggraph/eurographics symposium on computer animation, 2014, pp. 57–66.

[80] Meinhard Kuna, *Finite Elements in Fracture Mechanics*, Solid Mechanics and Its Applications, vol. 201, Springer Netherlands, Dordrecht, 2013 (en).

[81] Samuli Laine, *Restart trail for stackless BVH traversal*, Proc. of the conf. on high performance graphics, 2010, pp. 107–111.

[82] Thomas Larsson and Tomas Akenine-Möller, *A dynamic bounding volume hierarchy for generalized collision detection*, Comput. Graph. **30** (June 2006), no. 3, 450–459.

[83] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, *Fast bvh construction on gpus*, Computer Graphics Forum **28** (2009), no. 2, 375–384.

[84] C. Lauterbach, Q. Mo, and D. Manocha, *gproximity: Hierarchical gpu-based operations for collision and distance queries*, Computer Graphics Forum **29** (2010), no. 2, 419–428.

[85] Christian Lauterbach, Qi Mo, and Dinesh Manocha, *Work distribution methods on gpus*, 2009.

[86] Andrew Lenharth, Donald Nguyen, and Keshav Pingali, *Parallel graph analytics*, Commun. ACM **59** (April 2016), no. 5, 78–87.

[87] Tsai-Yen Li and Jin-Shin Chen, *Incremental 3d collision detection with hierarchical data structures*, Proc. of the acm symposium on virtual reality software and technology, 1998, pp. 139–144.

[88] D. Lin, C. Fairhurst, and A.M. Starfield, *Geometrical identification of three-dimensional rock block systems using topological techniques*, International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts **24** (1987), no. 6, 331 –338.

[89] Peter Lindstrom and Greg Turk, *Fast and memory efficient polygonal simplification*, Proceedings of the conference on visualization '98, 1998, pp. 279–286.

[90] Hang Liu, H. Howie Huang, and Yang Hu, *ibfs: Concurrent breadth-first search on gpus*, Proc. of the 2016 international conf. on management of data, 2016, pp. 403–416.

[91] Khaled Mamou and Faouzi Ghorbel, *A simple and efficient approach for 3d mesh approximate convex decomposition*, 200911, pp. 3501–3504.

[92] Morgan McGuire, *Computer graphics archive*, 2017.

[93] Donald Meagher, *Geometric modeling using octree encoding*, Computer Graphics and Image Processing **19** (1982), no. 2, 129 –147.

[94] Gang Mei and John C. Tipper, *Simple and Robust Boolean Operations for Triangulated Surfaces*, arXiv:1308.4434 [cs] (August 2013). arXiv: 1308.4434.

[95] Duane Merrill, Michael Garland, and Andrew Grimshaw, *Scalable GPU graph traversal*, Proc. of the 17th ACM SIGPLAN symposium on principles and practice of parallel programming, 2012, pp. 117–128.

[96] Nicolas Moës, John Dolbow, and Ted Belytschko, *A finite element method for crack growth without remeshing*, International Journal for Numerical Methods in Engineering **46** (1999), no. 1, 131–150.

[97] Neil Molino, Zhaosheng Bao, and Ron Fedkiw, *A virtual node algorithm for changing mesh topology during simulation*, Acm siggraph 2005 courses, 2005.

[98] George Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, 1966.

[99] S. E. Mousavi, E. Grinspun, and N. Sukumar, *Harmonic enrichment functions: A unified treatment of multiple, intersecting and branched cracks in the extended finite element method*, International Journal for Numerical Methods in Engineering **85** (2011), no. 10, 1306–1322.

[100] S. E. Mousavi and N. Sukumar, *Generalized Gaussian quadrature rules for discontinuities and crack singularities in the extended finite element method*, Computer Methods in Applied Mechanics and Engineering **199** (December 2010), no. 49, 3237–3249.

[101] Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim, *Real time dynamic fracture with volumetric approximate convex decompositions*, ACM Trans. Graph. **32** (July 2013), no. 4, 115:1–115:10.

[102] Matthias Müller and Markus Gross, *Interactive virtual materials*, Proceedings of graphics interface 2004, 2004, pp. 239–246.

