# Efficient Parallel Proximity Queries and an Application to Highly Complex Motion Planning Problems with Many Narrow Passages

Dissertation
zur Erlangung des Grades
"Doktor der Naturwissenschaften"

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität in Mainz,

vorgelegt von
**Rainer Erbes**
geboren in Mainz

Mainz, den 25.03.2013

Berichterstatter:

Tag der mündlichen Prüfung: 15. Mai 2013

D77

# Abstract

In industrial manufacturing, like the automotive industry, digital mock-ups are used to design complex machinery with the help of computer systems. In this field, motion planning algorithms play an important role to ensure the (de-)composability of the digital prototypes. In the last decades, sampling-based motion planning algorithms have shown themselves to be practical in this application. In order to construct a (dis-)assembly path for a virtual 3D object, these algorithms generate a high number of different placements for the object and validate these positions by utilizing a collision detection routine. Hence, collision detection is a very important sub-task in sampling-based motion planning and an efficient collision detection routine is essential to every sampling-based planner. One of the main difficulties for these planners results from "narrow passages" that appear whenever the movements of the assembled component are highly restricted. It can then be difficult to find a sufficient number of collision free samples and more sophisticated techniques might be required to enhance the performance of the algorithm.

The present work is organized in two parts: In the first part, we analyze parallel collision detection algorithms. Focusing on the application to sampling-based motion planning, we formulate the a basic collision query as testing the same two rigid body objects in a number of different relative placements. We implement and compare different approaches based on bounding volume hierarchies and hierarchical grids that are executed in parallel on multiple CPU cores. Besides, we describe the design of different parallel CUDA kernels that perform collision tests based on bounding volume hierarchies. In addition to different distributions of the work amongst the available GPU threads, we analyze the effect of different memory access patterns on the performance of our CUDA kernels. Furthermore, we propose to improve the performance of the algorithms at the expense of their accuracy and analyze a family of different approximate collision tests.

In the second part, we describe a novel parallel, sampling-based motion planner that is especially suited for complex motion planning problems with many narrow passages. The algorithm works in two phases. The basic idea is to conceptually allow small object interpenetrations and not to enforce a high accuracy in the first planning phase. We describe a non-conservative motion planner that is based on an Expansive Space Tree. The approach uses an iterative sample retraction mechanism to actively improve the performance in highly restricted environments. To further improve the performance, we optionally allow to apply approximate collision tests in the first planning phase. This results in a disassembly path that is not completely collision free. In the second phase, we repair this path with a new type of sampling-based motion planner that locally re-plans a valid path in close proximity to the non-conservative solution.
To benchmark our new algorithm and show its strength for highly complex motion planning problems with many narrow passages, we have modeled and solved a new family of very complex metal puzzles in addition to the well known alpha puzzle. To the best of our knowledge, there is no comparably complex benchmark set of rigid body motion planning problems publicly available, neither have comparable benchmarks been described in the motion planning literature.

# Zusammenfassung

In vielen Bereichen der industriellen Fertigung, wie zum Beispiel in der Automobilindustrie, werden digitale Versuchsmodelle (sog. *digital mock-ups*) eingesetzt, um die Entwicklung komplexer Maschinen möglichst gut durch Computersysteme unterstützen zu können. Hierbei spielen Bewegungsplanungsalgorithmen eine wichtige Rolle, um zu gewährleisten, dass diese digitalen Prototypen auch kollisionsfrei zusammengesetzt werden können. In den letzten Jahrzehnten haben sich hier sampling-basierte Verfahren besonders bewährt. Diese erzeugen eine große Anzahl von zufälligen Lagen für das ein-/auszubauende Objekt und verwenden einen Kollisionserkennungsmechanismus, um die einzelnen Lagen auf Gültigkeit zu überprüfen. Daher spielt die Kollisionserkennung eine wesentliche Rolle beim Design effizienter Bewegungsplanungsalgorithmen. Eine Schwierigkeit für diese Klasse von Planern stellen sogenannte "narrow passages" dar, schmale Passagen also, die immer dort auftreten, wo die Bewegungsfreiheit der zu planenden Objekte stark eingeschränkt ist. An solchen Stellen kann es schwierig sein, eine ausreichende Anzahl von kollisionsfreien Samples zu finden. Es ist dann möglicherweise nötig, ausgeklügeltere Techniken einzusetzen, um eine gute Performance der Algorithmen zu erreichen.

Die vorliegende Arbeit gliedert sich in zwei Teile: Im ersten Teil untersuchen wir parallele Kollisionserkennungsalgorithmen. Da wir auf eine Anwendung bei sampling-basierten Bewegungsplanern abzielen, wählen wir hier eine Problemstellung, bei der wir stets die selben zwei Objekte, aber in einer großen Anzahl von unterschiedlichen Lagen auf Kollision testen. Wir implementieren und vergleichen verschiedene Verfahren, die auf Hüllköperhierarchien (BVHs) und hierarchische Grids als Beschleunigungsstrukturen zurückgreifen. Alle beschriebenen Verfahren wurden auf mehreren CPU-Kernen parallelisiert. Darüber hinaus vergleichen wir verschiedene CUDA Kernels zur Durchführung BVH-basierter Kollisionstests auf der GPU. Neben einer unterschiedlichen Verteilung der Arbeit auf die parallelen GPU Threads untersuchen wir hier die Auswirkung verschiedener Speicherzugriffsmuster auf die Performance der resultierenden Algorithmen. Weiter stellen wir eine Reihe von approximativen Kollisionstests vor, die auf den beschriebenen Verfahren basieren. Wenn eine geringere Genauigkeit der Tests tolerierbar ist, kann so eine weitere Verbesserung der Performance erzielt werden.

Im zweiten Teil der Arbeit beschreiben wir einen von uns entworfenen parallelen, sampling-basierten Bewegungsplaner zur Behandlung hochkomplexer Probleme mit mehreren "narrow passages". Das Verfahren arbeitet in zwei Phasen. Die grundlegende Idee ist hierbei, in der ersten Planungsphase konzeptionell kleinere Fehler zuzulassen, um die Planungseffizienz zu erhöhen und den resultierenden Pfad dann in einer zweiten Phase zu reparieren. Der hierzu in Phase I eingesetzte Planer basiert auf sogenannten Expansive Space Trees. Zusätzlich haben wir den Planer mit einer Freidrückoperation ausgestattet, die es erlaubt, kleinere Kollisionen aufzulösen und so die Effizienz in Bereichen mit eingeschränkter Bewegungsfreiheit zu erhöhen. Optional erlaubt unsere Implementierung den Einsatz von approximativen Kollisionstests. Dies setzt die Genauigkeit der ersten Planungsphase weiter herab, führt aber auch zu einer weiteren Performancesteigerung. Die aus Phase I resultierenden Bewegungspfade sind dann unter Umständen nicht komplett kollisionsfrei. Um diese Pfade zu reparieren, haben wir einen neuartigen Planungsalgorithmus entworfen, der lokal beschränkt auf eine kleine Umgebung um den bestehenden Pfad einen neuen, kollisionsfreien Bewegungspfad plant.
Wir haben den beschriebenen Algorithmus mit einer Klasse von neuen, schwierigen Metall-Puzzlen getestet, die zum Teil mehrere "narrow passages" aufweisen. Unseres Wissens nach ist eine Sammlung vergleichbar komplexer Benchmarks nicht öffentlich zugänglich und wir fanden auch keine Beschreibung von vergleichbar komplexen Benchmarks in der Motion-Planning Literatur.

# Danksagung

Ich danke ganz herzlich meinem Betreuer und Doktorvater für seine Unterstützung, seine Begeisterung für die Themen dieser Arbeit, die mich stets aufs neue motiviert hat, seine solide Zuversicht, sowie für seine Hilfe und Anleitung in den letzten Jahren.

Im selben Maße danke ich der Zweitgutachterin und Betreuerin dieser Arbeit. Danke für viele wertvolle Hinweise und Anregungen. Darüber hinaus danke ich ihr, dass sie mir ermöglicht hat in ihrem Forschungsprojekt mitzuarbeiten, für das Vertrauen, das sie mir hierbei entgegengebracht und die Freiheiten, die sie mir in dieser Zusammenarbeit eingeräumt hat.

Mein besonderer Dank gilt meinen Kollegen. Danke für unzählige Diskussionen, Anregungen und Kritik aber auch Ermutigung und Begeisterung.

Nicht zuletzt möchte ich meiner Familie und meiner Freundin danken, die in den letzten Jahren viel Geduld mit mir beweisen mussten, wenn es mal etwas schleppender voran ging. Vielen Dank für Eure Hilfe und Eure Unterstützung.

# Contents

# 1 Introduction

Motion Planning is an important problem that has many practical applications in the fields of Computer Aided Design and Modeling (CAD/CAM). In industrial manufacturing, complex machinery is designed and constructed entirely with the help of computer systems. In addition to engineering drawings, the individual parts of the desired product are then available as 3D models in a virtual 3D environment. In the automotive industry as an example, this leads to digital mock-ups of an entire vehicle in an early phase of the conception. An important task is to assure that the individual parts of the model can be physically assembled (or disassembled) and how this can be achieved. It is of course desirable to decide these questions in an early prototyping phase and surely before the fabrication of physical models. Here, motion planning algorithms are of great use to automatically plan and verify collision free (dis-)assembly paths for the different components of a complex 3D model. The motion planning algorithm serves as a tool that allows the engineer to simply specify what parts he wants to be removed instead of tediously finding a disassembly path by hand. As Steven LaValle says in his book *Planning Algorithms* [29], motion planning algorithms help to "[...] convert high-level specifications of tasks from humans into low-level descriptions of how to move."

Amongst the classes of different motion planning algorithms, sampling-based techniques have shown themselves to be particularly useful to plan paths for a complex 3D object (the *robot*) in a 3D environment while avoiding collisions with some set of *obstacles*. This is sometimes referred to as the *piano mover's problem*. These algorithms incrementally build a map of the collision free positions of the robot by repeatedly sampling different (random) positions. There are many different ways to choose the samples and how to connect nearby samples to reveal the topological structure of the set of samples. These choices result in different motion planning strategies.

Historically, most motion planning algorithms are serial but due to the recent evolution of parallel compute devices, serious efforts have been made to design parallel motion planners. One critical step in sampling-based motion planning is collision detection. Virtually all sampling-based algorithms generate some sequence of random positions of the robot and utilize a collision checker to decide if a given sample corresponds to a collision free position. Collision detection typically consumes by far the biggest fraction of the overall running time. This is especially true if robot and obstacle are complex or detailed geometric objects. Hence, this is also a good operation where we can profit from parallelization. Hence, regardless of the actual planning strategy, it is worthwhile to carefully analyze the underlying collision tests (and particularly parallel collision tests)

so that we always have the right tool in hand when we want to design a motion planning algorithm. This is one topic we want to address in this work.

A problem with sampling-based motion planning can be *narrow passages* that appear whenever the movements of the robot are highly restricted by the obstacles (moving the piano through a stairway). This is a serious problem as these situations often appear in practice. For example, think about the many different, tightly packed components that are built in the engine bay of a car. Based our work on parallel collision detection, we have designed a novel planning algorithm that is especially suited to solve very complex motion planning problems with many narrow passages. This is the second big topic of this work. This implicates that, the present work is organized in two parts:

The first part deals with parallel collision detection of two rigid body objects that are given as two triangle sets. With regard to an application in motion planning, we formulate the basic collision queries as having two triangle sets (robot and obstacle) and and a sequence of rigid body transformations (samples). A particular transformation is used to transform one of the objects (the robot) and we want to perform a collision test for every relative placement of these same two objects. After the introduction, we give a brief overview of Bounding Volume Hierarchies (BVHs), which are one of the most popular acceleration structures for collision detection and other proximity queries. We then describe the elementary proximity tests we have used to implement bounding volume hierarchies, namely collision and distance tests for a pair of triangles and a pair of bounding volumes. The tests we describe in this section fall back on a result known as the Separating Axis Theorem (SAT) but we also sketch how the SAT can be combined with a very simple feature-based distance test to yield a more sophisticated test with only small overhead. The next section gives a quick refresher on CUDA, before we describe the details of our BVH-based collision test implementations. We have implemented and compared four different CPU implementations: two OBB trees, an AABB tree and what we call a box-tree which can be seen as an overlap-free AABB tree. All CPU collision queries have been parallelized over the main loop that iterates over the sequence of rigid body transformations. On the GPU-side, we have implemented and compared four CUDA kernels that differ in the granularity of the distribution of the work amongst the available GPU threads. There has been plenty of work on how to write well-behaved CUDA code that respects the restrictions given by the hardware in terms of memory accesses, thread divergence, etc. But only based on theoretical considerations, it is hard to tell what will be the best approach for all but the simplest kernels. So we went to experimentally compare our different implementations to show what works well and does not work well in the context of BVH-based collision detection. The results of this work have been presented on the conference series “Facing the Multicore-Challenge” in 2012 [6]. In addition to different work distributions, we have also described and tested a novel memory access pattern we called Coalesced Scattered Reads (CSR) to efficiently load bounding volumes on the GPU during BVH traversal.
As another alternative approach on the CPU, we describe a grid-based collision detection algorithm. This test can be seen as a hierarchical version of the known voxmap-pointshell algorithm. Compared to the BVH-based collision tests, the grid-based approach has the

additional advantage that it is possible to perform an offset operation on the grid and slightly shrink the query objects. Grid-based approaches offer many different possibilities for efficient approximate collision tests. We present different approximate tests and compare them to BVH-based approximate tests.
We complete the first section with a benchmarking and comparison of the presented algorithms. We now have a whole bunch of different collision tests and can begin to design efficient motion planning algorithms.

The algorithms and data structures that have been created during our work on collision detection are not only useful in the application of sampling-based motion planning. Throughout our research on the topic, we have worked on the RASAND research project at the HFT Stuttgart. The project has the goal to build a software library of robust and efficient algorithms for proximity queries between very large triangle sets. So it was also a concern of how to best integrate our results in this developing library.

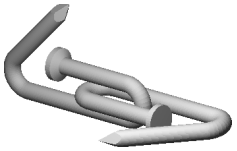

Fig. 1.1: The objective of the nails puzzle is to automatically separate the intertwined nails.

The second part of this work deals with sampling-based motion planning. Having many practical applications in CAD/CAM, we originally wanted to go in for solving the rather academic problem of the "nails puzzle": separate the two intertwined nails shown in Fig. 1.1. Having no practical meaning at first sight, this shows to be a perfect benchmark scenario for a very hard six degrees of freedom rigid body planning problem. In this case, the obstacle only allows very restricted movements and the robot has to travel several narrow passages on its way to a goal position. Most interestingly, and unlike to many other motion planning problems, it is not obvious for a human being how this puzzle can be solved. Over time, we added further metal puzzles, including the famous alpha puzzle, and ended up with a bunch of very hard motion planning problems. To the best of our knowledge, a comparably complex set of motion planning problems is not publicly available and has not been described in the motion planning literature. It is no surprise that we had not been able to solve these problems with the know standard planning algorithms. It took us several attempts with different motion planning strategies, ranging from Probabilistic Roadmaps, tree-based approaches like Rapidly-exploring Random Trees and hybrid approaches with roadmaps of trees to arrive at the algorithm we want to describe in this work. It is beyond the scope of this work to describe our other attempts as well, even if some of them have been successful in solving our benchmarks, but the reader should note that the final result we present here has been experimentally improved, adapted and redesigned over many iterations. This approach has only been possible due to the fundamental work, we have done in the first part of this work. The individual parts that assemble the whole algorithm may not seem to be highly inventive by their own. But we have carefully chosen, adapted and combined these components to yield a strong motion planning algorithm that is able to solve the complete family of metal puzzle benchmarks. One of the main novelties of our approach is how we have put these pieces together.
The chapter on sampling-based motion planning starts with a brief introduction and problem formulation. In the following section, we describe how to uniformly sample the orientations of the robot. While this might be clear for the translational degrees of



freedom, it is surely non-trivial to pick a random element of the rotation group SO(3).
We then gradually describe the individual components of our motion planning algorithm. The algorithm works in two phases. Phase I is based on a tree-based planning mechanism originally proposed by Hsu *et al.* that is now known as Expansive Space Tree (EST). We have modified this approach to give it a greedier behavior and parallelized it over multiple CPU cores. As an underlying collision test, we use the grid-based approach described in Sec. 2.7. We optionally allow to slightly shrink the robot by removing one layer of voxels of the grid representation. Not only does this simplify the planning process but it also speeds up the local planning by changing over to approximate collision testing. In order to improve the efficiency of the local planner, we utilize a simple retraction mechanism and thus actively integrate workspace information in the planning process.
Instead of using conservative error bounds for the solution paths by using an adequately small step size in the first planning phase, we explicitly allow non-conservativeness here in favor of a higher efficiency.
To fix the resulting non-conservative solution paths from phase I, we present a novel planning algorithm, Locally Biased RRT (LBRRT), that systematically re-plans the solution path only in a certain local neighborhood. In this second planning phase, we do not use approximate collision checking or sample retraction and use small step sizes to guarantee a good quality of the solution path.
Towards the end of the chapter, we analyze how the different parameters that are inherent to our approach and the individual components that compose to the overall planner influence its performance in our benchmark scenarios.

# 2 Parallel Collision Detection

## 2.1 Overview and Problem Statement

Collision detection and other proximity queries are used in computer simulated environments to determine the spatial relationship between geometric objects. They have many important applications in dynamics simulation, computer animation, computer aided design/manufacturing (CAD/CAM), motion planning and haptic rendering. Depending on the desired output and the representation of the tested objects, there exist a variety of different proximity queries, reaching from basic collision detection to more sophisticated tests. In this chapter, we want to examine proximity queries between two geometric objects that are represented as triangle sets. We divide proximity queries into the following classes:

- **Boolean Collision Query:** given two triangle sets, determine if there exists a collision between them, optionally identify one intersecting triangle pair
- **Extended Collision Query:** if the objects intersect, determine all intersecting triangle pairs,
- **Boolean Tolerance Query:** determine if the Euclidean distance of the objects is smaller than a given distance threshold $\theta$, we also call this an $\theta$-tolerance query,
- **(Approximate) Distance Query:** determine the Euclidean distance between the objects, optionally up to some absolute or relative error bound,
- **Extended Tolerance Query:** if the objects are closer than $\theta$, determine all triangle pairs that are closer than $\theta$.

For the extended versions of proximity queries, we can also think of a *simplified extended* version where we do not want to determine all *pairs* of triangles but rather a set of conflicting triangles for both objects with no pairing information.

These problems have been well analyzed over the last decades and special solutions exist for example for convex polytopes [8, 30]. We always want to assume that the query objects are given as triangle sets. Given a basic triangle triangle proximity test, one possible implementation of a proximity test between two triangle sets is to simply test all pairs of triangles. However, this results in an algorithm that has an asymptotic complexity similar to the product of the triangle count of the two objects and thus is impractical for complex models. Common collision and distance tests try to reduce the

total number of primitives that have to be tested by either using some sort of hierarchical object representation or space partitioning scheme.
There has been substantial work on using distance fields to perform proximity queries [19]. The easy structure of distance fields perfectly fits the way graphics cards process rendered pixel or fragment data in the render pipeline. Consequently, many of the early work on GPU-based proximity queries uses the rasterization capabilities of the graphics device to perform image space tests based on depth buffer or stencil buffer tests and distance field computation [34, 2, 22, 12, 11, 50, 33].
Bounding Volume Hierarchies (BVHs) have become a popular data structure for both collision detection and distance queries. They have been well analyzed and different types of bounding volumes and traversal schemes have been proposed to improve performance [43, 10, 24, 21]. Recent work has also been done to use BVHs for parallel collision queries on the GPU [27, 4, 39, 28] and multi-core CPU [20, 52].

In this work, we are mainly interested in the application of collision tests to sampling-based motion planning. We will therefore focus on the discussion of boolean collision queries. However, the techniques and ideas presented in this work can also be used to implement other proximity queries with moderate effort. We want to investigate how modern multi-core CPUs and many-core GPUs can be utilized to improve the performance of collision tests so that it becomes possible to solve very complex motion planning problems in moderate time. We analyze and compare two different data structures for collision detections. In Sec. 2.2, we use bounding volume hierarchies and in Sec. 2.7, we use a grid as a space partitioning scheme. For these data structures, we give a standard single core implementation that is extended to a multi-core CPU implementation. In addition, we describe and test four different implementations for BVH traversal on the GPU. We analyze the achieved performance in two benchmark scenarios and compare the different techniques in order to see where the strengths and weaknesses of the individual strategies are.

### Problem Formulation

Let $A$ and $B$ be two triangle sets that represent the query objects. We want to perform a number of boolean collision tests between the same rigid objects $A$ and $B$ in different relative positions. We specify the position of $A$ with a sequence $Q := (q_0, q_1, \ldots, q_{n-1})$ of rigid body transformations that we use to transform $A$ into world coordinates and denote the transformed triangle sets with $A(q_i)$. We always assume $B$ to be in the world coordinate frame. Depending on our implementation, the representation of the elements of $Q$ can change. We typically use either a $4 \times 4$ matrix $M \in \mathbb{R}^{4\times4}$ or a pair $(t, u) \in \mathbb{R}^3 \times \mathbb{H}$ of a transformation vector $t$ and a unit quaternion $u$ to represent an element of $Q$. This should be no source of confusion. Every element of $Q$ represents a relative position of the two objects $A$ and $B$. For example, this could be discrete samples on a motion trajectory or samples from a motion planning algorithm. Altogether, we define a problem instance as a triplet $(Q, A, B)$ where

| | |
|---|---|
| $Q = (q_0, q_1, \ldots, q_{n-1})$ | is a sequence of rigid body transformations, |
| $A$ | is a triangle set representing a moving object and |
| $B$ | is a triangle set representing a static object. |

Our algorithms will then compute an output sequence

$$R = (r_0, r_1, \ldots, r_{n-1}), \text{ with } r_i \in \{0, 1\}.$$

We want to interpret the result so that

$$r_i = 1 :\Leftrightarrow A(q_i) \cap B \neq \emptyset.$$

### Contribution

In the first part of this work, we implement and compare different collision detection techniques on both, CPU and GPU. We use different data structures on the CPU and compare different CUDA kernels on the GPU side. Our aim is to always keep the implementation as simple as possible while taking best advantage of the parallel compute capabilities to improve the throughput of our algorithms. Compared to other techniques described in literature, the beauty of our approach lies in its simplicity. We get along without heavy and difficult to implement work balancing schemes; Our algorithms do not require any pre-computations and do not rely on a certain distribution of the input configurations. The only assumption we want to make is that we have an input size that is sufficient to fully saturate the parallel device. Depending on whether this a CPU or GPU, we will see that the required input size ranges from a few hundred to hundreds of thousands of different input configurations.

We have taken good care to efficiently implement the resulting algorithms and to compare different implementations in the same benchmark scenarios. Not only do we present the



our best implementation, but we also show our second best implementations. In this way, the reader can see that different design choices do not turn out to be competitive and does not have to find out on his own.
In the chapter on GPU collision detection, we describe and benchmark a new memory access pattern to efficiently load bounding volumes from the GPU's main memory during a bounding volume hierarchy traversal. We use multiple threads to load one bounding volume node together to a shared memory cache instead of having every thread load a bounding volume on its own. The idea of using a shared memory cache to improve the memory access pattern is common practice in GPU programming, but the application of this idea in the context of BVH traversal and its realization has, to the best of our knowledge, not yet been discussed in the literature.

To some extent, the first part of this work provides the fundamentals to the second part on sampling-based motion planning. So one goal of our work on collision detection was to build a toolkit of different collision detection routines. We then had been free to design different motion planning algorithms and always have the right tool to handle the occurring collision queries. Similarly, we could have the same benefits from this work when using collision detection in other fields of application. In this respect, the first part of our work also has an intrinsic value.

## 2.2 Bounding Volume Hierarchies

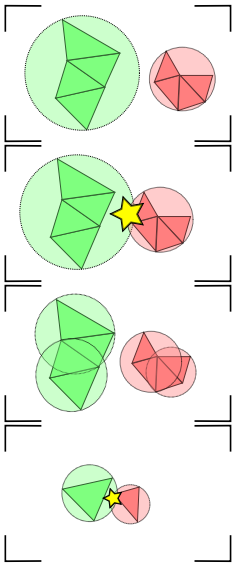

Fig. 2.1: Top to bottom: If BVs are disjoint, so are the bounded triangles. If BVs intersect, we can refine. For intersecting leaf nodes, test bounded triangles.

In this section, we give a short introduction to bounding volume hierarchies and how we can perform collision tests with this data structure. For a more in-depth discussion, a very good survey is given in [7], pages 235ff.

Bounding volume hierarchies are a widely used data structure to accelerate collision queries between rigid bodies that are represented as triangle sets. Amongst the common types of data structures for collision detection, bounding volume hierarchies have become a popular choice. They are easy to construct and maintain, require only a moderate amount of memory and can easily be transformed as the queried objects undergo rigid motion.

As we have stated earlier, a naive approach to test two triangle sets $A$ and $B$ for intersection would be to aggressively test all pairs $(a, b) \in A \times B$ for intersection. There are two problems with this approach: First, a single primitive test is comparatively expensive and more importantly, the number of pairwise tests grows dramatically for larger triangle sets.

Wrapping every triangle into a simple bounding volume (e.g. its bounding sphere) and testing the bounding volumes for intersection first is a common choice to reduce the average cost of a single primitive test. This can improve the running time for a collision query by a constant factor, but the overall number of pairwise triangle tests and thus the asymptotic time complexity remains the same. It is possible to reduce the number of pairwise tests by building a hierarchy of bounding volumes for both triangle sets. The bounding volumes enclosing the triangles become the leaf nodes in the hierarchy. Leaves that are in close proximity are clustered and merged into a single parent bounding volume. Recursively applying this procedure to the parent nodes results in a hierarchy of bounding volumes. It is important to note that a parent node does not necessarily have to bound its child volumes but the triangles enclosed in all of its children. Moreover, all bounding volumes should be chosen to fit the bounded primitives as tightly as possible. As far as collision queries are concerned, a bounding volume hierarchy has the following fairly obvious but important property:

> *If two bounding volumes do not collide,*
> *the bounded primitives cannot collide either.*

This allows to quickly prune away a large number of triangles that are not in close proximity and cannot collide. One single bounding volume test can replace a huge number of primitive tests.[1] If however two tested bounding volumes do collide, it is possible to descend the hierarchy and test all pairs of child volumes for overlap. Only if both volumes are leaf nodes, we have to test the contained triangles (see Fig. 2.1).

BVHs vs Grids

Bounding volume hierarchies differ from space partitioning schemes like grids or octrees

[1] In the best case we have to perform only a single bounding volume test to see that $A$ and $B$ are not colliding as opposed to $|A| \cdot |B|$ triangle triangle tests.



in several ways. First it is possible for two bounding volumes of the same hierarchy to overlap whereas all cells of a grid are disjoint. However, with bounding volume hierarchies one triangle is typically only contained in one bounding volume at a certain level of the hierarchy. Grids or octrees typically have to store every triangle in more than one cell. Another important difference is that bounding volume hierarchies are constructed in the objects local coordinate frame and can be easily transformed as the body undergoes rigid motion. Classical space partitioning schemes are attached to the world coordinate frame.

Implementation Concerns

There are numerous choices when it comes to implementing the abstract concept of a bounding volume hierarchy. When building a hierarchy there are different characteristics that are desirable to achieve. This could be the minimal volume and optimal fitting of the bounding volumes or the balance, memory layout and degree of the tree hierarchy. The objects used as bounding volumes can influence how good this characteristics can be achieved. Popular choices include spheres, axis-aligned bounding boxes (AABB), arbitrarily oriented bounding boxes (OBB), rectangular swept spheres (RSS) and k-DOPs [43, 10, 24, 21]. Another interesting and relatively new bounding volume type are so called Slab Cut Balls (SCBs) [25]. Easy bounding volumes (like spheres) can have very cheap intersection tests and transformation costs compared to more complex bounding volumes. But complex bounding volumes tend to fit the contained geometry more tightly. Gottschalk *et al.* tried to quantify what they call the "convergence rate" of their OBB hierarchies compared to sphere hierarchies. They claim that sphere hierarchies have a poorer approximation quality then OBBs. However, an OBB intersection test needs far more floating point operations than a sphere intersection test. For a more detailed discussion see [10]. The three half lengths of an OBB can be chosen independently and they can be arbitrarily oriented. This is especially beneficial in situations where the tested geometry comes close without actually intersecting and makes OBBs a popular bounding volume type for collision detection.

Cost Function

Weghorst *et al.* [56] and Gottschalk *et al.* [10] proposed to describe the cost of a bounding volume hierarchy traversal more formally by giving the following cost equation:

$$T = N_V \cdot C_V + N_P \cdot C_P + C_0.$$

Here, $N_V$ and $N_P$ reflect the number of bounding volume tests and the number of primitive tests (triangle triangle tests) tests that have to be performed per bounding volume hierarchy traversal, respectively. $C_V$ and $C_P$ are the costs of a single test and $C_0$ sums up the cost for all operations that have to be performed only once per traversal. This could be the initialization of the traversal stack or an initial coordinate transformation. As we have seen, those values are intrinsically linked which makes it difficult to optimize the equation. For example, reducing the cost for a single bounding volume test $C_V$ usually affects the overall number of tests $N_V$ that have to be performed. But it can also increase the number of primitive tests $N_P$.

Construction

Different mechanisms have been proposed in literature to construct bounding volume

hierarchies either top-down [10], bottom-up [38] or incrementally. Top-down approaches are easy to implement, fast and result in good hierarchies which makes them very popular. Bottom-up construction can result in better hierarchies but are more complicated to implement and more expensive to construct. Incremental approaches allow to dynamically grow the hierarchies but they do not provide a very tight fitting or have to be partially rebuilt from time to time. Again, a very good survey can be found in [7]. The detailed investigation of different construction schemes is beyond the scope of this work and we will use a popular top-down construction mechanism for our hierarchies.

Query and BVTT

In the query phase, bounding volume pairs are tested for overlap. Whenever a pair of bounding volumes intersects, its child nodes have to be tested as well. Every bounding volume pair $(a, b)$ can be interpreted as a node in a tree, whose child nodes are the pairs of bounding volumes that have to be inspected if $a$ and $b$ intersect. This results in the so called bounding volume hierarchy traversal tree (BVTT) [24]. Traversing a pair of bounding volume hierarchies can thus be seen as finding a leaf node with intersecting bounding volumes in the corresponding traversal tree. The order in which the bounding volume tests are performed and whether one or both of the intersecting nodes are refined offer plenty different traversal schemes.

Common schemes are to search the BVTT in breadth first or in depth first order (BFS, DFS). If we implement DFS with a stack and BFS with a queue, we have to choose in which order to put the next nodes on this data structures. It is possible to use information that is gained during the bounding volume tests to influence this order. For example, we could try to use an order in which the bounding volume pairs that are near will be processed first. This is called an *informed traversal scheme* as opposed to an ordering that is independent of the bounding volume tests. Furthermore, it is possible to use the stack/queue to store all bounding volume pairs that still have to be tested for intersection (the children of a positive intersection test) or we could test the children first and store just the bounding volume pairs that have been shown to overlap.
It is not always necessary to refine both nodes for an intersecting bounding volume pair. It can also be promising to descend only into one hierarchy. For two bounding volume hierarchies $A$ and $B$, this results in different *descent rules*. Possible choices are: descend $A$ before $B$, descend round robin, descend the bounding volume of the larger volume, descend all, etc. It is not clear what is the best choice and there is no clear winner. The performance of a descent rule depends entirely on the structure of the data.
Traversing the BVTT in a breadth first fashion can result in a very high memory consumption. It also leads to dispensable work if we want to interrupt the traversal at the first contact point found. This is why depth first search is used in most bounding volume hierarchy traversal schemes. On the other hand, the small number of nodes that are present simultaneously during DFS can be a problematic if we think about parallelization if a single traversal with many threads. In [13], the authors have proposed a combined mechanism for hierarchy construction (PBFS: partial breadth-first search). A similar idea could also be used for BVH traversal.
By using a priority queue to store all pending BVTT nodes that have to be visited, it is possible to always pick the most promising one. But the extra cost for managing this



data structure does not necessarily have to pay off. The extra time spent to determine the best node at every step could be used by a simpler scheme to perform more tests.

A widely used public library for collision, distance and tolerance queries is PQP (for Proximity Query Package) that was developed by Eric Larsen, Stefan Gottschalk *et al.* in 1999 and extends the RAPID package. It uses trees of oriented bounding boxes (OBBs) for collision detection and rectangular swept spheres (RSS) for distance computation and tolerance verification. OBBs have become a popular choice for collision detection and Gottschalk *et al.* have shown that OBB trees can outperform AABB trees for collision queries, especially in close proximity scenarios [10]. We want to use PQP as a reference to compare our approaches with existing work.

The reader should note that for queries between rigid bodies, the hierarchies can be pre-computed once and traversed at runtime. For example, this is the case when performing collision queries in the fields of motion planning and path validation in CAD/CAM. These problems can be formulated as having two geometric objects, given as triangle sets, and a number of rigid body transformations that describe the relative position of the objects. We then want to detect the collision status of the objects for every relative placement.

**Parallel BVH Traversal**
We want to investigate how a high number of BVH-based collision tests can be efficiently performed in parallel on multi-core CPUs and many-core GPUs. Our goal is to improve the performance of high level algorithms for sampling-based motion planning, where collision testing consumes a significant amount of the overall running time. We have used CUDA to implement and compare several different alternatives to distribute the work to the parallel GPU threads. As a reference, we have used OpenMP to parallelize the collision queries on the CPU.

There has also been substantial work on how to exploit the parallel compute capabilities of the GPU to improve collision tests and other proximity queries. Historically, many GPU-based collision checkers use the rasterization capabilities of the graphics device and perform image space tests based on depth buffer or stencil buffer tests and distance field computation [34, 2, 22, 12, 11, 50, 33].

More recent work has focused on parallel BVH construction [27] and traversal [4, 39, 28] on the GPU and multi-core CPUs [20, 52]. Lauterbach et al. [28] describe a proximity query framework that is applicable to (continuous) collision detection, distance computation and self intersection. They treat a bounding volume intersection test a basic task and describe a load balancing scheme to distribute these tasks evenly amongst all GPU cores. The authors in [42] aim at performing a high number of collision tests in parallel in the field of sampling-based motion planning. They intent to cluster similar transformations of the queried objects (that lead to similar BVH traversal) to achieve a performance gain.

Compared to previous work, our goal is to execute a high number of independent collision tests in parallel. Instead of using a sophisticated load balancing scheme or pre-processing

of the input transformations, we rely on the high number of tests to evenly utilize the GPU cores while keeping the implementation complexity at a minimum. We parallelize the problem at different granularities and analyze how the achieved collision test throughput varies with the total number of tests per parallel collision query.



## 2.3 Elementary Proximity Tests

In this section, we have a look at the elementary proximity tests we need in order to implement the collision detection algorithms described in the upcoming sections. As we are going to use OBB hierarchies of two triangle sets, we have to discuss how to test two OBBs and two triangles for intersection at least. Towards the end of this work, we propose to use continuous collision tests to conservatively validate a disassembly path that has been constructed with a motion planning algorithm. We will then need to perform distance computations instead of collision tests to compute conservative motion bounds. To give an idea of how this can be achieved, we also sketch how we compute the distance between two rectangles (to determine the distance between two RSS bounding volumes) and the distance between two triangles.

### 2.3.1 Intersection and Distance of Two Triangles

Let A and B be two triangles in $\mathbb{R}^3$ with $a_0 \in A$ and $b_0 \in B$ two closest points, i.e.

$$d(a_0, b_0) = \min_{a \in A, b \in B} d(a, b).$$

We want to define the vector

$$\delta(A, B) := b_0 - a_0$$

to be the vector that connects two closest points $a_0 \in A$ and $b_0 \in B$. Even though the points $a_0$ and $b_0$ don't have to be unique, the reader should convince himself that the definition of the vector $\delta$ is independent of the choice of different pairs of closest points.[2] The vector $\delta(A, B)$ is either perpendicular to the plane of A or $a_0$ has to be a border point of A. The same condition also holds for $b_0$ and $B$. Likewise, if $a_0$ is a point on an edge $e_\circ(A)$ of A, either $\delta(A, B)$ is perpendicular to $e_\circ(A)$ or $a_0$ is a border point of $e_\circ(A)$, i.e. a vertex of A.

We decompose the triangle A (and B) into seven *features*, namely the triangle's face, its 3 edges and its 3 vertices (cf. Fig. 2.2). All these subsets are taken without borders, so that A is the disjoint union of its features:

$$\begin{aligned} A &= f(A) \,\dot\cup\, e_0(A) \,\dot\cup\, e_1(A) \,\dot\cup\, e_1(A) \,\dot\cup\, v_0(A) \,\dot\cup\, v_1(A) \,\dot\cup\, v_2(A) \\ &=: f_0^A \,\dot\cup\, f_1^A \,\dot\cup \ldots \dot\cup\, f_7^A \end{aligned}$$

It is now easy to see that a pair $(A, B)$ of triangles is either intersecting or we can find its distance by consecutive determination of the distances between all possible pairs

[2] If A and B are arbitrary sets, $\delta(A, B)$ is not necessarily well-defined.

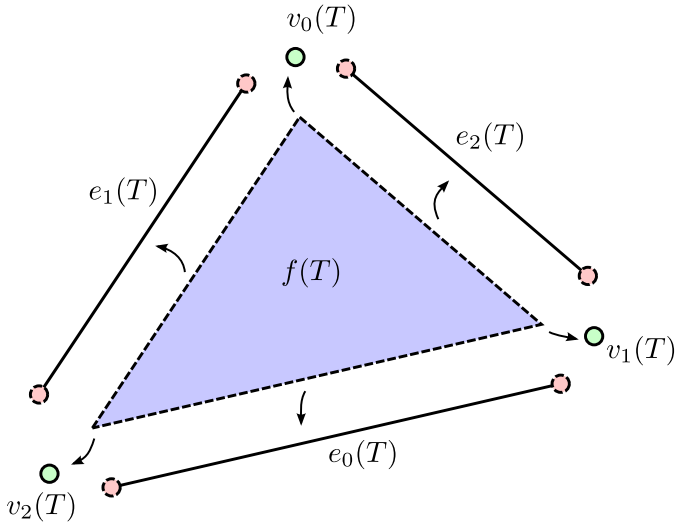


Fig. 2.2: We decompose a triangle $T$ into 7 features: Its face $f(T)$, which is the triangle without border, three edges $e_{0,1,2}(T)$ without the start and end points and three vertices $v_{0,1,2}(T)$.