[103] Matthias Müller, Leonard McMillan, Julie Dorsey, and Robert Jagnow, *Real-time simulation of deformation and fracture of stiff materials*, Proceedings of the eurographic workshop on computer animation and simulation, 2001, pp. 113–124.

[104] Ken Museth, *Vdb: High-resolution sparse volumes with dynamic topology*, ACM Trans. Graph. **32** (July 2013), no. 3, 27:1–27:22.

[105] Alan Norton, Greg Turk, Bob Bacon, John Gerth, and Paula Sweeney, *Animation of fracture by physical modeling*, The Visual Computer **7** (1991Jul), no. 4, 210–219.

[106] James F. O'Brien, *Graphical modeling and animation of ductile fracture*, Proceedings of the 29th international conference on computer graphics and interactive techniques. electronic art and animation catalog., 2002, pp. 161–161.

[107] James F. O'Brien and Jessica K. Hodgins, *Graphical modeling and animation of brittle fracture*, Proceedings of the 26th annual conference on computer graphics and interactive techniques, 1999, pp. 137–146.

[108] Stanley Osher and James A Sethian, *Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-jacobi formulations*, Journal of Computational Physics **79** (1988), no. 1, 12 –49.

[109] Jia Pan, Christian Lauterbach, and Dinesh Manocha, *g-planner: Real-time motion planning and global navigation using gpus*, Proc. of the twenty-fourth aaai conf. on artificial intelligence, 2010, pp. 1245–1251.

[110] Jia Pan and Dinesh Manocha, *GPU-Based Parallel Collision Detection for Real-Time Motion Planning*, Algorithmic Foundations of Robotics IX, 2010, pp. 211–228 (en).

[111] Jia Pan and Dinesh Manocha, *Gpu-based parallel collision detection for real-time motion planning*, Algorithmic foundations of robotics ix: Selected contributions of the ninth international workshop on the algorithmic foundations of robotics, 2011, pp. 211–228.

[112] J. Pantaleoni and D. Luebke, *Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry*, Proc of hpg, 2010, pp. 87–95.

[113] Eric G. Parker and James F. O'Brien, *Real-time deformation and fracture in a game environment*, Proceedings of the 2009 acm siggraph/eurographics symposium on computer animation, 2009, pp. 165–175.

[114] Miguel Patricio and Robert M M Mattheij, *Crack Propagation Analysis (casa-report: Vol. 0723).* (2007), 25 (en).

[115] Christoph J. Paulus, Lionel Untereiner, Hadrien Courtecuisse, Stéphane Cotin, and David Cazier, *Virtual cutting of deformable objects based on efficient topological operations*, Vis. Comput. **31** (June 2015), no. 6-8, 831–841.

[116] Mark Pauly, Richard Keiser, Bart Adams, Philip Dutré, Markus Gross, and Leonidas J. Guibas, *Meshless animation of fracturing solids*, Acm transactions on graphics (proceedings of acm siggraph 2005), 2005August, pp. 957–964.

[117] J. P. Pereira, C. A. Duarte, D. Guoy, and X. Jiao, *hp-generalized fem and crack surface representation for non-planar 3-d cracks*, International Journal for Numerical Methods in Engineering **77** (2009), no. 5, 601–633.

[118] Tobias Pfaff, Rahul Narain, Juan Miguel de Joya, and James F. O'Brien, *Adaptive tearing and cracking of thin sheets*, ACM Trans. Graph. **33** (July 2014), no. 4, 110:1–110:9.

[119] Xiang Ren and Xuefei Guan, *Three dimensional crack propagation through mesh-based explicit representation for arbitrarily shaped cracks using the extended finite element method*, Engineering Fracture Mechanics **177** (2017), 218 –238.

[120] Casey L Richardson, Jan Hegemann, Eftychios Sifakis, Jeffrey Hellrung, and Joseph M Teran, *An xfem method for modeling geometrically elaborate crack propagation in brittle materials*, Institute for Computational and Applied Mathematics **90095** (2009), 1555.

[121] Matthew Scarpino, *OpenCL in Action: How to Accelerate Graphics and Computations*, 2011.

[122] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, *Scan primitives for gpu computing*, Proceedings of the 22nd acm siggraph/eurographics symposium on graphics hardware, 2007, pp. 97–106.

[123] Johnathan Richard Shewchuk, *Robust adaptive floating-point geometric predicates*, Proceedings of the twelfth annual symposium on computational geometry, 1996, pp. 141–150.

[124] SideFX, *Houdini - 3d modeling, animation, vfx, look development, lighting and rendering*, 2019.

[125] Daniel Sieger and Mario Botsch, *Design, implementation, and evaluation of the surface_mesh data structure*, Proceedings of the 20th international meshing roundtable, 2012.

[126] Eftychios Sifakis, *Algorithmic aspects of the simulation and control of computer generated human anatomy models* (20076), 203.

[127] Eftychios Sifakis, Kevin G. Der, and Ronald Fedkiw, *Arbitrary cutting of deformable tetrahedralized objects*, Proceedings of the 2007 acm siggraph/eurographics symposium on computer animation, 2007, pp. 73–80.

[128] Jeffrey Smith, Andrew Witkin, and David Baraff, *Fast and controllable simulation of the shattering of brittle objects*, Computer Graphics Forum **20** (2001), no. 2, 81–91.

[129] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg, *Whippletree: Task-based scheduling of dynamic workloads on the gpu*, ACM Trans. Graph. **33** (November 2014), no. 6, 228:1–228:11.

[130] M. Stolarska, D. L. Chopp, N. Moës, and T. Belytschko, *Modelling crack growth by level sets in the extended finite element method*, International Journal for Numerical Methods in Engineering **51** (2001), no. 8, 943–960.

[131] M. Stolarska and D.L. Chopp, *Modeling thermal fatigue cracking in integrated circuits by level sets and the extended finite element method*, International Journal of Engineering Science **41** (2003), no. 20, 2381 –2410.

[132] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle, *A material point method for snow simulation*, ACM Trans. Graph. **32** (July 2013), no. 4, 102:1–102:10.

[133] Jonathan Su, Craig Schroeder, and Ronald Fedkiw, *Energy stability and fracture for frame rate rigid body simulations*, Proceedings of the 2009 acm siggraph/eurographics symposium on computer animation, 2009, pp. 155–164.

[134] N. Sukumar, D.L. Chopp, and B. Moran, *Extended finite element method and fast marching method for three-dimensional fatigue crack propagation*, Engineering Fracture Mechanics **70** (2003), no. 1, 29 –48.

[135] N. Sukumar, N. Moës, B. Moran, and T. Belytschko, *Extended finite element method for three-dimensional crack modelling*, International Journal for Numerical Methods in Engineering **48** (2000), no. 11, 1549–1570.

[136] Ivan E. Sutherland and Gary W. Hodgman, *Reentrant polygon clipping*, Commun. ACM **17** (January 1974), no. 1, 32–42.

[137] Min Tang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha, *PSCC: Parallel self-collision culling with spatial hashing on GPUs*, Proceedings of the ACM on Computer Graphics and Interactive Techniques **1** (2018), no. 1, 18:1–18.

[138] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong, *Collision-streams: Fast gpu-based collision detection for deformable models*, Symposium on interactive 3d graphics and games, 2011, pp. 63–70.

[139] Min Tang, Dinesh Manocha, and Ruofeng Tong, *Mccd: Multi-core collision detection between deformable models using front-based decomposition*, Graphical Models **72** (2010), no. 2, 7–23.

[140] Min Tang, Ruofeng Tong, Rahul Narain, Chang Meng, and Dinesh Manocha, *A gpu-based streaming algorithm for high-resolution cloth simulation*, Computer Graphics Forum **32** (2013), no. 7, 21–30.

[141] Min Tang, Huamin Wang, Le Tang, Ruofeng Tong, and Dinesh Manocha, *CAMA: Contact-aware matrix assembly with unified collision handling for GPU-based cloth simulation*, Computer Graphics Forum (Proceedings of Eurographics 2016) **35** (2016), no. 2, 511–521.

[142] Min Tang, Tongtong Wang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha, *I-cloth: Incremental collision handling for gpu-based interactive cloth simulation*, ACM Trans. Graph. **37** (2018), no. 6, 204:1–204:10.