$(f_i^A, f_j^B)$ of a feature of A and a feature of B. As all features are subsets of the triangles, $f_i^A \subset A$ and $f_j^B \subset B$, it always holds that

$$d_{ij}(A, B) := d(f_i^A, f_j^B) \geq d(A, B).$$

And because the union of all features covers the triangles, there is at least one feature pair $(f_{i_0}^A, f_{j_0}^B)$ that realizes the distance:

$$d(f_{i_0}^A, f_{j_0}^B) = d(A, B).$$

Clearly, the minimum of this finite number of feature distances is the distance of the two triangles. This results in an algorithm that is easy to understand and to implement. We refer to this procedure as *feature based distance computation*. It is the basic idea behind many triangle triangle distance algorithms.
More sophisticated approaches aim to use the spatial information that can be derived by a feature test in order to rule out other feature tests [5]. This can result in complex decision trees with many branches and makes the algorithm less readable and maintainable. There also exist several algorithms to compute the distance between arbitrary convex polytopes, like the GJK or the Lin-Canny algorithm, that could be used to determine the distance between two triangles [8, 30].

We want to follow a different path to improve the feature based distance test.An interesting fact and direct result from the feature based approach is the following: For a pair of triangles A and B, there is only a finite number of directions that the distance vector $\delta(A, B)$ can point to. These are the directions of the distance vectors of the feature



pairs:

$$\delta_{ij}(A, B) := \delta(f_i^A, f_j^B).$$

For every direction $\delta_{ij}$, we can now orthogonally project A and B to an arbitrary line with the same direction vector. This results in two intervals $I_A$ and $I_B$ on the line. We refer to the distance of these intervals as the *directed distance* of A and B in the direction of $\delta_{ij}$:

$$\Delta_{ij}(A, B) := \min_{a \in A, b \in B} \left\{ \frac{\delta_{ij}(A, B)^\top (b - a)}{|\delta_{ij}(A, B)|} \right\}.$$

Obviously, the directed distance of A and B is always smaller than the Euclidean distance:

$$\Delta_{ij}(A, B) \leq d(A, B).$$

If $\Delta_{ij}(A, B) > 0$, the triangles are shown to be disjoint and the corresponding line is called a *separating axis*. Conversely, if any of the directions $\delta_{ij}$ defines a separating axis, we can conclude that the triangles are intersecting.

Triangle Intersection

It can be shown that for a collision test, the vertex-vertex directions are redundant. It is sufficient to test the $3 \times 3$ edge-edge directions (i.e. the $3 \times 3$ cross products of one edge of A and one edge of B) and the 2 faces' normal directions. In the degenerate case of coplanar triangles, the edge-edge cross products are equal to the normal directions and do not serve as separation axes. It is then necessary to also test all directions that result as a cross product of an edge and the normal of the same triangle (resulting in $2 \times 3$ additional directions). This result is known as the *separating axis theorem* and the corresponding collision test is referred to as the *separating axis test* [9].

Triangle Distance

Our basic idea for the design of a triangle distance test is to combine the feature based approach with the separating axis test and simultaneously execute both tests. For every tested feature pair $(f_i^A, f_j^B)$, the distance of the two features serves as an upper bound to $d(A, B)$, their directed distance in the direction of $\delta_{ij}$ serves as a lower distance bound:

$$\Delta_{ij}(A, B) \leq d(A, B) \leq d_{ij}(A, B).$$

A distance algorithm could now iterate over all feature pairs and successively attempt to increase a lower distance bound $d_{min}$ with $\Delta_{ij}$ and to decrease an upper distance bound $d_{max}$ with $d_{ij}$. The algorithm can terminate if it determines that $d_{max} \leq d_{min}$. In this case we have identified the feature pair that realizes the distance between A and B and we do not have to test the remaining pairs. It then holds that $d_{max} = d(A, B) = d_{min}$.

Beyond its application as a distance test, the algorithm can also be used as a tolerance test, i.e. to determine if $d(A, B)$ is below or above a certain tolerance value $\theta \in \mathbb{R}$. In

this case, we do not have to calculate the exact distance value but the algorithm can already terminate if it determines that either

$$d_{max} \leq \theta, \text{ i.e. tolerance violation or}$$
$$d_{min} \geq \theta, \text{ i.e. no tolerance violation.}$$

### 2.3.2 Intersection of Two OBBs

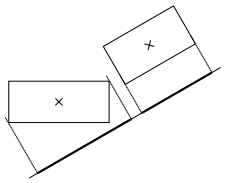

Fig. 2.3: A separating axis for two rectangles in $\mathbb{R}^2$.

The separating axis theorem can also be used to implement an efficient and robust intersection test for arbitrarily oriented boxes. This was proposed by Gottschalk and is explained in detail in [9]. In short, we can summarize the statement as follows: If for any two sets $A, B \in \mathbb{R}^3$ we find a plane $E$ that separates them, then A and B are disjoint. For *convex* objects, the converse is also true. If A and B are disjoint, there exists a plane that separates them. Instead of testing if a plane separates the objects, we can equivalently test if the projections of both objects on an axis that is perpendicular to the plane are disjoint (see Fig. 2.3). Such an axis is called a *separating axis*. The separating axis theorem states that for convex polytopes, it is sufficient to test a finite number of axes, namely all directions that are either a normal of a face (of either A or B) or the cross product of one edge of A and one edge of B. This leads to an easy intersection test: For every candidate axis, check if it is a separating axis. If so, the objects are disjoint. If we do not find a separating axis amongst the candidate axes, the objects intersect.
In the previous section, we have already sketched how we use the separating axis theorem to implement an intersection test for triangles. The test is especially interesting for an OBB intersection as well, because several face normals and edges of an OBB point to the same direction. An OBB has 6 faces and 12 edges. For two arbitrary polytopes this would result in $6 + 6 + 12 \times 12 = 156$ candidate axes that had to be tested in the worst case. Taking identification of equal directions into consideration, we only have to test at most $3+3+3\times 3 = 15$ different directions! If we represent an OBB with its center point $c$, three orthogonal unit vectors $r_1, r_2, r_3$ to describe the orientation of the box and three half-lengths $l_1, l_2, l_3$, we can elegantly determine the interval $[L, R]$ that results from the projection on an axis with direction vector $d$:

$$\begin{aligned} L &= \min \left\{ \; d^\top (c \pm l_1 r_1 \pm l_2 r_2 \pm l_3 r_3) \; \right\} \\ &= d^\top c - \sum_{i=1}^{3} l_i \; |d^\top r_i| \end{aligned}$$

and

$$\begin{aligned} R &= \max \left\{ \; d^\top (c \pm l_1 r_1 \pm l_2 r_2 \pm l_3 r_3) \; \right\} \\ &= d^\top c + \sum_{i=1}^{3} l_i \; |d^\top r_i|. \end{aligned}$$



If we define $\mathfrak{r} := (\ |d^\top r_1|,\ |d^\top r_2|,\ |d^\top r_3|\ )^\top$ and $l := (\ l_1,\ l_2,\ l_3\ )^\top$ this can be written as

$$L = d^\top c - l^\top \mathfrak{r} \text{ and}$$
$$R = d^\top c + l^\top \mathfrak{r}.$$

In Sec. 2.5.2 we are going to traverse two AABB hierarchies, where all boxes of the same hierarchy have the same orientation. In this case, the vectors $\mathfrak{r}$ are constant and can be precomputed for all 15 separating axis directions.

### 2.3.3 Distance of Two Rectangles

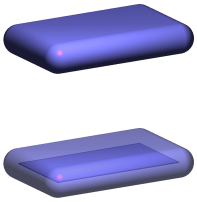

Fig. 2.4: A RSS volume can be generated by offsetting a supporting rectangle.

In Sec. 3.10, we want to perform continuous collision tests between two triangle sets that move along a trajectory. This requires to efficiently calculate the distance between the triangle sets to compute conservative motion bounds. To achieve this, we are going to use bounding volume hierarchies as described for collision tests in Sec. 2.2. The distance computation algorithm is very similar to a BVH-based collision test. Essentially, we only substitute a collision test between two bounding volumes for a distance test and likewise, we use distance tests between two triangles. We have already seen how we can compute the distance between two triangles. OBBs are not especially designed for distance tests and are better suited for collision checking. This is why Larsen *et al.* [24] proposed to use what they call Rectangular Sphere Volumes (RSS) in the context of distance tests. A RSS is a rectangle in $\mathbb{R}^3$, that is offset by a certain radius $r$ (see Fig. 2.4). To get the distance between two RSS volumes, we can then simply determine the distance between their supporting rectangles and subtract the radii.
To determine the Euclidean distance between two Rectangles A, B in $\mathbb{R}^3$, we can use the same algorithmic idea we have described for triangles above. The features of a rectangle are its four vertices, four edges and one face. For every pair $(f_i^A, f_j^B)$ of one feature of A and one feature of B, we determine the Euclidian distance $d_{ij}$ and directed distance $\Delta_{ij}$. We use these distances to update the distance bounds $d_{min}$ and $d_{max}$. If we observe that $d_{max} \leq d_{min}$, we have found the feature pair that realizes the distance and can stop the procedure.
Like in the case of the OBB intersection test, we can profit from the fact that a rectangle only has two different edge-directions. All together, there are 16 vertex-vertex directions, 2×8 vertex-edge / edge-vertex directions, 4 edge-edge directions and 2 face-directions that have to be tested. Just like in the case of triangles, we could also perform tolerance tests with rectangular swept spheres in the same manner.

## 2.4 A Quick Refresher on CUDA

Most of the information in the following section is taken from the NVIDIA Fermi Compute Architecture Whitepaper [37] and the NVIDIA Cuda Proramming Guide [36]. The section summarizes information about Cuda that might be helpful to understand our GPU-based collision tests described in Sec. 2.6. For more detailed information, the interested reader may consult these references directly. The NVIDIA Fermi Compute Architecture Whitepaper reads

> "CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages."

The graphics processing unit (GPU) is a specialized processor that was designed to support hardware accelerated rendering in graphics applications. The term was first defined by Nvidia in 1999 with the release of the GeForce 256. Designed to process a high amount of independent vertex and fragment data, GPUs became highly parallel devices. They were equipped with special vertex and pixel pipelines that ran shader programs on an input data stream in a data parallel manner.

Efforts have been made to exploit this resource for non-graphical applications as well. This was done by expressing the problems in terms of vertex or fragment coordinates, textures and shader programs. The input data was encoded in one or more textures which were then processed by specialized shader programs. This efforts to use the graphics API for general purpose computing is referred to as GPGPU computing. Main drawbacks of this approach are that there are severe limitations in what can be expressed with a shader program. For example, it was not possible to randomly access main memory and there was no possibility for inter-thread communication. Moreover, the formulation of a problem in terms of a rendering process can lead to a complex and obscure program structure and programmers needed a good understanding of the graphics API and the rendering pipeline.

With the introduction of the G80 unified graphics and compute architecture in 2006, Nvidia eliminated some of this drawbacks and tried to simplify the development of GPU applications. With the new G80 - series, Nvidia changed the design of the GPU and exposed its functionality to the user through a high level programming language. The separate vertex and pixel pipelines were replaced with a single processor that now executes vertex, geometry, pixel and computing programs. Every single core acts as a scalar processors which eliminates the need to manually manage vector registers. Instead, Nvidia introduced the "single instruction multiple thread" (SIMT) concept where a number of independent threads can execute concurrently using a single instruction. Inter-thread communication was enabled by the use of shared memory and barrier synchronization. Developers could now write C programs with CUDA extensions to target the GPU. The marketing term used by Nvidia to signify this new way of GPU programming is *GPU computing*.



In June 2008, Nvidia released the second generation unified architecture with the GT200-series of graphics chips. This new architecture offered more CUDA cores, more register space and better hardware memory coalescing support. It added new instructions for atomic memory operations and double precision floating point arithmetic.

From early 2010, Nvidia introduced a number of graphics chips based on a new hardware architecture code-named *Fermi* which is, at the time of writing, the most advanced family of Nvidia GPUs.[3] With Fermi, Nvidia tried to satisfy the needs of the GPU programming community, worked on the performance of double precision and atomic operations and added ECC support to detect memory failures. They enhanced the memory architecture with a true cache hierarchy and a magnified shared memory bank. A unified memory space enabled support for true function calls and C++ features.

### The hardware design of current Nvida GPUs

In this work we are concerned with programming the Fermi-series of Nvidia GPUs. Most of the described terms and amounts of hardware resources will be given for Nvidia GPUs with compute capability 2.x. Please refer to the Nvida Fermi Whitepaper [37] or the Nvidia CUDA Programming Guide [36] for further information.

Thread organization

From a programmers point of view, the GPU can be utilized by writing parallel CUDA kernels. A kernel is a function written in C with CUDA extensions that executes in parallel across a number of parallel threads. All threads are organized in *blocks* of threads and a grid of thread blocks. Every thread executes the same kernel function and can be distinguished from all other threads by a unique thread ID within its block. Threads within the same block can communicate through barrier synchronization and shared memory. A thread block has a unique block ID within its grid. A grid is an array of thread blocks and the threads within one grid synchronize between different kernel calls.

Memory architecture

Every thread has its own private memory space for register spilling, function calls and automatic array variables that are accessed dynamically trough the kernel execution. This is called the threads *local memory*. Each block owns a portion of *shared memory* that can be accessed by all threads of the same block. Shared memory can be used for inter-thread communication or to share common results. All threads in a grid share the same *global memory* space.

As Volkov *et al.* pointed out in [55] the ability to execute scalar threads and use arbitrary memory access patterns offers an extensive flexibility in CUDA programming. However, simply viewing the GPU as an array of independent scalar processor cores can have severe performance implications. In order to avoid pitfalls and achieve a high performance, it is important to algorithmically expose and to express parallelism as it is required by the hardware and to have an understanding of the memory system of the GPU.

[3] During the review of this work, the new *Kepler* family of Nvidia GPU became publicly available.

Streaming multiprocessors

The GPU consists of an array of *streaming multiprocessors* (SM). Each multiprocessor has 32 scalar processors that are referred to by Nvidia as *CUDA cores*. A CUDA core executes one floating point or integer instruction per clock for one thread. For execution, threads are grouped together in sets of 32 consecutive threads called *warps* and one SM can process one warp in parallel by fetching the next pending instruction and executing the same instruction on all threads of the warp. Volkov *et al.* propose to regard a streaming multiprocessor a SIMD unit with the scalar processors being its SIMD lanes. They also use the term SIMD core for a SM. Practically, one SM can only execute one warp *simultaneously* at a given time. But the warp scheduler of the SM can manage a number of warps *concurrently*. The ratio between the number of threads that run concurrently on a SM for a given kernel configuration and the maximum number of threads that can be handled by the thread scheduler is called *occupancy*.

### Latency hiding

TLP vs ILP

Every time the execution of one warp stalls, the warp scheduler performs a context switch and executes another warp. This happens for example when the input operands of the next instruction depend on the result of prior instructions. In the case that all involved operands are registers, latency equals the execution time of the pending instruction which can be up to about 22 clock cycles. But if one operand resides in off-chip memory, latency can be as high as 400 to 800 clock cycles. On devices with compute capability 2.0, the thread scheduler fetches one instruction per clock cycle, which means that there have to be at least 22 and up to 400 active warps per SM to fully hide the latency with thread level parallelism (TLP). However, it is also possible to hide latency with instruction level parallelism (ILP) without any context switch. Volkov claims that high arithmetic and memory throughput can be achieved even at low occupancy when the code has enough independent instructions [54]. The total number of warps that have to be present to hide instruction and memory latencies depends on the structure of the executed code and it is even possible to achieve higher performance with lower occupancy in some cases.

This concept is different to modern CPUs that try to hide latencies with prediction, caching and pipelining. GPUs remove much of the logic for branch prediction, memory prefetching, etc. and rather use the resulting Die-space for arithmetic units. They use very lightweight threads as compared to CPUs that allow fast context switches.[4] In this manner, the GPU tries to hide memory latency with arithmetic. This has of course severe implications on the design of programs for the GPU as it relies on the fact that there is always enough independent work to be done. To best hide the memory latencies, GPU programs should have a high arithmetic to memory operation ratio and run enough independent threads. Programmers can best utilize the device when writing code with

[4] The execution context of every thread is kept in the register file for its whole lifetime which enables context switches with very low cost. Of course, this also means that the register file has to be shared amongst all active threads.



a high thread level parallelism (high occupancy) or a high instruction level parallelism (many independent operations).

### Memory system of the GPU

Device memory is physically divided into slow off-chip main memory and fast on-chip memory. Depending on how the memory is accessed and for what purpose it is used, memory is logically divided into the following regions:

*Global Memory:* Global memory is the device's main memory. Fermi devices accesses global memory through 64-bit GDDR5 memory controllers. With a memory clock of 1848 MHz and 6 memory controllers, this results in a theoretical peak memory throughput of

$$(2 \cdot 1848 \text{ MHz}) \cdot (6 \cdot 64 \text{ bit}) \sim 177 GB/s.$$

Global memory accesses are cached through L1 and L2 cache or through L2 cache only if L1 was disabled during compilation. Memory accesses that are cached in L1 and L2 cache are serviced with 128-byte memory transactions and memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Memory requests are performed per warp like any other instruction.[5] If the size of the words accessed by each thread is too big or the threads of the warp access scattered memory addresses, the request is split into as many separate 128-byte (or 32-byte) memory requests as necessary to fully cover the required data with aligned 128-byte (32-byte) memory patches. This can result in memory over-fetching and reduce the effective memory bandwidth for an application. In the best case, the thread of a warp accesses consecutive 4 byte words, resulting in just one 128-byte memory transaction. If however the words accessed by each thread all lie in different cache lines, the memory access would result in 32 independent 128-byte memory transactions.

*Local Memory:* Every thread has 512 KB of its own private *local memory* it can use for register spilling, dynamically accessed arrays and function call stacks. Local memory resides in device memory and has the same high latencies. However, it is managed in a way that if all threads in a warp access the 4 byte word with the same relative address, the memory access will perfectly coalesce into a single memory transaction. Access to local memory is cached like any other global memory access.

*Shared Memory:* Every SM owns a portion of 64 KB very fast on-chip memory that can be partitioned in either 16 KB L1 cash and 48 KB shared memory or 48 KB L1 cache and 16 KB shared memory. Shared memory can be seen as a user-managed cache that can be accessed directly by the application. Because it is on chip, access to shared memory is orders of magnitude faster than access to global (or local) memory and as it is shared amongst all threads of the same thread block, together with barrier synchronization it can be used to achiever inter-thread communication. To speed up the accesses to

[5] Up to compute capability 1.2 memory requests have been per half-warp.

shared memory, it is organized in 32 *banks* so that consecutive 4-byte words belong to consecutive banks. If all threads of a warp request memory addresses that belong to different banks, all memory transactions can be done in parallel. If all threads access the same bank, a broadcast mechanism can support all threads with the resulting value with just a single memory transaction. In the other cases a *bank conflict* occurs and the memory access has to be serialized into a number of conflict free memory requests.

*Texture- and Constant Memory:* Apart from those memory regions, the device has a designated constant and texture memory, both cached through their own caches. Caching of texture memory is optimized for 2d spatial coherence and can enable texture filtering, clamping, etc. We will not use these memory regions and will not describe them in further detail. The interested reader may refer to the Nvidia CUDA Programming Guide.

*Global Memory Operations:* Having this said, we want to spend this paragraph to focus on the global memory operations that can be performed by the graphics cards memory system. Regardless of the actual code a programmer writes, all accesses to global memory have to be broken down to these basic memory operations. Although not necessary for correctness, for best performance it is strongly recommended to write code that respects the limitations of the hardware and best fits the implied memory transaction patterns and sizes. On the device there are two types of loads: caching and non-caching. When compiling in caching mode (which is set by default), a load instruction first tries to hit L1 cache, then L2 and finally global memory. The *load granularity* is one 128-byte cache line. In non caching mode[6], a load instruction tries to hit L2, then global memory and load granularity is only 32-byte memory segments. A store instruction invalidates the associated line in L1 and writes back to L2 cache. All memory operations are always issued per warp (unlike per half-warp as with pre-Fermi architectures). When issuing one memory operation, all 32 threads of a warp provide a memory address they wish to read. According to the memory access mode, the hardware then determines which memory lines/segments are needed and requests these lines/segments from memory. One should note that a 128-byte cache line or 32-byte memory segment are always the elementary memory transaction sizes. Every memory transaction will always move either 128 byte or 32 byte over the memory but even if for example all threads in one warp will address the same byte in memory. Uncoalesced and misaligned memory accesses can force the hardware to read more data than needed which leads to a discrepancy between the hardware memory throughput and the effective throughput of data in the software. This is the reason why one should always strive for perfect coalescing.