[143] Michael Tao, Christopher Batty, Eugene Fiume, and David Levin, *Mandoline: Robust cut-cell generation for arbitrary triangle meshes*, ACM Transactions on Graphics (2019).

[144] J. Teran, S. Blemker, V. Ng Thow Hing, and R. Fedkiw, *Finite Volume Methods for the Simulation of Skeletal Muscle*, Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2003, pp. 68–74.

[145] Joseph Teran, Eftychios Sifakis, Silvia S. Blemker, Victor Ng-Thow-Hing, Cynthia Lau, and Ronald Fedkiw, *Creating and simulating skeletal muscle from the visible human data set*, IEEE Transactions on Visualization and Computer Graphics **11** (May 2005), no. 3, 317–328.

[146] Demetri Terzopoulos and Kurt Fleischer, *Modeling inelastic deformation: Viscolelasticity, plasticity, fracture*, SIGGRAPH Comput. Graph. **22** (June 1988), no. 4, 269–278.

[147] 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** (2005), no. 1, 61–81.

[148] Niels Thrane, Lars Ole Simonsen, and Advisor Peter Ørbæk, *A comparison of acceleration structures for gpu assisted ray tracing*, Aarhus University, 2005.

[149] Stanley Tzeng, Anjul Patney, and John D. Owens, *Task management for irregular-parallel workloads on the gpu*, Proc. of the conf. on high performance graphics, 2010, pp. 29–37.

[150] Leslie G. Valiant, *A bridging model for parallel computation* **33** (August 1990), no. 8, 103–111.

[151] Carsten Wächter and Alexander Keller, *Instant ray tracing: The bounding interval hierarchy*, Proc of egsr, 2006, pp. 139–149.

[152] Ingo Wald, *On fast construction of sah-based bounding volume hierarchies*, Proc of the 2007 ieee symp on interactive ray tracing, 2007, pp. 33–40.

[153] Tongtong Wang, Zhihua Liu, Min Tang, Ruofeng Tong, and Dinesh Manocha, *Efficient and reliable self-collision culling using unprojected normal cones*, Computer Graphics Forum **36** (2017), no. 8, 487–498.

[154] Xinlei Wang, Min Tang, Dinesh Manocha, and Ruofeng Tong, *Efficient BVH-based collision detection scheme with ordering and restructuring*, Computer Graphics Forum (Proc of Eurographics 2018) **37** (2018), no. 2.

[155] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens, *Gunrock: A high-performance graph processing library on the GPU*, Proc. of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming, 2016, pp. 11:1–11:12.

[156] Yuting Wang, Chenfanfu Jiang, Craig Schroeder, and Joseph Teran, *An adaptive virtual node algorithm with robust mesh cutting*, Proceedings of the acm siggraph/eurographics symposium on computer animation, 2014, pp. 77–85.

[157] Kevin Weiler and Peter Atherton, *Hidden surface removal using polygon area sorting*, SIGGRAPH Comput. Graph. **11** (July 1977), no. 2, 214–222.

[158] René Weller, Nicole Debowski, and Gabriel Zachmann, *kdet: Parallel constant time collision detection for polygonal objects*, Comput. Graph. Forum **36** (May 2017), no. 2, 131–141.

[159] René Weller, Nicole Debowski, and Gabriel Zachmann, *kdet: Parallel constant time collision detection for polygonal objects*, Computer Graphics Forum **36** (2017), no. 2, 131–141.

[160] Martin Wicke, Daniel Ritchie, Bryan M. Klingner, Sebastian Burke, Jonathan R. Shewchuk, and James F. O'Brien, *Dynamic local remeshing for elastoplastic simulation*, Acm siggraph 2010 papers, 2010, pp. 49:1–49:11.

[161] Chris Wojtan, Matthias Müller-Fischer, and Tyson Brochu, *Liquid simulation with mesh-based surface tracking*, Acm siggraph 2011 courses, 2011, pp. 8:1–8:84.

[162] Joshuah Wolper, Yu Fang, Minchen Li, Jiecong Lu, Ming Gao, and Chenfanfu Jiang, *Cd-mpm: Continuum damage material point methods for dynamic fracture animation*, ACM Trans. Graph. **38** (July 2019), no. 4, 119:1–119:15.

[163] Tsz Ho Wong, Geoff Leach, and Fabio Zambetta, *An adaptive octree grid for gpu-based collision detection of deformable objects*, Vis. Comput. **30** (June 2014), no. 6-8, 729–738.

[164] Jun Wu, Rudiger Westermann, and Christian Dick, *A survey of physically based simulation of cuts in deformable bodies*, Comput. Graph. Forum **34** (September 2015), no. 6, 161–187.

[165] Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson, *Mesh arrangements for solid geometry*, ACM Trans. Graph. **35** (July 2016), no. 4, 39:1–39:15.

[166] Yufeng Zhu, Robert Bridson, and Chen Greif, *Simulating rigid body fracture with surface meshes*, ACM Trans. Graph. **34** (July 2015), no. 4, 150:1–150:11.

[167] O.C. Zienkiewicz and R.L Taylor, *Finite element method*, 5th ed., Butterworth-Heinemann, 2002.

Appendix A

Quick Primer on GPU Architecture

This appendix briefly describes GPUs and how they are designed for massively parallel computation. It also describes the challenges of using GPUs. An illustrative overview is shown in Fig. A.1. We also recommend to readers the book by Hennessy and Patterson [55] for a more thorough treatment.

Broadly, a GPU is an array of highly-threaded multiprocessor cores (CUDA 'symmetric multiprocessor' or OpenCL 'compute units') with three distinct memory regions that are classified by access policies and associated latency. Global memory is the largest region which has high-bandwidth but low latency, and with read-write access. Local (i.e. shared) memory is the faster and much smaller scratchpad memory that is on a multiprocessor core, which is used for fast communication and data sharing. Private memory represents the registers that are accessible by a single *logical* lane of execution.

During execution, compute work is organised hierarchically to map data elements across cores and with scheduling done at the granularity of batches of logical lanes of execution (CUDA warps and AMD wave-fronts). A batch is a thread of single-instruction multiple-data (SIMD) instructions which is the machine object that the hardware creates, manages, schedules, and executes[1]. Each batch is scheduled by a SIMD thread scheduler which issues instructions when they are ready. Execution of batches occurs using *functional* SIMD lanes in the hardware (OpenCL 'processing elements'), which execute operations on a single data element at a time. The SIMD lane represents a datapath and register file portion of a multiprocessor core that executes operations for one or more logical lanes of a batch.

Batches are further organised into groups which provide a more coarse-grained

[1] It is a 'thread' in the classical sense but it contains exclusively SIMD instructions.