If coalescing is however impossible or very hard to achieve, it can be beneficial to disable caching to reduce memory over-fetch. This strongly depends on the actual implementation and it is worthwhile to try both memory configurations to see which one works best in a certain context. One last word on caching: The GPUs L1 cache is only 16 KB / 48 KB in size. This is 128 / 384 cache lines that have to be shared amongst all threads that run on a SM. Theoretically, a single SM can run 48 warps in 8 blocks concurrently

[6] Can be achieved with the current `nvcc` version by compiling with the `-Xptxas-dlcm=cg` compiler flag.



totaling in 12,288 threads. Even if we do not reach this border, it should be clear that there is an enormous pressure on the cache compared to a CPU that has a similar cache size but only has to serve a single thread. The GPUs L1 cache quickly gets polluted and should not be thought of as a CPU cache. We can mostly profit from caching to reduce the effect of misalignment when the overall access pattern is coalesced or when nearly all threads that run on the same SM have a good temporal and spatial locality. Due to the 128 byte wide cache lines, cached memory accesses can drastically reduce the effective memory throughput when performing scattered memory accesses.

### Branching

Control flow instructions can impact the performance if they depend on the thread ID. There are two ways the GPU can handle if-statements: branching and branch predication and the compiler tries to choose the best alternative depending on the number of instructions in the different execution paths and the possibility that execution may diverge for a given if-statement. When processing a warp, all threads in the warp execute the same instruction. The instruction can be associated with a per-thread condition code, or *predicate*, that is set according to the condition. Instructions with a false predicate do not write results or read operands. This way it is possible to mask several threads in a warp as idle. When using branch predication, none of the branches of an if-statement gets skipped. Instead both branches are executed and the instructions are associated with an appropriate predicate.
The compiler may also choose to implement a branch with predicated conditional jumps. In this case additional instructions for control flow are necessary but code execution is only serialized in case the threads of one warp take different execution paths. It is sometimes preferable to manage if-clauses with branch predication for small branches with only a few instructions because there are no additional instructions needed for conditional jumps.
As a performance guideline, the programmer should try to avoid large branches that cause divergence between the threads in one warp.

## 2.5 CPU Implementation of BVHs

In the following sections, we will give a more detailed description of the different bounding volume hierarchies we have used in our implementations. All hierarchies use binary trees and we use a top-down construction scheme as proposed by Gottschalk *et al.* [10]. The hierarchies are represented as an array of bounding volume nodes. Each inner node contains two references to its children. The leaf nodes use these references to store the start and end index in an array of primitives (triangles). The triangles within this range refer to the geometry contained in the corresponding leaf node. We use different bounding volume types and traversal schemes to find a setup that is best suited for our application in sampling based motion planning. For the CPU versions of the code, we parallelize the main loop that iterates over the individual collision tests. We compare several optimized multi-core CPU implementations with many-core CUDA implementations.

### 2.5.1 On the Implementation of OBB - Trees

Simple Traversal

In our CPU implementation, we represent each oriented bounding box (OBB) as a 3x3 rotation matrix $R$ together with the box's center point $c$ and a vector $h$ holding the three half lengths. This results in $9+3+3=15$ floats per OBB. Together with the references to its children, every bounding volume node occupies 68 bytes of memory. We use two different traversal schemes. In the first scheme, we store the bounding volume nodes of our hierarchies in the same coordinate frame as the root node of the tree. When testing a pair $(a, b)$ of bounding volumes in the query phase, we perform a transformation $M$ on $a$ to bring both volumes into the same coordinate frame and perform an OBB intersection test on $(Ma, b)$. We note that $M$ is always the same transformation for the whole BVH traversal, namely the transformation that brings the coordinates from the coordinate frame attached to $A$ to the coordinates of $B$. As we always assume that $B$ is given in world coordinates, $M$ is the transformation that brings $A$ in world coordinates, in our case.

Parent-Relative Traversal

The second approach uses the same traversal scheme that was used in the PQP[7] library. Here, every OBB is stored in the coordinate frame that is given by the orientation matrix and center point of its parent node. We refer to these volumes as being *parent relative*. Formally, this means that if we have two OBBs

$$OBB_p = (R_p, c_p, h_p) \text{ and}$$
$$OBB_c = (R_c, c_c, h_c)$$

and if $OBB_p$ is the parent of $OBB_c$, we make $OBB_c$ parent relative by transforming it to

$$OBB_c^{'} = (R_p^{\top} R_c,\ R_p^{\top}(c_c - c_p),\ h_c). \tag{2.1}$$

[7] The Proximity Query Package – see http://gamma.cs.unc.edu/SSV/



We traverse the tree with a "descend largest" descend rule, which means that for every intersecting bounding volume pair $(a, b)$ we create two new bounding volume pairs for the further traversal. These are either the pairs $(left(a), b)$ and $(right(a), b)$ or the pairs $(a, left(b))$ and $(a, right(b))$, depending on whether $a$ has a bigger volume than $b$ or vice versa. If $M =: (R, c)$ was the transformation that took $a$ into the coordinate frame of $b$, we have to determine a new transformation matrix for the child nodes because they are not in the same coordinate frame. If we traverse into the children of $a$, we have to update $M$ to

$$\begin{aligned} M_l &= (R_l, c_l) & M_r &= (R_r, c_r) \\ R_l &= R \cdot R_{left(a)} & R_r &= R \cdot R_{right(a)} \\ c_l &= R \cdot c_{left(a)} + c & c_r &= R \cdot c_{right(a)} + c \end{aligned}$$

where $M_l$ / $M_r$ takes $left(a)$ / right(a) in the coordinate frame of $b$. And similarly, we have to alter $M$ into

$$\begin{aligned} M_l &= (R_l, c_l) & M_r &= (R_r, c_r) \\ R_l &= R_{left(b)}^{\top} \cdot R & R_r &= R_{left(b)}^{\top} \cdot R \\ c_l &= R_{left(b)}^{\top} \cdot (c - c_{left(b)}) & c_r &= R_{right(b)}^{\top} \cdot (c - c_{right(b)}) \end{aligned}$$

to take $a$ into the appropriate coordinate frames if we traverse into the children of $b$. This means that compared to the first traversal scheme, we have to use different transformation matrices for every OBB test and as these transformations become constructed during traversal, we have to store them on the traversal stack together with the bounding volume nodes to enable backtracking. Despite these difficulties, this approach has one main advantage over the former traversal scheme: In both cases, we repeatedly have to apply a transformation $M$ to bring both OBBs in a common coordinate frame and perform an intersection test. However, in the case of parent-relative traversal not only can we achieve that both OBBs are in a common coordinate frame, but the OBB $a$ is transformed to the coordinate frame that is defined by the orientation matrix and center point of $b$. This can be exploited by using a specialized OBB intersection test that relies on this relative orientation of the two OBBs. Because the axes of $b$ are now the coordinate axes, we can save arithmetic operations in the separating axis test when computing dot products and cross products with these axes.

### 2.5.2 On the Implementation of AABB - Trees

We represent an axis aligned bounding box (AABB) as its center point $c$ together with a half length vector $h$. Together with the two child references, an AABB bounding volume hierarchy node occupies $3 + 3 + 2 = 8$ floats or 32 bytes of memory. Again, we use an "descend largest" traversal scheme but we do not make our hierarchies parent relative. We observe that because the AABBs of one hierarchy all have the same orientation (aligned to the axes of the BVH's local coordinate frame), all paris $(a, b)$ of one bounding

volume of hierarchy $A$ and one bounding volume of hierarchy $B$ will always have the same relative orientation. Essentially, if hierarchy $A$ undergoes a rigid motion, all tests $(a, b)$ are in fact intersection tests between oriented bounding boxes. But when we perform a separating axis intersection test (cf. Sec. 2.3), the 15 directions used for projection are always the same and can be precomputed once in advance. This leads to a cheaper intersection test as compared to an OBB intersection test. The drawback of this approach is that AABB hierarchies tend to fit the underlying geometry less tightly as OBB hierarchies which can increase the total number of bounding volume tests that are necessary to decide one collision query.

Volume Minimal AABBs

To construct our AABB hierarchies, we use two different top-down construction schemes. Out first approach recursively constructs AABBs on a given triangle set, starting with all triangles of the whole object. We then attempt to split the triangle set in two, probing all three coordinate axes as splitting directions. For every direction, we use a sweep-line approach to find the split position that results in two child boxes with a minimal accumulated volume. Finally, we use the bounding volume pair with the smallest accumulated volume as child nodes. We stop the recursion at a maximum recursion depth or when a box contains fewer than a given number of triangles.

The Box-tree

Our second method starts with the objects smallest enclosing AABB. At each iteration of the process, we split the parent box $P$ into two child boxes $S$ and $T$ in the direction of its longest edge. $S$ and $T$ are subtended with the smallest enclosing AABBs of the triangles they intersect. The resulting boxes are used as the child volumes of $P$. If a volume descends too deep in the tree or if it becomes too small, a leaf node is created and it is not further refined. Strictly speaking, this is not a classical bounding volume hierarchy because one triangle can be contained in multiple volumes at the same level of the hierarchy. This is typically avoided when using bounding volume hierarchies and it can lead to repeated tests of the same triangle pairs during traversal. However, the approach has the advantage that no pair of bounding volumes at the same level can ever intersect which leads to a better spatial differentiation. One can see this a hybrid approach that is in between a classical bounding volume hierarchy and a hierarchical space partitioning scheme. A similar approach was also presented by Zachmann [59] and van den Bergen [53].



## 2.6 GPU Implementation of BVHs

### 2.6.1 On GPU traversal of OBB - Trees

When implementing a bounding volume hierarchy traversal with CUDA, there are three major problems to be addressed: memory access, code divergence and workload balance.

*Memory Access.* There are two problems with memory access as far as BVH traversal is concerned. First, memory transactions between the GPU and video memory can only happen in chunks of 128 or 32 bytes. And second, it can be problematic to use cached memory accesses together with a random memory access pattern. Modern Nvidia GPUs use a two level caching mechanism. If we represent a bounding volume hierarchy as an array of bounding volume nodes, it is not possible to determine the order in which these volumes will be accessed. Also, if we try to fit multiple bounding volumes into one cache line (e.g. we could attempt to store child nodes near their parents) it is likely that the cache line has already been evicted when we need the children, because L1 cache is a scarce resource. We therefore try to access main memory through L2 only and disable L1 caching. This leads to a load granularity of only 32-byte memory segments. OBBs are an ideal candidate bounding volume to fit into 32 byte memory chunks. We recall from our CPU implementation that we represent an OBB with 9 floats for the rotation matrix $R$, 3 floats for its center point $c$, 3 floats for its halflengths $h$ and 2 floats for its child references. This sums up to 17 floats or 68 bytes per OBB bounding volume node. If we could safe one float, our OBBs would perfectly fit into two 32 byte segments.
We achieve this by not storing the whole 3x3 matrix $R =: (r_x,\ r_y,\ r_z)$ per OBB but just the first two column vectors $r_x$ and $r_y$. This saves us 3 floats and as $R$ is known to be orthonormal, $r_z$ can be easily reconstructed by taking the cross product of the other two column vectors: $r_z := r_x \times r_y$. We use one additional float value to store the volume of an OBB and add one float for padding. Eventually, this means we represent each OBB with 16 floats which is exactly 64 bytes.

*Code divergence.* Every 32 threads of one warp always share the same instruction. However, it is possible that different threads take different execution paths at conditional instructions when the condition depends on the thread ID. Whenever this happens, execution has to be serialized so that the whole warp takes both execution paths, having part of the threads masked idle. It is therefore strongly recommended to avoid divergent branching as much as possible.

*Workload balance.* A topic that is closely related to code divergence is workload balance. In the parallelization process, we have to break down the problem into small parallel sub-problems. In our case, we solve separate collision tests for a high number of different configurations and we use one thread to solve one query. It is possible that different sub-problems take a different amount of time, causing the threads of one warp to run for a different amount of time. But the warp cannot stop its execution until all threads are

ready. This can possibly cause a large number of threads to stall. We can minimize this effect by trying to balance the workload amongst all threads of a common warp.

### 2.6.2 On the Organization of Stack Traversal

As we want to perform a high number of different proximity queries at a time (for the same two triangle sets but in different positions), we have different opportunities to distribute the work to the parallel threads. Possible choices are:

$$\left\{ \begin{array}{c} one \\ some \\ all \end{array} \right\} \text{thread(s) can perform} \left\{ \begin{array}{c} one \\ some \\ all \end{array} \right\} \text{collision test(s)}$$

The following cases are interesting for our application:

*A. One thread performs one test:* This is the easiest choice for parallelization. Every test is performed in a serial manner and parallelization is performed solely over the different tests. There has to be no communication between different threads. If a traversal can be early interrupted, only the corresponding thread has to be stopped. One possible modification could also be not to give every thread its own private stack, but to let a number of threads share one common stack. This is called packet traversal in [42].

*B. One thread performs some tests:* This is very similar to the previous approach. One possible improvement is better workload balance. When every thread is performing a high number of tests, all threads are more likely to be busy for more or less the same amount of time.

*C. Some threads perform one test:* Pros: If a group of threads that is working on one test has a size that is a multiple of one warp, the whole group can stop working when the corresponding traversal has finished. The problem of workload balancing practically does not exist. It is also possible to achieve a good access pattern for the traversal stack. Cons: Inter-thread communication is needed between the threads of one group. For an almost empty stack, possibly not all threads of the group can be occupied. If one thread determines a termination condition, the work of the other threads for that cycle is wasted. For the latter two reasons, the group size should not be chosen too big.

*D. All threads perform all tests:* Pros: Very good workload balancing can be achieved because all bounding volume tests are evenly distributed over all threads. Every thread uses the same pool of bounding volume tests. There is no inter-thread communication necessary and good access pattern for the global stack can be achieved. Cons: It is difficult to interrupt the traversal of a specific test. Possibly all stack entries for one specific test have to be touched even though termination has already been determined. Also, every stack entry has to explicitly store the information to what test a bounding volume pair belongs and transformation matrices have to be loaded for every test.



*E. Some threads perform some tests:* This pattern enables good workload balancing for the same reason as D. However, as one group of threads is responsible for only a small number of tests, it has the advantage that the corresponding transformation matrices can be loaded once in advance and reused for the whole traversal. On a higher level, Pattern C can be seen as a special case of this traversal scheme with the additional advantage that as only one transformation is used per thread group, this information is implicitly known and has not to be stored explicitly for every pair of bounding volumes.

For our tests, we have implemented the traversal schemes A, B, C and E and used it together with an OBB bounding volume hierarchy for basic boolean collision detection. We are not going to further investigate pattern D, as it has no outstanding advantage over the other traversal schemes in our application. The scheme is better suited for more complex proximity queries, like distance tests, that do not follow an early exit traversal strategy and have to perform a comparably high number of bounding volume tests per proximity test. A similar pattern was proposed by Lauterbach *et al.*, for example [28].

In order to reduce the resource consumption of the CUDA kernels and to avoid code divergence, we first want to discuss a modified collision query that does not perform collision tests for the primitives contained in the bounding volume leaf nodes. In this context, we classify the proximity status of two objects solely based on bounding volume tests. For a collision test this means that if there exist two leaf nodes that are intersecting, we assume the objects to be colliding (and stop further BVH traversal). We note that this leads to a conservative collision test, i.e., we can possibly declare two objects to be colliding that are not, but we will never falsely declare two objects to be collision free.

### 2.6.3 A. One Thread Performs One Test

kernel A

In our first implementation, we want one thread to perform one collision test. The kernel gets two bounding volume hierarchies, $bvh_A$ and $bvh_B$ and an array of configurations Q (cf. Alg. 1). Every thread manages its own traversal stack in a dedicated piece of a large global memory array. In an initialization phase, it uses its thread-id to identify the right configuration and an offset to a private region of stack memory. The main loop is very CPU-like: In every iteration, we pop a stack entry before we load, transform and intersect the corresponding bounding volumes. All pairs of child volumes are pushed to the stack until we find a pair of intersecting leaf-nodes or the stack runs empty. Eventually, the thread reports its result to a dedicated place in a result array.
The implementation is very straight forward. It is understandable and can easily be maintained and debugged as there is no inter-thread communication. In fact, the only parts of the code that explicitly depend on the id of an individual thread are the initialization and the output of the result.

**Algorithm 1** KERNELA

```
void kernelA( Configs Q, BVH bvh_A, BVH bvh_B )
{
    qIdx = blockDim × blockIdx + threadIdx;
    S.start = globalStackArray + qIdx × STACK_SIZE; // ptr to private region
    q = Q[qIdx];                                    // of global stack

    S = { (0,0) }; // init stack with root nodes
    col = false;
    while( !S.empty() && !col )
    {
        (p_0, p_1) = S.pop();
        (A, B) = loadVolumes(p_0, p_1);

        A = transform(q, A);
        if( intersect(A,B) )
        {
            if( A.isLeaf() && B.isLeaf() )
                col = true;
            else
                S.push( children(A,B) );
        }
    }

    result[qIdx] = col;

}
```



### 2.6.4 B. One Thread Performs Some Tests

kernel B

Despite the advantages we have just mentioned, the implementation has one main drawback: it has no concept for workload balancing. The GPU schedules and executes groups of 32 threads (*warps*) in a SIMD fashion. This means that if the first 31 threads of a warp exit the main loop right away because their root bounding volumes have shown to be disjoint, they would still have to wait for the last thread to finish the main loop before they could write their result. Until this happens they become idle and waste GPU resources.
This effect is softened if all threads of the same warp consume more or less the same computation time (e.g. if the array of input configurations would be sorted by ascending traversal time). However, as we cannot expect that and as we cannot determine the traversal cost of our tests in advance, we use a different strategy: The basic idea is to let every thread perform more than just one collision test and to read configurations and write results from inside the main loop (cf. Alg. 2). This has the advantage that if one thread has finished one collision test, it does not have to wait for the other threads in the same warp but it can launch the next test. One possible way to implement that would be to have a shared pool of jobs that serves all our threads. However, this would require some sort of inter-thread communication to dynamically assign the jobs to the parallel threads. Thus, we use a static mapping of the collision tests to the threads where every thread performs the same number of collision tests. Now, if we assume that the number of main-loop iterations that have to be performed per collision test are uniformly distributed over the collision tests, the accumulated time that one thread needs to perform a high number of tests will roughly be the same. The drawback of this method is that we need more collision tests than in case of kernel A to have the GPU fully utilized. So it is not promising to let one thread perform a big number of tests.

### 2.6.5 C. Some Threads Perform One Test

kernel C

In our third implementation, we want a group of threads to work on the same collision test. There are several ways how this can theoretically be achieved: First of all, we have to decide how many threads should belong to a group that handles one collision test. The groups could all be of the same static size or of different sizes. We could even dynamically map all available threads to the individual collision tests which would lead to a pattern similar to pattern E described above. Second, every thread in a group could perform a pending bounding volume intersection test from the traversal stack or we could split a single bounding volume test over several threads.

As we deal with collision tests, the number of bounding volume intersection tests that have to be performed per collision test can be very volatile. It can vary from just a single intersection test to find out that the pair of root bounding volumes is disjoint to a high number of collision tests in case of two near but disjoint query objects. On average, we expect that we have to perform only a small number of bounding volume tests to

**Algorithm 2** KERNELB

```
void kernelB( Configs Q, BVH bvh_A, BVH bvh_B )
{
   qIdx = blockDim × blockIdx + threadIdx;
   nuOfThreads = blockDim × gridDim;
   S.start = globalStackArray + qIdx × STACK_SIZE; // ptr to private region
                                                   // of global stack
   q   = Q[qIdx];
   col = false;
   S = { (0,0) }; // init stack with root nodes
   while( true )
   {
      if( S.empty() || col )
      {
         result[qIdx] = col;

         qIdx += nuOfThreads;
         if( qIdx ≥ Q.size() ) return;

         q   = Q[qIdx];
         col = false;
         S   = { (0,0) };
      }

      (p0,p1) = pop(S);
      (A,B) = loadVolumes(p0,p1);

      A = transform(q, A);
      if( intersect(A,B) )
      {
         if( A.isLeaf() && B.isLeaf() )
            col = true;
         else
            S.push( children(A,B) );
      }

   } // end of while

}
```



localize a pair of intersecting bounding volume leaf nodes or to see that the hierarchies are disjoint. In the beginning and towards the end of the traversal, the traversal stack contains only a moderate number of bounding volume pairs. If we let every tread of the same group perform its own bounding volume intersection test from a shared traversal stack, a certain fraction of the threads cannot contribute to the iteration of the main loop whenever the stack size becomes lower than the group size.
This is why we want to use the second approach and split a single bounding volume test to a group of threads. A natural group size for current Nvidia GPUs would be a multiple of one warp (i.e. 32 threads). In the context of collision tests, this is already a quite large group size. Recall that we want to use OBB hierarchies and that we use a separating axis test to decide if two oriented bounding boxes are disjoint (cf. Sec 2.3.2). The separating axis test is an ideal candidate for parallelization, as we can independently test all 15 candidate axes.

These considerations led to the following choices in our implementation (cf. Alg. 3): We use groups that are composed of a fixed number of 16 threads. There is a dedicated master thread that manages the traversal stack and pops a new stack entry at every iteration of the main loop. As we represent an OBB node with 15 floating point values, we use the first 15 threads of the group to load the two corresponding bounding volumes A and B from memory with coalesced reads of 15 floats. The master thread transforms A according to the configuration that belongs to the collision test. After that, the first 15 threads perform a projection of A and B on one of the 15 different separating axes. The threads share the result of the OBB intersection test through a common shared memory variable.

The reader may have noticed that we only use the first 15 threads of every group. The 16th thread is always idle! This means that we waste 1/16th or about 6% of our computational resources, in the first place. However, if we mask out this one thread per group, we can achieve that the threads of every two groups sum up nicely to one warp. If we decided to enforce a group size of 15 instead, we would introduce a stride of 2 which results in a sub-optimal alignment of our logical units (groups) to the physical unit of one warp. In practice, we have observed that it is beneficial to sacrifice the 16th thread.

### 2.6.6 E. Some Threads Perform Some Tests

kernel E

In our implementation of pattern E, we use a group of 32 threads to perform 8 collision tests. Here, we use no parallelization of a single bounding volume intersection test. Every thread of the group pops a stack entry, loads the corresponding volumes, performs an intersection test and possibly pushes new entries on the stack. To compensate for the low thread utilization ratio in the beginning of the traversal, all threads in one group share the same traversal stack. This requires to store not only the indices of the pending bounding volume pairs on the stack but also to record an index for the corresponding

**Algorithm 3** KERNELC

```
void kernelC( Configs Q, BVH bvh_A, BVH bvh_B )
{
   groupSize = 16;
   groupsPerBlock = blockDim / groupSize;

   tIdx = threadIdx % groupSize;
   gIdx = threadIdx / groupSize;

   qIdx = groupsPerBlock × blockIdx + gIdx;
   S.start = globalStackArray + qIdx × STACK_SIZE; // ptr to private region
   q = Q[qIdx];                                    // of global stack


   __shared__ volatile int col;
   __shared__ volatile OBB A, B;

   S = { (0,0) }; // init stack with root nodes
   col = false;
   while( true )
   {
      if( tIdx == 0 )
      {
         if( S.empty() )
            col = true;
         else
            (idx_A, idx_B) = S.pop();
      }

      if( tIdx < 15 )
         (A,B) = loadVolumes(idx_A, idx_B);

      if( tIdx == 0 ) A = transform(q, A);


      __shared__ cut = 1;
      if( tIdx < 15 )
      {
         n = getSepAxis( A, B, tIdx );
         if( sepAxisProject( A, B, n ) ) shCut = 0;
      }
      if( tIdx == 0 && shCut )
      {
         if( A.isLeaf() && B.isLeaf() )
            col = true;
         else
            S.push( children(A,B) );
      }
   } // end of while
}
```



transformation.[8] The pseudocode for this processing scheme is given in Alg. 4. First, we initialize the global stack and load eight transformations to a shared memory location to avoid a reload of the transformations for every bounding volume intersection test. In the main loop, we utilize as many threads as we have stack entries left, but at most 32 threads. Every thread loads and transforms a bounding volume pair and performs an intersection test. Depending on the result of the collision test some but possibly not all threads need to write new pairs of child volumes on the stack. To improve the access pattern for the global memory writes to the stack, we first write all produced stack entries densely to a shared memory array and then copy the whole array to the global memory stack.

We have tested two different implementations for the parallel_write to the shared memory array. The first implementation uses a static addressing scheme to assign the entries of the output array uniquely to the threads of the group. If not all threads perform a write operation, the resulting array is sparse and has to be compacted before we copy it to the global memory stack. The second implementation uses atomic operations to increment a pointer to the shared array and to perform a dense output in the first place. We found that the second option was preferable in our test cases.

### 2.6.7 Coalesced Scattered Reads

When implementing a bounding volume hierarchy traversal with CUDA, we have different choices for management of the traversal stack, data organization of the hierarchy, etc. One common idea in GPU programming is trying to keep the working data in shared memory as far as possible to circumvent the high latencies of global memory access. However, shared memory is a very limited resource. For example, the Fermi family of Nvidia GPUs offers at most 48 KB of shared memory that has to serve all threads that run on the same SM. This means that holding the BVH traversal stack in shared memory, which is certainly a good idea as far as memory access times are concerned, highly influences the number of threads that can run concurrently on every SM. If we represent a stack entry by two integer values and if we assume a maximum stack size of 32 this means that 192 threads can run simultaneously on one SM. This is 6 of potentially 48 warps that can be handled by the tread scheduler.

In contrast to this idea, we want to achieve good performance in the context of BVH traversal by using global memory accesses, but taking special care to use coalesced memory accesses whenever possible. We want to relativize the opinion that global memory accesses are very expensive and should be avoided by any means. We want to design a traversal algorithm that holds both, the hierarchies and the traversal stack, in global memory and rather use shared memory to implement a good memory access pattern. Our basic consideration is as follows: The peak memory bandwidth of a GTX 480 Fermi GPU is $> 170GB/s$. We can represent an OBB volume with 64 bytes. At every step of

[8] In order to save memory and not to use a whole 32 bit integer for the transformation index, we encode the index in the high level bits of the first bounding volume index.

**Algorithm 4** KERNELE

```
void kernelE( Configs Q, Bvh bvh_A, textscBvh bvh_B}
{
   groupSize = 32;
   groupsPerBlock = blockDim / groupSize;

   tIdx = threadIdx % groupSize;
   gIdx = threadIdx / groupSize;

   qIdx = groupsPerBlock × blockIdx + gIdx;
   S.start = globalStackArray + qIdx × STACK_SIZE;

   __shared__ Configs q[8];
   if( tIdx < 8 )
      q[tIdx] = Q[qIdx];

   // init stack with root indices for all 8 configs
   S = { (0,0,0), (1,0,0), ..., (7,0,0) };
   while( !S.empty() )
   {
      if( tIdx < S.size() )
      {
         (t, p0, p1) = S.pop( tIdx );
         (A, B) = loadVolumes(p0, p1);

         A = transform( q[t], A );
         if( intersect( A, B ) )
         {
            if( A.isLeaf() && B.isLeaf() )
               col[t] = true;
            else
               parallel_write( sharedArray,
                               {t} × children(A,B) );
         }
      }

      copy( S, sharedArray );
   } // end of while

   if( tIdx < 8 ) result[qIdx] = col[tIdx];
}
```



a BVH traversal a stack entry (8 byte) and two bounding volumes have to be read from memory, a bounding volume intersection test has to be performed and possibly two new stack entries have to be written. So overall $8 + 2 \times 64 + 2 \times 8 = 152$ bytes have to be moved over the bus. If the BVH traversal kernel was memory bound and if we achieved near peak memory throughput, it would be possible to perform more than one billion bounding volume tests per second. And if a BVH-based collision test required a few hundred OBB intersection tests in average, we still could perform millions of rigid body collision tests per second on the GPU.

When we are aiming to achieve peak memory throughput, memory accesses have to be made according to certain rules (cf. Sec. 2.4 or the CUDA Programming Guide [36]). For example, we should always try to arrange memory access in a way that consecutive threads are accessing consecutive memory addresses. In this way, it becomes possible for the hardware to coalesce several memory operations to a single transaction. This is not always possible when loading the bounding volumes during BVH traversal. Consecutive threads can perform a completely different traversal and need to load different bounding volumes at different times. This leads to scattered reads. There are at least two ways to circumvent this problem:

(a) Arrange the different collision tests in a way that all threads in one warp perform the exact same traversal (referred to as "packet traversal" in [42], for example).

(b) Use a different access pattern: We can use all threads of the same warp to load the same bounding volume together.

When we can cluster the threads like proposed in (a), a number of threads can share the same traversal stack. All operations on the stack and loading the bounding volumes have to be performed only once per thread group (e.g. one warp). In principle this could lead to a data throughput *per thread* that is higher than the theoretical throughput of the device. However, we can only expect this approach to work well if the traversals that are performed by the threads of one warp are sufficiently similar.
We want to focus on the second approach, instead: To load the 16 consecutive float values of every OBB volume, we use 16 consecutive threads of the same warp. The bounding volumes for the whole warp are then loaded one after another instead of simultaneously. Of course, this implies that the bounding volumes have to be loaded into shared memory. However, all global memory accesses will always be coalesced. This scheme allows to perform the scattered reads of different OBBs during BVH traversal with coalesced memory accesses. The volumes are stored in shared memory and can be used by the individual threads later on. We call this the *Coalesced Scattered Reads* (CSR) access pattern. It can easily be applied to conventional BVHs and we do not have to perform any preparatory work on the input configurations.

Test setup

We have analyzed the proposed access pattern in the following test scenario: We use a test kernel, that operates on an array A with $M$ 64-byte structs (i.e. 16 float values per struct). This could be an array of OBB bounding volume nodes, for example. The kernel loads a struct, computes the sum of the 16 floats and stores the result in an output array.

| cache configuration | N | mem access | naive reads | CSR |
|---|---|---|---|---|
| L2 cache only | $2^{17}$ | sequential | 46.20 | 74.89 |
| | | scattered | 21.72 | 65.74 |
| | $2^{15}$ | sequential | 48.58 | 84.98 |
| | | scattered | 23.49 | 72.49 |
| | $2^{7}$ | sequential | 48.21 | 87.14 |
| | | scattered | 23.86 | 87.12 |
| with L1 cache | $2^{17}$ | sequential | 68.62 | 80.09 |
| | | scattered | 32.82 | 54.27 |
| | $2^{15}$ | sequential | 121.83 | 85.81 |
| | | scattered | 34.34 | 63.08 |
| | $2^{7}$ | sequential | 133.57 | 101.48 |
| | | scattered | 80.02 | 101.75 |

Table 2.1: Memory throughput for different configurations on a Nvidia Quadro5000. Naive reads vs. Coalesced Scattered Reads. All numbers are in GiB/s.

To access this array of structs, we use an index-array perm of length $N$ that contains indices $i \in [0, \ldots, M-1]$. We are then reading the struct at position perm(i) and writing the result at position $i$ in the output array. In our CSR pattern, the threads of a warp load the structs at perm(i), perm(i+1) to perm(i+31) together to a shared memory location to perform their own calculations afterwards. As a reference, we use a kernel that performs the same operations but uses a naive access pattern, where every thread is reading its own structs into local variables. The results of our experiment can be found in tables 2.1 and 2.2 for a Quadro 5000 and GTX 480 graphics card, respectively.

Test parameters

All together, we read, accumulate and store $N = 2^{20}$ structs. We set the number of structs in our array to be either $M = 2^{17}$, $2^{15}$ or $2^{7}$. For sequential memory accesses, we initialize the perm array with sequential numbers: perm[i] := i % M. We apply a random permutation in order to simulate scattered reads. Both graphics cards are equipped with a small 16KB L1-cache per SM and a 768 KB L2-cache shared by all SMs. The L1-cache can be deactivated during compilation but all accesses to the global memory always go through L2-cache (cf. Sec. 2.4). The upper half of the tables shows the behavior of our test kernels with L1-cache deactivated, the lower half with an active L1-cache configuration.

First, we can see that the GTX 480 graphics card performs better than the Quadro 5000. This is because of a wider memory bus and higher memory clock. The GTX 480 has a theoretical peak memory throughput of 177.4 GB/s whereas the Quadro 5000 is theoretically bound to 120 GB/s. We should keep in mind that these are theoretical bounds and we cannot expect any but the simplest real world applications to actually reach these values.

Discussion

When L1-cache is turned off, we can see that the memory throughput for the naive



| cache configuration | M | mem access | naive reads | CSR |
|---|---|---|---|---|
| L2 cache only | $2^{17}$ | sequential | 81.99 | 123.78 |
| | | scattered | 37.43 | 103.18 |
| | $2^{15}$ | sequential | 87.81 | 153.20 |
| | | scattered | 41.60 | 124.24 |
| | $2^{7}$ | sequential | 89.17 | 161.01 |
| | | scattered | 44.32 | 161.02 |
| with L1 cache | $2^{17}$ | sequential | 119.71 | 134.15 |
| | | scattered | 57.72 | 81.79 |
| | $2^{15}$ | sequential | 205.70 | 154.20 |
| | | scattered | 62.47 | 106.17 |
| | $2^{7}$ | sequential | 245.36 | 187.18 |
| | | scattered | 148.82 | 187.50 |

Table 2.2: Memory throughput for different configurations on a Nvidia GTX480. Naive reads vs. Coalesced Scattered Reads. All numbers are in GiB/s.

access pattern is highly influenced by scattered accesses and drops to 40–50% compared to sequential access. Reducing the number of structs to 128 so that the whole array fits into L2-cache can not improve the throughput by much. The CSR access pattern results in higher throughputs in our example and is less affected by reading from scattered memory locations. It can also benefit from the locality introduced by reducing the number of structs. If we turn on the cache, things are different. The naive reads can greatly benefit from accesses through L1-cache for a sequential access pattern. It is even possible to achieve a memory throughput that is above the theoretical bandwidth of the device if it is possible to reuse the same data for several memory requests. This becomes more and more likely as we shrink the working set (i.e. for lower values of M). However, random memory accesses have a very negative effect to the memory throughput and we can achieve 30–60% better rates if we use CSR access. As far as the impact of L1-cache for CSR is concerned, we can see that caching can improve performance for sequential access patterns or small working sets. In the case of random memory access, the CSR pattern cannot benefit from the cache.

Conclusion

To summarize our observations, when reading large structs from global memory we see that caching is best used in combination with sequential memory access or very small working sets. In this case it is also recommended to use a naive access pattern. For large working sets or when reading the structs in a random pattern, it is better to use coalesced scattered reads and turn off the L1-cache. In practice, the structs that we are reading are bounding volumes of two hierarchies. Each hierarchy can easily have hundreds of thousands of bounding volumes and the pattern in which a single thread accesses the structs is not predictable in advance. We can therefore expect the Coalesced Scattered Reads to be a good choice for bounding volume hierarchy traversal. However, the implementation of the scheme consumes a non trivial amount of shared memory

which limits the total number of threads that can be executed concurrently per SM. Again, as we represent an OBB node with 15 floats and every thread needs two OBBs, we need

$$2\ {}^{OBBs}/_{thread} \cdot 15\ {}^{floats}/_{OBB} \cdot 32\ {}^{threads}/_{warp} \cdot 4\ {}^{byte}/_{float} = 3,840\ {}^{byte}/_{warp}$$

of shared memory to implement the CSR pattern. With 48KB of shared memory per SM, this means we can run at most 12 warps concurrently on every SM. This corresponds to a maximal achievable occupancy of 25%. We are going to analyze how the different access patterns can be used in practice in Sec. 2.9.5.



## 2.7 Grids

When testing two triangle sets for intersection, we have to identify at least one intersecting triangle pair or we have to prove that no such pair exists. It is inappropriate to simply test all triangle pairs as this would result in an exorbitant number of triangle-triangle intersection tests. The basic idea behind many advanced collision detection techniques is to (hierarchically) *approximate* the triangle sets and to *localize* the relevant regions. In section 2.2 we have seen how this can be achieved with bounding volume hierarchies. Bounding volume hierarchies partition a triangle set by hierarchically assigning the triangles to different bounding volumes.
Another choice to narrow down the regions of possible object interpenetration is to use space partitioning schemes like grids, octrees, k-d trees or BSP-trees. The main difference to the bounding volume approach is that these data structures are intended to partition the world space into several regions. The triangles of both objects are inserted into the regions they occupy, which can cause one triangle to be represented by more than one region. In the query phase only the triangles of adjacent regions have to be tested for intersection.

Apart from bounding volume hierarchies, we have also tried to use grids to accelerate collision queries between two triangle sets. An excellent survey on how grids can be utilized for collision detection is given in the book Real-time Collision Detection of C. Ericson [7]. The common approaches mostly differ in the data representation and how the objects are assigned to the grid cells. A uniform grid could be represented as an array of lists, a hash table, et cetera. Objects that overlap more than one grid cell can either be inserted in all occupied cells or in just one cell, according to a single reference point. These choices have implications on the neighborhood of a cell that has to be considered in the query phase (see [7] for a detailed discussion). The grids cell size should be chosen according to the average triangle size for best efficiency. If the cells are too big, they cannot clearly separate different triangles, if the cells are too small, the number of cells that have to be tested in the query phase is possibly very high. It is not always possible to choose one appropriate cell size if the objects that are inserted into the grid are of varying size. In this case, one could use a hierarchical grid (or multigrid) that consists of several layers with different cell sizes. Hierarchical grids in this respect are very similar to octrees and the borders between hierarchical grids, octrees and the box-tree we have presented in Sec. 2.5.2 are fluent.

### 2.7.1 Uniform Grids and Voxelization

We use the term of a (uniform) *grid* $\mathcal{G}$ to describe a partition of space into disjoint axis-aligned cuboids or *grid cells* of equal size. The extent of a cell can be described by a vector $b \in \mathbb{R}^3$ holding its edge lengths in $x, y, z$ - direction. As we typically do not need to cover the whole space, we limit a grid to an axis aligned box $B$. For simplicity, we

always want to assume, that the edge lengths of $B$ are integer multiples of the cells edge lengths $b$. This means, we can represent a grid as a tuple of two axis-aligned boxes:

$$\mathcal{G} := (B, b),$$

where $B$ is the domain of the grid and $b$ represents the dimensions of a grid cell. For every box, we always use the bottom left corner (the point with minimal x, y and z coordinate) as a reference point and we denote the reference point of $B$ with $o$, the *origin* of the grid. To identify a grid cell we can use its reference point. And as per construction all reference points are elements of $\{o + \mathbb{N}^3 \cdot b\}$, we can simply identify a grid cell by a unique vector of integers $c \in \mathbb{N}^3$.