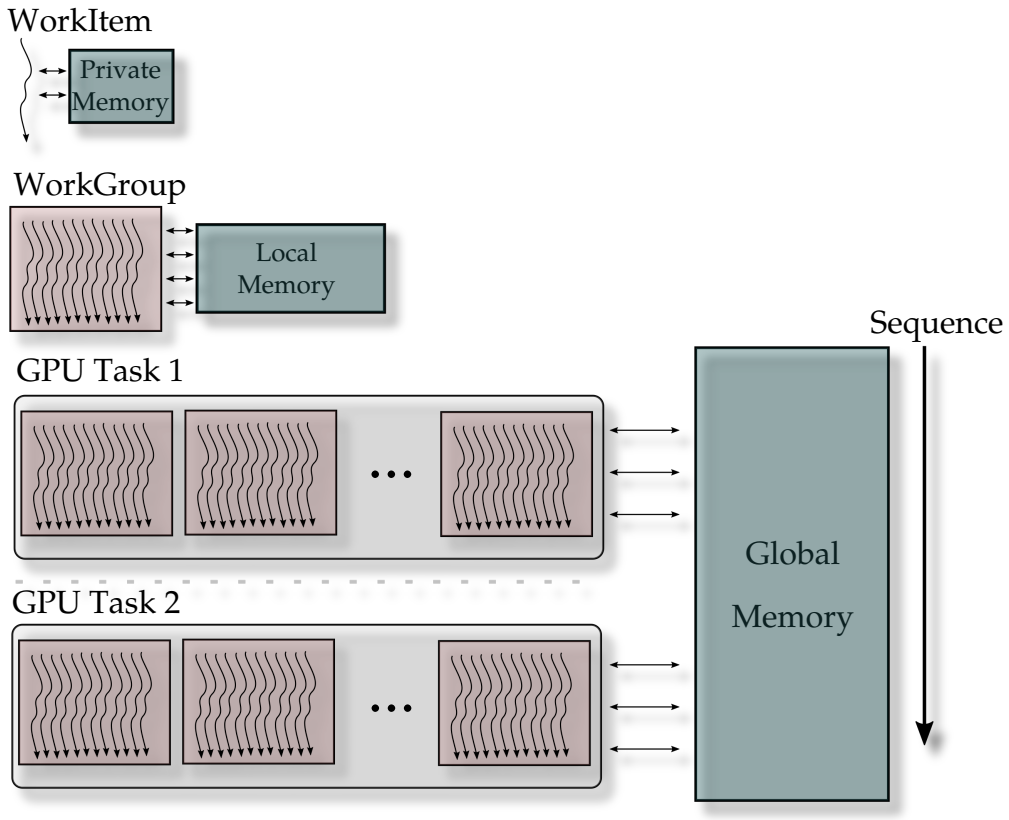


Figure A.1: *GPU memory structures (*cf. *Fig.4.18 in* [55]*). Global memory is shared by GPU tasks, local memory is shared by all threads in a group, and private memory is the non-shared portion of the register file which is accessible only by a single lane of execution (CUDA Thread or OpenCL work-item).*

parallel decomposition of workloads (CUDA ‘blocks’ and OpenCL ‘work-groups’). Synchronisation on the GPU is only permitted between execution lanes in the same group. Each group is scheduled on a multi-processor core and has access to its local memory region. Multiple groups may also be concurrently scheduled on a single multiprocessor-core depending on resource constraints (e.g. register allocation).

The specific design of GPUs makes for an extra challenge in software programs with parallelism as compared to CPUs. These programs must rely on features like memory coalescing to reduce load/store memory transactions by ensuring that data is accessed in contiguous and aligned blocks. Coalescing is used to overcome the latency of global memory accesses by strategically harnessing its high-bandwidth data transfer rates. Another hardware strategy for mitigating latency issues is the favouring massively data parallel work by sacrificing serial performance for throughput using interleaved batch scheduling - when a batch stalls another is scheduled in its place. In general, GPUs are suited for simpler computation (number crunching) by dedicating more hardware resources to arithmetic logic rather than control-logic. Thus, branching and control-flow divergence amongst logical SIMD lanes must be avoided lest dealing with hardware under-utilisation and non-coalesced global memory accesses.

Appendix B

κ-ary Ostensibly-Implicit Trees

For completeness, a κ-ary ostensibly implicit tree ($\kappa = 2, 3, 4, \ldots$) is described in this appendix, which is similar to descriptions given in Section 3.4.1 and Section 3.4.2. We start by defining a set $\mathcal{X}_\kappa(L_v) = \{\kappa^{y_1}, \kappa^{y_2} \ldots, \kappa^{y_N}\}, y_i \in \mathcal{Y}_\kappa(L_v)$, where $\mathcal{Y}_\kappa(L_v) = \{y_1, y_2, \ldots, y_N\}$, such that

$$
\begin{aligned}
y_1 &= \lfloor \log_\kappa(L_v) \rfloor \\
y_2 &= \lfloor \log_\kappa(L_v - \kappa^{y_1}) \rfloor \\
&\ldots \\
y_N &= \left\lfloor \log_\kappa \left(L_v - \sum_{i=1}^{N-1} \kappa^{y_i} \right) \right\rfloor .
\end{aligned}
$$

The total number of virtual nodes in the tree will be (from Eq. (3.5))

$$
N_v = \sum_{i=1}^{N} \left(\frac{\kappa x_i - 1}{\kappa - 1} \right), \quad x_i \in \mathcal{X}_\kappa(L_v), \tag{B.1}
$$

and memory locations are computed as in Eq. (3.9) but with

$$
L_{vl} = \left\lfloor \frac{L_v}{\kappa^{\bar{l}-l}} \right\rfloor, \tag{B.2}
$$

which is the general form of Eq. (3.8).