The map that assigns a point $p \in B$ to the integer-coordinates of the cell it occupies is given as

$$\begin{aligned} f\colon B &\to \mathbb{N}^3 \\ p &\mapsto \lfloor (p - o)/b \rfloor, \end{aligned}$$

where the division is performed coordinate-wise. Likewise, we can easily assign a cell's integer coordinates to its reference point by applying

$$\begin{aligned} g\colon \mathbb{N}^3 &\to B \\ (c_x,\ c_y,\ c_z) &\mapsto (c_x b_x,\ c_y b_y,\ c_z b_z) + o. \end{aligned}$$

When we want to represent a triangle set with a grid, we determine the set of all grid cells $c$, that are occupied by at least one triangle. For every cell, we also store identifiers of the intersecting triangles as meta-information. We denominate the representation of a triangle set as a set of boxes a *voxelization* $\mathcal{V}$ of the object and we use the term *voxel* synonymously for grid cell in this context.
To make the difference clear: a grid denominates the spatial data structure that is used to subdivide a certain domain $B$ into evenly sized grid cells. We can insert a triangle into the grid by storing an identifier of the triangle for every intersecting grid cell. If we insert the triangles of a triangle set, the non-empty cells of the grid serve as our object representation, we call this the *grid representation* of the object. If we want to forget about the underlying grid structure and just talk about the set of boxes enclosing the object, we call this a *voxelization* of the object. However, in our application a voxelization is always associated with a grid and the terms grid representation and voxelization are closely related. Because a voxelization is just a set of boxes (voxels), it is easier to represent than the whole underlying grid. As all voxels are evenly sized, it is sufficient to store just one reference point per voxel. Opposed to grid cells, we typically use the center point of a voxel as a reference point. In this regard, a voxelization is closely related to a *point cloud* representing the triangle set.

We have just seen that the terms grid representation and voxelization are closely related and there is no big difference between the two structures. The main difference lies in



the representation of the two data structures in memory. Whereas a grid representation is implemented as a grid on a certain domain $B$ together with the two functions $f$ and $g$ and the meta-data of the occupied cells, we represent a voxelization as a set of points (the voxels' centers) together with a vector $b \in \mathbb{R}^3$, holding the extent of a voxel. Having this in mind, if we represent an object $A$ with a grid $\mathcal{G}_A$. This grid representation yields an object voxelization $\mathcal{V}_A$. Finally, $\mathcal{V}_A$ yields a point set $\mathcal{P}_A$. The reason for the different memory representation is the support of different operations on these data structures: our voxel representation allows easy iteration over all voxels, but it is a non-trivial operation to decide if a given point $p \in \mathbb{R}^3$ belongs to the voxelization. With a grid representation, this operation can be performed in constant time.

### 2.7.2 The Voxmap-Pointshell Algorithm

The original voxmap-pointshell algorithm was proposed by McNeely, Puterbaugh and Troy [31] and later improved by Renz, Preusche und Hirzinger [45]. For a collision query between two objects, the authors consider one object to be static and the other object can be placed dynamically. The static object is represented with a so called voxmap[9], the dynamic object with a set of points that are distributed on its surface: a pointshell. For every collision test, the pointshell is transformed into the local coordinate frame of the voxmap and a lookup is performed for every transformed point in order to decide if it hits a voxel. In the best case, only a few points have to be tested to detect a collision. However, in case of no collision, all points have to be tested and so the worst case running time of the algorithm grows linearly with the number of points.

The basic algorithm solely relies on the representation as a voxmap / pointshell which induces an approximation error and can change the objects topology. The sampling of the pointshell has to be carefully chosen in order to reflect sharp features of the objects. Main advantages of the approach are its simplicity and robustness. A single point lookup is a very cheap operation that has to be performed many times. This makes the algorithm a good candidate for parallelization on current SIMD architectures.

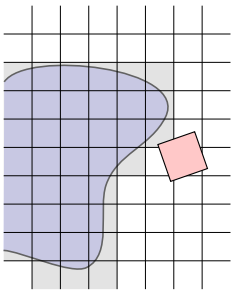

Fig. 2.5: We have to check if a voxel intersects an occupied grid cell.

### 2.7.3 The Particles Algorithm

Our approach is similar to the voxmap-pointshell algorithm and other grid-based approaches (see [35] Chap. 29 or 32 for example). But we tried to address the problems mentioned in the last paragraph and always enable the algorithm to perform collision queries that are accurate on the underlying triangle sets. For approximate collision tests, we want the algorithm to be conservative in a way that it will never miss a collision but will at most give false positive results.

When performing proximity queries, we can first generate a grid representation $\mathcal{G}_B$ of the static object $B$ and represent the dynamic object $A$ with its voxelization $\mathcal{V}_A$. We then

[9] The term reflects the fact, that a vox-map can be seen as a 3d bitmap whose elements are voxels.

have to decide for every voxel $v \in \mathcal{V}_A$ if it intersects an occupied grid cell $c \in \mathcal{G}_B$ (see Fig. 2.5). Whenever $v$ intersects a grid cell $c$, we can simply test all triangles that are recorded in $v$ and $c$ for intersection. As the objects' voxelizaion and grid representation are conservative, we won't miss any collisions due to sampling based effects. If we decided to drop the triangle tests and define $A$ and $B$ to be intersecting solely on the voxel information, this would yield in an approximate, yet conservative, collision test.
We want to preserve this behavior while trying to change the elementary voxel – grid-cell test into a simple lookup. To do so, we change the representation of $A$ from its voxelization $\mathcal{V}_A$ to the corresponding point set $\mathcal{P}_A$ by shrinking every voxel to its center point. We account for this simplification in the grid representation $\mathcal{G}_B$ of $B$ by performing an offset operation on the grid: We do now insert a triangle $t \in B$ not only in its intersecting grid-cells but in all cells $c$ with

$$| c - t | \leq \delta,$$

where $\delta$ if half the diameter of a voxel $v \in \mathcal{V}_A$. As no point inside of a voxel has a distance greater than $\delta$ from its center point, we can now safely test the center point of a voxel against the "offset" grid-representation of $B$ (Fig. 2.6). If we choose the cell size of $\mathcal{G}_B$ to be at least $\delta$, we can approximate the offset by simply considering the 27 neighbors of every occupied grid cell as being "$\delta$-close" to the contained triangles. This can simplify the implementation (compared to computing an exact offset) and speed up the construction process of the grid (Fig. 2.7).
Following this logic, it is also possible to offset the grid by a value that is greater than a voxel diameter. This way, the approach could also be used to perform a tolerance test instead of a collision test.

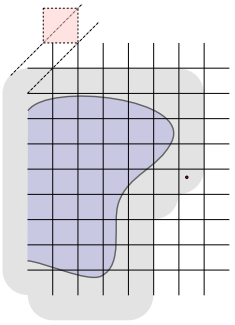

Fig. 2.6: If we shrink the voxel to a point and offset the grid by the half the diameter of a voxel, we still have a conservative test.

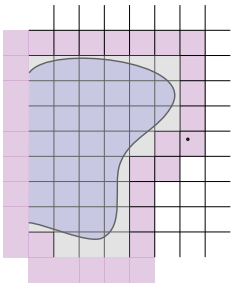

Fig. 2.7: We conservatively approximate the exact offset with one additional layer of grid cells.

### 2.7.4 The Hierarchical Particles Algorithm

Like the voxmap-pointshell algorithm, the resulting approach is very easy to implement and a single collision query can be trivially parallelized over the, performed grid lookups. If the queried objects intersect, possibly only a few points $p \in \mathcal{P}_A$ have to be tested to identify one intersecting triangle pair. However, as it lacks a hierarchical representation, in the non-colliding case all points of $\mathcal{P}_A$ have to perform a lookup in $\mathcal{G}_B$ which leads to a worst case time complexity of $O(|\mathcal{P}_A|)$.

To take this problem into account, we want to propose a hierarchical multi-grid approach. We repeatedly coarsen the object voxelization and grid-representation which results in several layers of conservative object representations with decreasing resolution. Starting at the top level layer, we test the center points of a coarse voxelization against the corresponding grid. In case the queried voxel does not collide with any grid-cell, we can safely skip all the voxels it contains at the next hierarchy level. Like in the previously described approach, we just perform grid-lookups as an elementary operation as compared to the box-box-intersection tests we had to perform with a bounding volume hierarchy approach. Moreover, we do not need to keep track of the conflicting (voxel,



grid-cell)-pairs when descending to the next level of the hierarchy, nor do we have to ever explicitly determine *all* such pairs. Because the assigned grid structures present a global view for every level of the hierarchy we only have to keep track of the conflicting voxels. Whenever a voxel is in conflict with just one grid-cell, we have to examine its child voxels. We do not have to determine if the voxel is in conflict with other grid-cells, too. In other words, with this approach, we do not have to traverse two hierarchies in tandem, like with bounding volume hierarchies, but just the hierarchical voxelization of *one* object. The implementation of the algorithms and the required data structures are discussed in more detail in the next section.

## 2.8 CPU Implementation of Grids

**Grid.** Our implementation of a grid is pretty straightforward. We use a grid only to subdivide space into evenly sized cells. Thus, it consists of a bounding box $B$, the domain of the grid, and a vector $b$ representing the dimensions of the cells. Moreover, the grid structure implements the two maps $f\colon B \to \mathbb{N}^3$ and $g\colon \mathbb{N}^3 \to B$ stated earlier and a map to convert integer-coordinates into a single hash-value:

$$\begin{aligned} hash\colon \mathbb{N}^3 &\to \mathbb{N} \\ (x, y, z) &\mapsto x + res_x \cdot (y + res_y \cdot z), \end{aligned}$$

where $res_x$ and $res_y$ denote the number of grid-cells in $x, y$-direction respectively.

**Voxelization.** To get a conservative object voxelization, we start out with a pair $(B, T)$, where $B$ is an axis aligned box that bounds the whole triangle set $T$ representing our object. Obviously, this box contains all triangles. We then divide $B$ into eight evenly sized sub-boxes $B_1, \cdots, B_8$ and determine the eight subsets $T_1, \cdots, T_8 \subset T$ of the triangles that intersect the sub-boxes. The resulting pairs $(B_i, T_i)$ are recursively processed in the exact same way. The recursion stops, when the boxes are sufficiently small. We end up with a sequence of boxes (the voxels) together with a sequence of triangle sets that correspond to the triangles contained in every specific box. As we have stated earlier, we do not have to store a whole box to represent a voxel, as all voxels have the exact same size. Instead, we represent the boxes as a sequence of points $\mathcal{P}$ together with the grid $\mathcal{G} = (B, b)$ that is implied by the bounding box $B$ underlying the voxelization and the vector $b$ representing a voxels dimensions. See Algo. 5 for the pseudocode.

**Grid Representation.** In order to represent an object as a gird, we first generate a voxelization as described in the previous subsection and insert the voxel center points into the implied grid. Our in-memory representation of this voxel grid is inspired be the implementation of the Nvidia SDK Particles Demo.[10] It consists of two arrays of integers: `start` and `end`. Both arrays hold as many entries as there are cells in the gird. The entries reference into an array `data` of triangle ids. After the grid is properly

[10] http://docs.nvidia.com/cuda/cuda-samples/index.html

**Algorithm 5** VOXELIZE

```
void VOXELIZE( Voxelization V, Box B,
    Triangleset T, double vox_size )
{
    (B_1,···,B_8) = childBoxes( B );
    for( int i=0;i<8;i++ )
    {
        T_i = ∅;
        for( t ∈ T )
        {
            if( intersect(t, B_i) )
                T_i = T_i ∪ t;
        }

        if( diameter(B_i) < vox_size )
            V.add( B_i, T_i );
        else
            VOXELIZE( B_i, T_i );
    }
}
```

initialized, it holds that every cell $(x, y, z)$ with hash value $h = hash(x, y, z)$ contains the triangles with the triangle ids stored in the sequence

```
data[ start[h] ], ..., data[ end[h] - 1 ].
```

Which means that `start[h]` denotes the start of a sequence of triangle ids stored in the `data` array that correspond to the triangles stored in the grid cell with hash value `h`. `end[h]` points right behind the end of this sequence.

**Coarsening.** With the data structures of a voxelization and a grid representation, we can already implement a simple collision detection algorithm. The algorithm can easily be executed on a data parallel device like the GPU as it performs basically the same operation for every voxel of a voxelization. The procedure scales naturally with the number of present GPU cores. Its main disadvantage is the lack of a hierarchical structure. In case the queried objects do not intersect, the algorithm has to touch all voxels to conclude that there was no collision.
To get a hierarchical object representation, we repeatedly *coarsen* the underlying voxelization. This results in a sequence of voxelizations where each voxel knows all voxels it contains of the next finer level. We represent a *coarsening* $\mathcal{C}_i$ at level $i$ with an array of voxel center points `v` together with two arrays `start` and `end` of the same size as `v`. When the coarsening sequence is properly initialized, the $k$-th voxel of the coarsening at level $i$, $\mathcal{C}_i$.`v[k]`, contains the voxels

$$\mathcal{C}_{i+1}.\texttt{v}[\texttt{start}], \cdots, \mathcal{C}_{i+1}.\texttt{v}[\texttt{end} - 1],$$



where

$$\texttt{start} := \mathcal{C}_i.\texttt{start}[\texttt{k}] \text{ and}$$
$$\texttt{end} := \mathcal{C}_i.\texttt{end}[\texttt{k}].$$

In addition to the center points, we also store the grid $\mathcal{G}_i$ that is underlying the voxelization for every level. We want to consider the underlying object voxelization $\mathcal{V}$ as bottom level of the hierarchy. To build a coarsening $\mathcal{C}_{i-1}$ of $\mathcal{C}_i$, we set the grid of $\mathcal{G}_{i-1}$ to a coarsened version of $\mathcal{G}_i := (B, b)$:

$$\mathcal{G}_{i-1} := (\tilde{B}, 2^l \cdot b).$$

For an uneven resolution, the grid domain $B$ possibly has to be enlarged to contain all enlarged voxels, hence it may be that $\tilde{B} \supset B$. All points $\mathcal{C}_i.v$ are inserted into the new grid $\mathcal{G}_{i-1}$ and we record the center points of the occupied cells for the pointset $\mathcal{C}_{i-1}.\mathtt{v}$. The reader should note that the order of the sequence of voxel center points has to respect this coarsening operation. This means that whenever a number of points in level $i$ are merged to a single point in the (i-1)-th level, these points have to be in consecutive order. We can achieve this by carefully considering the point order during the voxelization process. Or we can sort an existing voxelization according to the value of their Morton codes (also referred to as Z-order), afterwards. It turns out that this order naturally respects the coarsening operation.

**Voxmap.** For an efficient lookup of the coarsened voxels of the moving object, we need an explicit grid representation of the static object. Because we do not need to store the information, of the contained triangles for the upper hierarchy levels, we want to take a simpler representation for all but the bottom level of the voxel-hierarchy. We represent these levels explicitly with voxmaps, i.e. an array $\mathcal{VM}$ of bits. The array has as many entries as there are cells in the grid and a grid-cell with integer-cordinates $(x, y, z)$ corresponds to the bit $\mathcal{VM}[hash(x, y, z)]$. When converting the object representation from a voxelization to a voxmap, we take care to apply a proper offset as described in the paragraph about the grid representation. Otherwise, we had to check not only if a voxel midpoint falls into an occupied grid cell, but we also had to check a certain neighborhood. By offsetting the grid-representation, we deal with the process of inspecting a cell's neighborhood in the construction phase already. This has to be performed only once and saves time in the query phase.

**Traversal.** The actual traversal algorithm that performs a collision test can be recursively formulated as follows: At any given time, the algorithm has to inspect a sequence `[start, end)` of voxels at a certain level $d$ in the coarsening $\mathcal{C}_A^{(d)}$ of $A$. Initially, these are the top level voxels of $A$. A voxel's reference point $p$ is transformed into world coordinates and a lookup is performed in the voxmap $\mathcal{VM}_B^{(d)}$ at level $d$ of $B$. If the voxel is in conflict with the voxmap, we recursively proceed with its sub-voxels in level $d + 1$. The recursion stops if we find a conflicting bottom level voxel $v \in \mathcal{V}_A$. In this case, we

**Algorithm 6** COLLIDEVOXMAPR

```
//  T:      transforms A in world coordinate frame
//  start:  beginning of voxel sequence in C_A^(d)
//  end:    end of voxel sequence in C_A^(d)
//  d:      level of coarsening, d ∈ [0, MAX_DEPTH)
//  C_A, G_B, VM_B are assumed to be in the global scope

bool collideVoxmapR( Configuration T, int start, int end, int d )
{
   for(i = start; i<end; i++)
   {
      p = T · C_A^(d).point[i];
      h =  G_B^(d).hash( p );

      if(  VM_B^(d).contains(h) )
      {
         if( d == MAX_DEPTH-1 )
         {
            b = collideTriangles( T, i );
            if( b ) return true;
         }
         else
         {
            b = collideVoxmapR( T, C_A^(d).start[i], C_A^(d).end[i], d+1 );
            if( b ) return true;
         }
      }
   }

   return false;
}
```

determine the conflicting grid-cell $c \in \mathcal{G}_B$ in the grid representation of $B$ and test all associated triangles for intersection. Algo. 6 shows the pseudocode.



## 2.9 Benchmarks and Results

We have seen how we can use various data structures in order to accelerate collision queries between two triangle sets. As a first type of acceleration structure, we have used bounding volume hierarchies together with the common bounding volume types axis aligned bounding boxes (AABBs) and arbitrarily oriented bounding boxes (OBBs). When we build hierarchies with these bounding volume types, for every fixed hierarchy level, the volumes can overlap. This is because we request every triangle to belong to exactly one bounding volume of the same level. The triangles are assigned to the bounding volumes according to the relative position of their centroids to a splitting plane in the BVH construction phase. Thus, it is possible for triangles to straddle bounding volume borders.
Clearly, if the overlap is very high the hierarchy fails to give a clear spatial partition of the underlying triangle set. The scale of this effect depends on the quality of the triangle set and the used bounding volume type. Partitioning does not work well if the triangles have varying sizes or if the triangle set contains very long triangles with acute angles. A few malformed triangles can possibly mess up the structure of the whole hierarchy.
To lessen this effect, we relaxed the restriction that a triangle can be contained in only one bounding volume. There are several ways proposed in the ray tracing literature how this can be achieved [49]. The easiest methods suggest to simply refine the triangle set as a pre-process and then pass the refined set to an ordinary bounding volume hierarchy construction mechanism. Because the refinement cuts large, malformed triangles into smaller pieces, it is now possible for some triangles to be referenced in more than just one bounding volume leaf node.
Another opportunity is to alter the construction process, in order to allow the triangles to be referenced more than once. We have realized this approach in our box-tree as described in Sec. 2.5.2. The box-tree is a regular AABB tree, but it is always guaranteed that there is no overlap between the bounding volumes of a fixed level. In the construction phase, all triangles that straddle the splitting plane are referenced in both child nodes that are created by the split. To deal with large, misaligned triangles, we do not stop the construction when the number of referenced triangles is sufficiently small but take the size of a created bounding volume as a termination criterion. The box-tree is thus somewhere in between a regular bounding volume hierarchy and an octree-based approach and the termination criterion is related to the maximal cell size of the tree.

We have analyzed how different parallelization techniques can be used to improve the performance of collision queries. A collision query consists of numerous collision tests of two triangle sets at different positions. Because a single binary collision test typically only has a relatively small number of independent sub-tasks, it is a natural choice to treat every individual collision test between the two queried objects as one sub-task and to process a number of independent collision tests in parallel. We have used OpenMP to realize the parallel implementations on the CPU and Nvidia CUDA for a GPU implementation. Different choices of how to map the individual collision tests to the parallel GPU threads led to a number of CUDA kernels as described in Sec. 2.6. We have seen

how a careful choice of the memory access pattern can highly influence the data throughput of a simple CUDA kernel that reads large structs from the GPU global memory. In this chapter, we will see how the different access patterns influence the performance of a BVH traversal algorithm on the GPU.

In addition to bounding volume hierarchies, we have also discussed a grid-based approach in Sec. 2.7. It should be clear to the reader that the borders between our box-tree, a regular octree and a hierarchical grid are fluent and we could also see a hierarchical grid as a box-tree where all boxes of one level have the exact same size. As with the box-tree, the user can specify a cell size in the construction phase and we want to examine how this choice influences the performance. We recall that the main advantages of using a grid over a bounding volume hierarchy are that:

a) We can substitute every BV-BV intersection test by a simple look-up.

b) As every level of our multi-grid is a global representation of the static object, there is no need to traverse two hierarchies in tandem as is typically done with bounding volume hierarchies. It is sufficient to traverse the hierarchical representation of the dynamic object and to perform look-ups at the corresponding multi-grid level.

This also motivates, that our hierarchical grid representation uses a branching factor of more than two as opposed to our bounding volume hierarchy implementations. On the other hand, the main disadvantages of a grid-based approach are

a) that all boxes always have to have the same size and

b) that in order to allow simple look-ups of points in the grid, this grid is a somewhat over-conservative approximation of the underlying triangle set (because we are shrinking every voxel by half its diameter and offset the grid to compensate for this operation).

In this respect, there are certain similarities between a multi-grid and a sphere-tree, where every sphere bounds a voxel. The inherent relationship between box-trees and multi-grids makes it an interesting question to compare their performance with respect to a user defined cell-size in our benchmark scenarios.

### 2.9.1 The Benchmark Scenarios

We want to utilize collision tests for sampling-based motion planning in the next chapter. In sampling-based motion planning, collision tests are used to implicitly decide whether a certain point $q$ of the configuration space corresponds to a collision free placement of the robot object and thus belongs to the free space. To simulate the application of our collision tests in the context of motion planning, we designed our benchmarks to test a high number of randomly chosen configurations from a predefined sampling domain around the obstacle object.
As we will see in Chap. 3, depending on the actual motion planning algorithm, there are



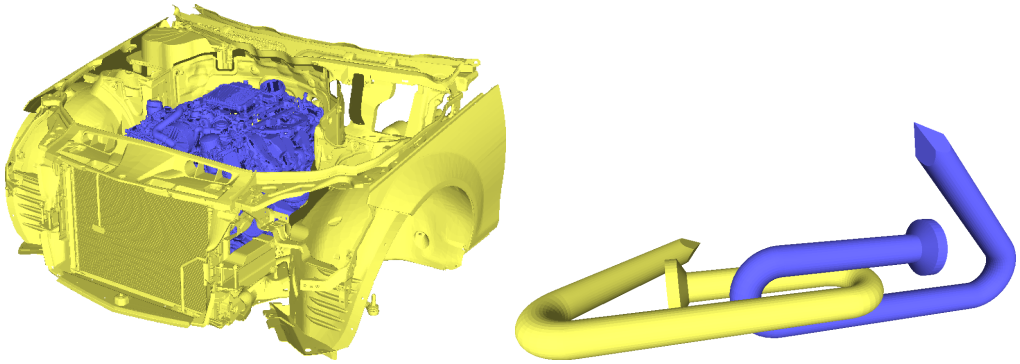

Fig. 2.8: The Nails and Engine-Bay Benchmarks. In both scenarios, we performed one million collision tests for random configurations of the robot (blue object).

several collision related tasks that have to be performed in the planning process like for example:

- decide for a number of configurations if they fall in free space,
- check if a local path that connects two nearby samples lies in free space,
- validate that the solution path fully lies in free space.

And it would probably be worthwhile to design specialized collision test mechanisms for these scenarios. For example, it could be beneficial to exploit the spatial coherency when checking small local paths or a solution path. These sophisticated techniques are beyond the scope of this work and we will not go into further detail about it.
We will use two different models for our benchmarks. The first scenario consists of an engine and an engine bay (Fig. 2.8, left). The related motion planning problem is to disassemble the engine. Our second scenario is the nails puzzle (see Fig. 2.8, right) and the objective of a motion planner would be to separate the intertwined nails on a collision free path.

### 2.9.2 Triangle Refinement

First, we want to examine how the distribution of the triangle sizes affects the performance of our bounding volume hierarchy based collision tests. Fig. 2.9 shows the algorithms' performance against the maximal triangle edge length. The top row of the figure shows the results for the nails benchmark scenario, the bottom row for the engine bay scenario. The ordinates in the left column shows the running time, whereas the right column shows the performance in terms of achieved tests per second. Both benchmarks perform one million collision tests and the robot is placed at uniformly distributed

points within the obstacle's bonding box with a random orientation.[11] We compare the performance of the following bounding volume hierarchy types:

**obb** standard top-down constructed OBB-hierarchy,

**pqp** collision test of the Proximity Query Package,

**obb-rel** same OBB-hierarchy as above, but with parent-relative traversal scheme,

**aabbE** top-down constructed, "volume minimal" AABB hierarchy.

Before the construction of the hierarchies, the triangles are refined so that the longest edge is smaller than the value shown on the abscissa of the corresponding plot. The refinement increases the number of triangles referenced by the bounding volume hierarchy leaf nodes. It can lead to deeper hierarchies with higher memory consumption. Moreover, as the pieces that result from the refinement of a triangle all reference this same parent triangle, it can occur that the same triangle pairs get tested during bounding volume hierarchy traversal several times. However, we can hope that applying the top-down construction scheme to small, evenly sized triangles results in better hierarchies than applying it to the original triangle set.

The analysis of the results is quite interesting. Let us first have a look at the OBB-type hierarchies. For the engine bay benchmark, we can see that as we refine the input triangles, the performance successively increases. Reducing the max. edge length from 250mm to 10mm, results in a performance improvement of around 41% for the standard OBB implementation and 34% for the parent-relative OBB test. PQP can also benefit from the refined input data (30% performance improvement). Things are a little bit more involved for the nails scenario: we observe an increase in performance when we refine the input triangles from 3mm to 2.5mm. From this point on, performance begins to drop again. It has another peak for a refinement length between 1mm and 1.5mm. The maximal performance improvement was less then 10% in our tests.

To better understand this, we have a look at the distribution of the triangle sizes in the corresponding meshes. Fig. 2.11 shows the histograms for the robot of the nails benchmark[12] and the robot and obstacle of the engine bay benchmark. The distribution of the maximal edge lengths of the nail model shows two clear peaks and almost all triangles have an edge length between 0.5mm and 2.5mm. There are no outliners, that have significantly higher edge lengths. Thus, cutting off the rightmost peak by choosing a refinement value of 2–3mm can slightly improve the performance but for even smaller values, traversal cost exceeds the marginal benefit of hierarchy restructuring and the performance begins to drop. We want to direct the readers attention to how the peaks in the distribution of the triangle edge lengths nicely match the performance peaks in Fig. 2.9. For the engine bay benchmark things are different. The distribution of the triangle sizes has a strong bias. Virtually all triangles have a maximal edge length of

[11] For a detailed description about how to sample random orientations refer to Sec. 3.2

[12] As the obstacle is the same mesh in an other position, the distribution is the same.



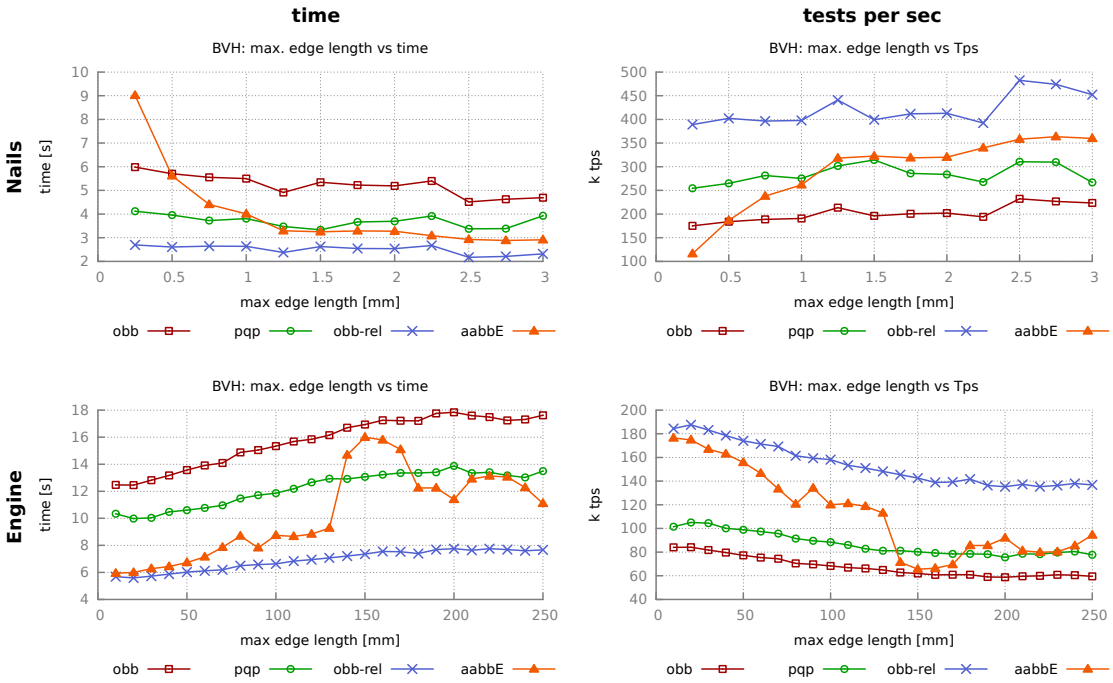


Fig. 2.9: The plots show the performance of one million BVH-based collision tests, when the triangles are trimmed to a maximal edge length before BVH construction. All tests run in parallel on a 4-core CPU.

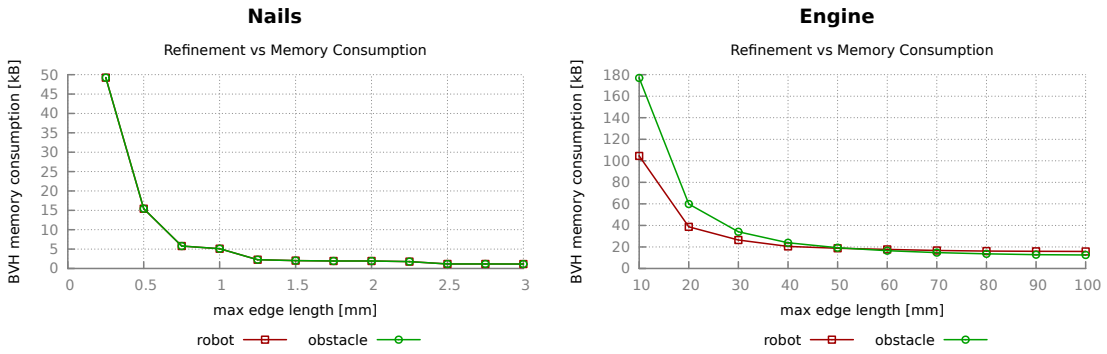


Fig. 2.10: The plots show the memory consumption of our OBB hierarchies when the triangles are trimmed to a maximal edge length before BVH construction.

0mm-100mm. But as we can see from the magnifications in the bottom row of Fig. 2.11, for both, the robot and obstacle model, there exist several hundred triangles with a very high maximal edge length. In this situation, triangle refinement can cut off the long tail of the distribution and improve the overall performance.

As far as the overall performance is concerned, we can see that the parent-relative OBB-hierarchy traversal scheme clearly outperforms the other approaches in our test scenarios. It can perform more than 450k test per second for the nails benchmark and around 190k tps for the engine scenario.

Now let us have a look at the AABB hierarchy. For the nails benchmark, we cannot determine any positive effect from triangle refinement. The improved approximation quality of the refined triangles with axis aligned boxes is only noticeable in deeper hierarchy levels and it cannot make up for the extra traversal cost. Similar effects are responsible for the strange behavior in the engine scenario. Both, engine and engine bay have relatively large axis aligned structures. Refining large triangles in these regions does not lead the volume optimizing construction scheme to produce an other splitting, here. At first, this increases the traversal cost without a noticeable improvement of the approximation quality. Only after refining a sufficiently large number of triangles, we can improve the performance. Interestingly, the approach works very good for an aggressive refinement of the input triangles and almost reaches the performance of the parent-relative OBB scheme in the engine scenario.

Obviously, giving an upper limit to the maximal edge length of the triangles increases the number of triangles and thus the number of bounding volumes in our hierarchies. Fig. 2.10 shows the memory consumption of the OBB hierarchies for a different refinement of the input triangles. We can see that for a very high refinement, the memory consumption begins to grow dramatically. In the engine bay scenario, the memory consumption of the obstacle BVH grows from around 15MB for the un-refined mesh to 180MB when trimming all triangles to a maximal edge length of 10mm. The number of triangles grows from 107k to 1.6 million triangles. Still, this is not much of a problem for todays hardware and bounding volume hierarchies of a few hundred mega bytes fit comfortably in our machine's memory and also in the video memory of modern GPUs.



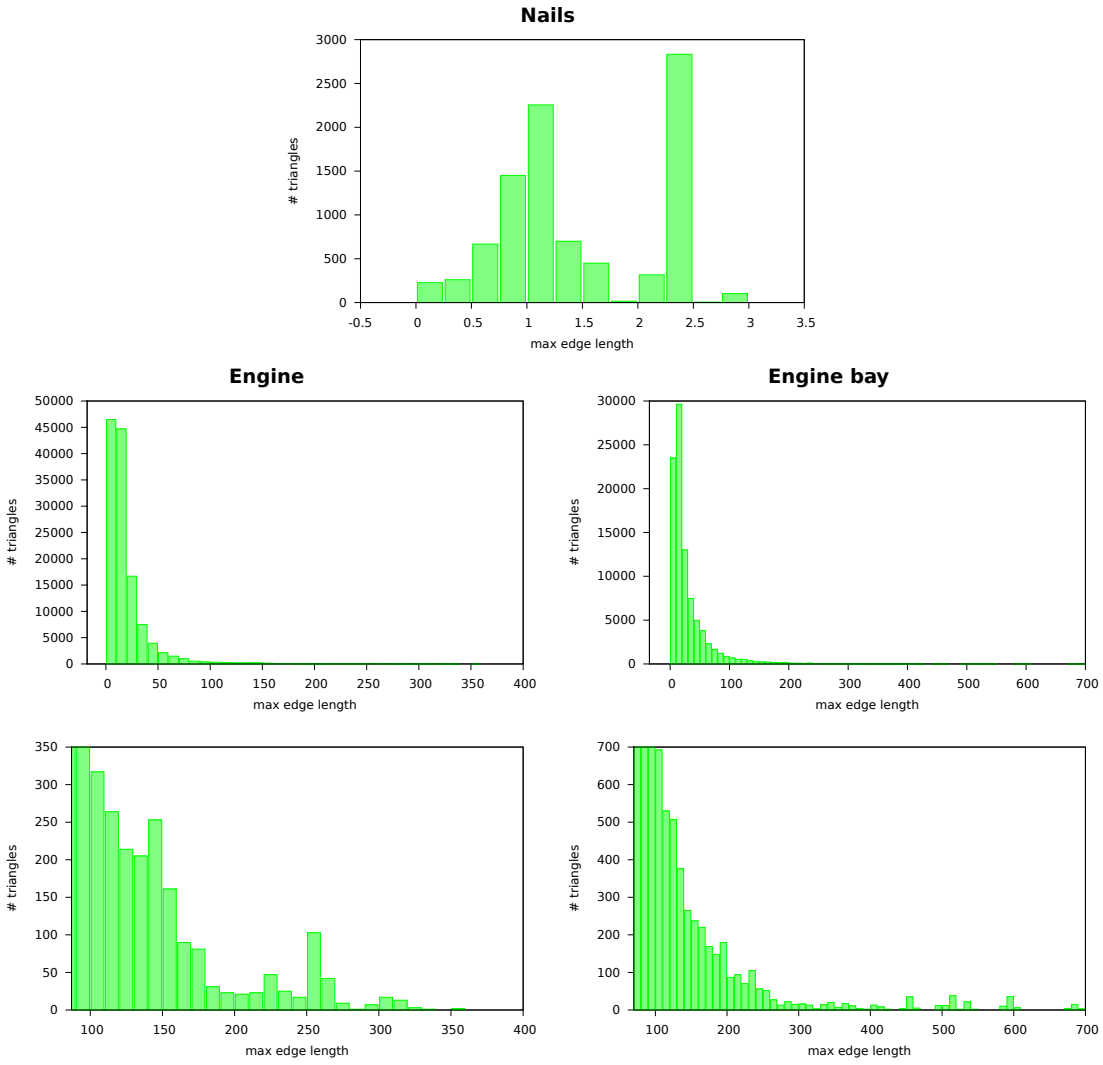


Fig. 2.11: The histograms show the distribution of the edge lengths for the different models. The bottom row shows a magnification for the tails of the distributions of Engine and Engine bay.

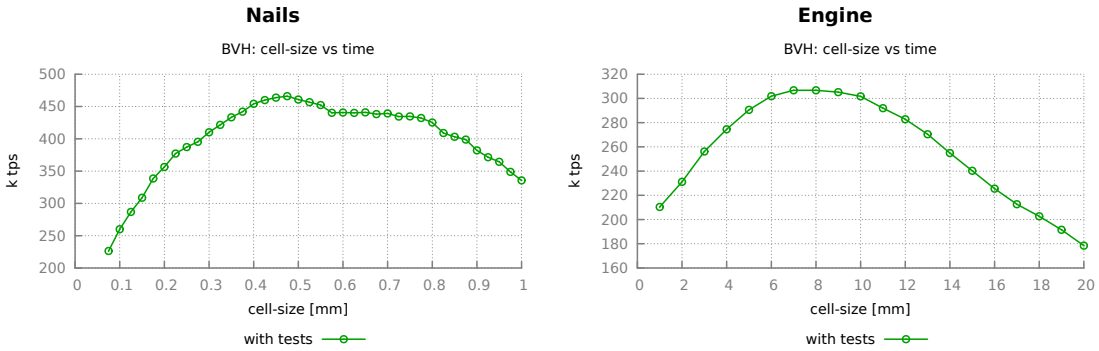


Fig. 2.12: Box-tree: cell-size against achieved tests per second.

### 2.9.3 Box-tree Cell-size

The box-tree (cf. Sec. 2.5.2) uses another technique to deal with malformed triangles. It performs spatial splits in the construction phase and duplicates the straddling triangles. Thus, just like with triangle refinement it is possible that a pair of triangles is tested multiple times during hierarchy traversal. But the resulting hierarchies are overlap free, which hopefully results in a good spatial separation. Fig. 2.12 shows the running time and achieved tests per second of the corresponding collision test as a function of the maximal cell-size. We use the nails and engine bay test scenarios from above with one million random samples each.
Clearly, a very shallow hierarchy with large cell size cannot approximate the underlying triangle set very well and is not spatially distinctive. Very deep hierarchies (small cells) on the other hand introduce a lot of duplicate triangle references and hence become more expensive to traverse. Between these two extremes, there has to be an optimal cell-size. We can see that for the nails benchmark it is best to use a cell-size between 0.4mm and 0.6mm. This results in a peak performance of around 460k tests per second. For our second test, a cell size of 6–10mm results in a peak performance of more than 300k tps.

### 2.9.4 CPU Parallelization

As each scenario consists of one million independent collision tests, it is very easy to process in parallel by viewing every collision test as a parallel task. On the CPU, this can easily be implemented by using the OpenMP library to split the main loop that iterates over the different configurations into several parts and to assign every part to its own thread. The above tests have all been parallelized in this fashion. All tests were performed on a Intel® Xeon® E5620 CPU @ 2.40 GHz with 4 cores. With respect to hyper-threading, this means that every core can physically process two threads in parallel. Thus, the CPU is fully occupied when processing 8 threads. Fig. 2.13 shows how



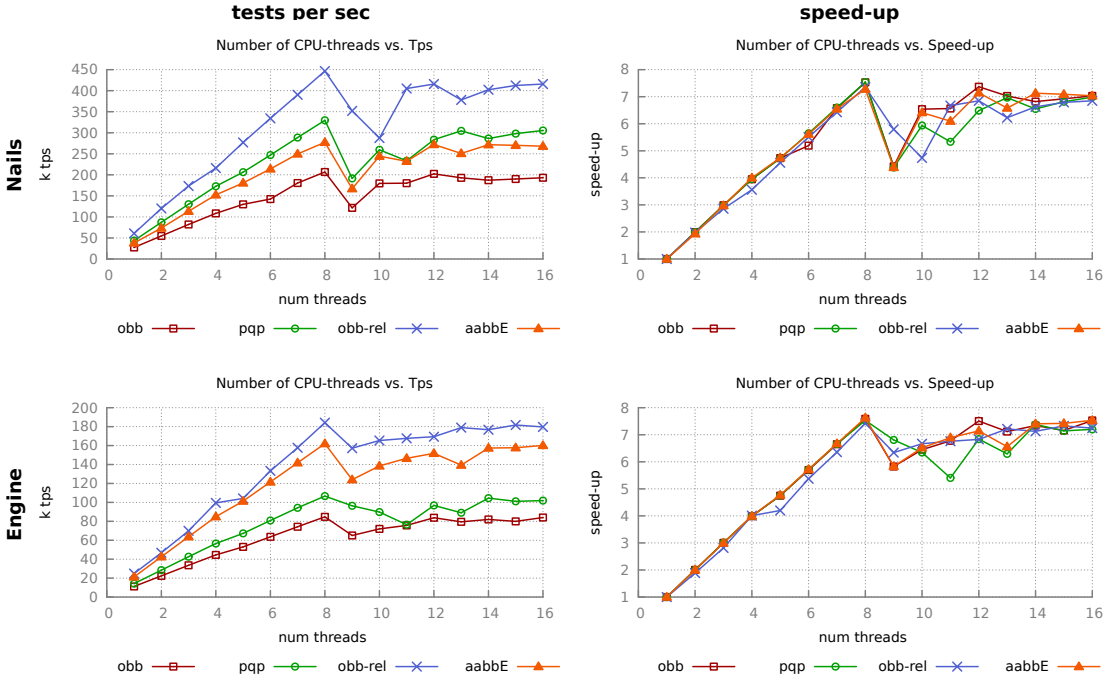


Fig. 2.13: CPU - Parallelization: number of threads against tests per second (left) and achieved speed-up relative to single-core implementation (right).

the number of threads affects the running time in our benchmarks. In practice, we use more than just 8 threads to compensate negative effects, when the tested configurations are not uniformly distributed in terms of the number of bounding volume tests that are necessary to process a single collision test. If it was the case that, for example, the first half of our tests could all be decided by just looking at the root bounding volumes and the second half would be very difficult to decide, then four of our eight threads would be finished quickly, leaving the CPU with only four active threads. As OpenMP statically assigns every thread an evenly sized portion of the main loop, we use a higher number of threads for better workload balancing. This approach is similar to the technique described in Sec. 2.6 in conjunction with our CUDA implementation. We found that running all tests in parallel on a 4-core machine with hyper-threading can improve the overall computation time by a factor of around 7.5. We recall that this improvement can be achieved with a minimal effort from an existing single-core CPU implementation by parallelizing the main loop with OpenMP.
We used the same mechanism to run the box-tree based collision tests in parallel on the CPU and achieved similar results.

### 2.9.5 GPU Parallelization

Next, we want to investigate how our CUDA implementations behave in our benchmark scenarios. There are different aspects, we want to discuss. First of all, the different traversal schemes described in Sec. 2.6 gave rise to the different kernel implementations we denoted as kernel A, B, C and E. We want to examine, how these patterns behave when performing a number of collision tests. The basic versions of kernels A and B make no use of the GPU's shared memory. We want to investigate two modifications of the kernels that try to take advantage of this GPU resource: (a) moving the traversal stack from global memory to shared memory in order to save global memory transactions and (b) using a shared memory data field to implement the Coalesced Scattered Reads global memory access pattern (see Sec. 2.6.7). Finally, we want to have a look at how the number of collision tests that are processed in parallel influence the GPU performance.
First, we are going to compare these implementations without performing triangle triangle intersection tests at the bounding volume hierarchy leaf level. In this setup, we are operating solely on the bounding volume hierarchies. This approach masks out the effect of the triangle intersection tests on the running time and focuses on the comparison of the different traversal schemes. We will have a look at how the algorithms behave when adding the triangle tests and compare approximate against exact collision tests, afterwards. We ran all tests on a Nvidia GTX 480 consumer graphics card.

**GPU – Kernel Version.** Fig. 2.14 (left) shows how the different kernel versions perform in our benchmark scenarios. We can see that the kernel versions A and B clearly outperform the other approaches. We were able to perform more than one million collision tests per second for the nails benchmark and around 480k tests per second for the more complex engine bay benchmark. Although version D is not too bad in the engine bay scenario, the more sophisticated traversal scheme of the kernel cannot compensate for the burden of inter-thread communication. The idea to let a group of threads share a common traversal stack cannot compete with the simpler versions A and B. Furthermore, our results show that it is not advantageous to use a very fine granular parallelization as was done with kernel C where we parallelized a single bounding volume test.

**GPU – Shared Memory Traversal Stack.** The CUDA kernels A and B make no use of the GPU's shared memory. We can therefore try to modify the kernels and hopefully take advantage of this resource. Our first attempt is to move the traversal stack of every thread to a shared memory array in order to save global memory transactions. Every stack entry is a pair of two integers and identifies a pair of bounding volumes that has to be processed. As we are using a depth first order traversal scheme, this stack cannot grow much in size. If we reserve stack space for 64 entries for every thread, we need 16KB of shared memory per warp. Because every SM only has 48KB of shared memory available, this means that a SM can only run three warps concurrently. The result of our experiment with a fixed stack size of 64 can be found in Fig. 2.15. The red and green bars show the same results we have already seen in Fig. 2.14 with a traversal stack



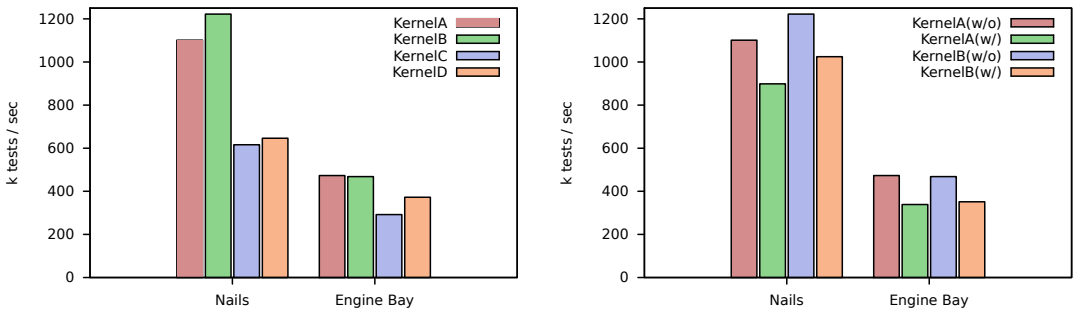


Fig. 2.14: **left:** Performance of the four different CUDA kernels. Kernels do not perform any triangle tests. **right:** Effect on the running time when running kernels A and B with or without triangle tests.

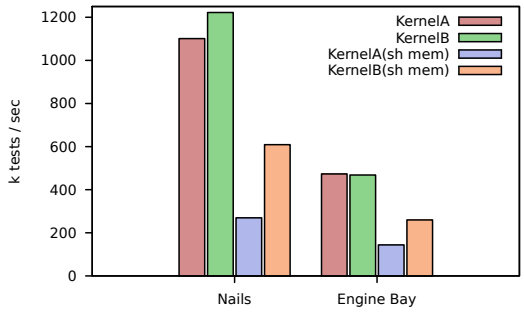


Fig. 2.15: Performance of Kernels A and B when the traversal stack resides in the GPU's shared memory. We use a fixed stack size of 64 entries.

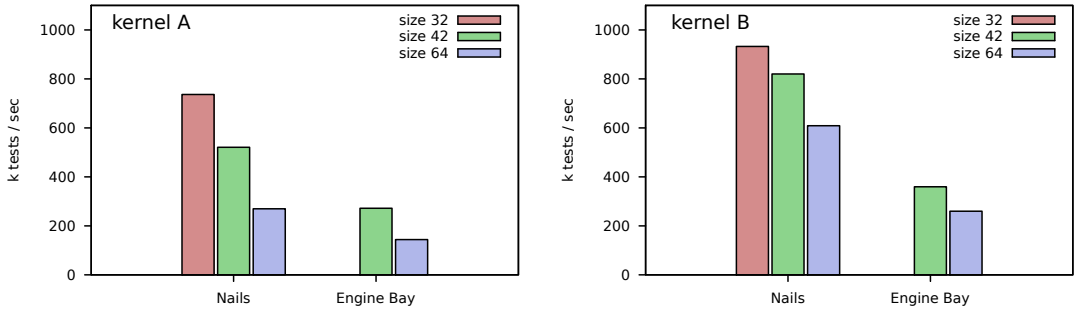


Fig. 2.16: Performance of Kernels A and B when the traversal stack resides in the GPU's shared memory. We compare different stack sizes.

that resides in global memory. We can see that for both kernel versions, the relocation of the traversal stack to the GPU's shared memory does not have a positive effect on the performance. The high resource consumption restricts the number of threads that can run concurrently per SM. This negative effect cannot be compensated by a reduced number of global memory transactions.
One option would be to reduce the stack sizes. For a stack size of 32 entries, every SM could run up to 6 warps concurrently. However, we found out that a stack size of 32 was not sufficient for all our test queries. High refinement rates or a high triangle count cause deep hierarchies which can cause stack overflow and incorrect results. In Fig. 2.16, we sum up our results for different stack sizes of 64, 42 and 32 stack entries. The plots show how reducing the stack size can improve the performance of the algorithm. However, in the engine bay benchmark, the use of a 32 element stack caused a stack overflow which resulted in a crash of the kernel and produced incorrect results. Small stack sizes can also be problematic when using an aggressive refinement of the input triangles as described earlier. If we compare these results with Fig. 2.14, we can see that the performance of the 32 element stacks is comparable to the global-stack version. Overall, the use of a shared memory traversal stack could not produce a significant performance gain.

**GPU – Performing Triangle Tests.** Fig. 2.14 (right) shows the effect of adding triangle-triangle intersection tests for the intersecting leaf nodes of the hierarchies. We show the result for kernels A and B only. Adding triangle tests causes increased computation time, extra time for loading the triangles and a higher traversal time because we probably do not stop the traversal for the first pair of intersecting bounding volume leaf nodes. It can also result in more divergent code as not all threads of a warp want to perform triangle tests at the same time. We found that if we perform triangle tests, the performance of our kernels drops by $30\% - 35\%$. This is comparable to the effect we have observed when performing the same test with the CPU version of OBB bounding volume hierarchies as we will see in Sec. 2.9.6. For the CPU version, we have only observed a performance improvement of 11% to 25%, when skipping the triangle-tests.

**GPU – CSR Memory Access.** As we have seen in the previous paragraph, although



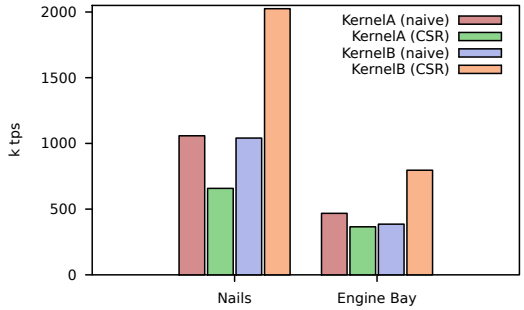


Fig. 2.17: Performance of CUDA Kernels A and B with and w/o using CSR the memory access pattern.

very promising at first sight, it was not helpful to use the GPU's shared memory to hold the traversal stacks. We can also use the shared memory to implement the Coalesced Scattered Reads traversal scheme described in Sec. 2.6.7. We have seen that the careful choice of a good memory access pattern can greatly improve memory throughput for a GPU-kernel that is reading numerous large structs (an OBB node consumes 64 bytes of memory). Even though the traversal algorithm performs a number of additional memory transactions for managing the traversal stack and reading the triangles[13], we can hopefully improve the performance of our algorithm with this pattern. In this section, we want to have a look at how the choice of the memory access pattern can influence the performance of kernels A and B.
Fig. 2.17 shows how the CSR versions of these kernels compare to the versions with a naive memory access pattern. We can see that while the performance of kernel A degrades with the new access pattern, kernel B can highly benefit from CSR accesses and almost double the collision query throughput. We recall that in kernel B, every CUDA thread performs not just one but a certain number $k$ of collision tests. Besides the access pattern we use to load the bounding volumes from memory, the number of tests that are assigned to every thread are also a parameter of the approach. Fig. 2.18 compares the performance of kernel B with and without using the CSR pattern against different group sizes $k$. We see that when using the CSR pattern, we should avoid to use small group sizes and that we can successively improve the performance by assigning more collision tests to a single thread. Things are different, when using the naive memory access pattern. Here, it is beneficial to use small group sizes and we achieved the best performance when using a group size of $k = 8$.

In the results we have just presented, we compared kernel A and B in the simplified version without taking the triangle intersection tests at the BVH leaf level into account. Fig. 2.19 shows the performance of our CUDA - OBB hierarchy traversal algorithm, kernel B, with triangle intersection tests. Like we have done with the CPU version, we plot the effect of triangle refinement against the throughput of the kernel in terms of tests per second. In addition to the GTX 480 graphics used so far, we also give the

[13]In our tests, we did not use CSR-access for reading triangles.

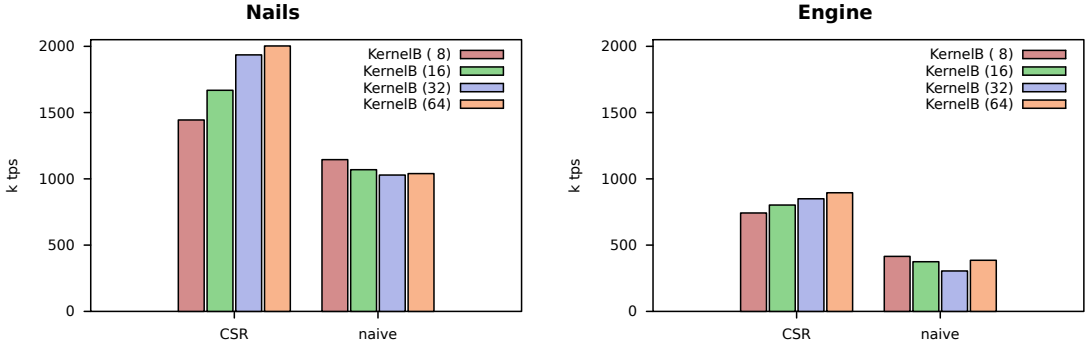


Fig. 2.18: Performance of Kernels B with and without CSR accesses for different group sizes $k = 8, 16, 32, 64$.

results when performing the same test on a Quadro 5000.

CSR access could improve the performance of the collision detection algorithm by a factor of 2 for the Quadro 5000 and a factor of 2–3 for the GTX 480. The qualitative response of the algorithm to progressively refined input triangles is comparable to the behavior of the CPU OBB-algorithms. We can see that the refinement of the already evenly sized triangles in the nails benchmarks brings hardly any improvement. If the triangles are getting too small, performance drops. For the engine bay benchmark, triangle refinement led to a performance improvement of around 100% for CSR access. The GTX 480 graphics card is two times faster than the Quadro 5000 in the engine bay benchmark, and 1.73 times faster in the nails scenario. This is consistent with our results form Sec. 2.6.7 as the GTX 480 has a higher peak memory throughput than the Quadro card. This strongly suggests that our BVH traversal kernel is memory bound. By using a better memory access pattern, we can improve the memory throughput of our kernel and therefore the collision test throughput. Of course, if our code is memory bound, it is also beneficial to use a device with a wider memory bus and higher peak memory throughput.

**GPU vs. CPU – Saturation of the Processing Unit.** As we remember from Sec. 2.4 about CUDA, modern GPUs are capable of running hundreds of threads in parallel and even thousands of threads concurrently. The GPU's thread scheduler actively uses a high number of concurrent threads to hide memory latencies. Thus, for the device to reach peak performance, it is necessary to always have a sufficiently high number of threads running. In our case, this means that we have to perform enough collision tests at a time. Because the CPU relies on other strategies for latency hiding like caching and branch prediction, it may not need as many jobs to fully utilize the device. None the less, we can expect that the CPU can also benefit from performing a sequence of tests right away as opposed to running just a single collision test. In our saturation test setup, we always perform the exact same collision tests, but we split them to an increasing number of (kernel) calls. If we have one million collision tests to be performed, we first process



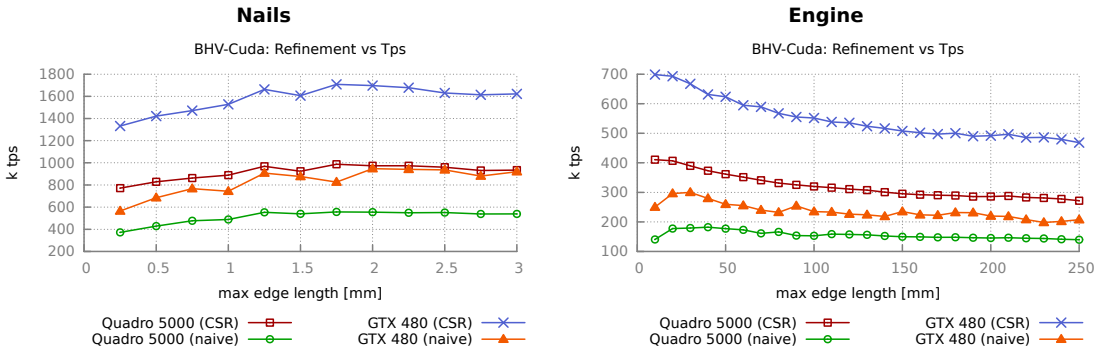


Fig. 2.19: Performance of Kernel B when refining the input triangles. Results are shown for two different graphics cards with and without CSR memory access.

all tests in parallel with just a single kernel call. We then process the same tests with two, four, eight, ... successive calls where every call processes one half, one fourth, one eighth, ... of the tests. We record the accumulated time it takes to process all collision tests to determine the throughput of our kernels for a different input sizes.
Fig. 2.20 shows the effect that the number of tests has on the throughput of the algorithms. We compare kernel A against kernel B of our GPU implementation. For kernel version B, we use different group sizes and show the results with and without CSR memory access scheme. Because we assign a number of collision tests to a single CUDA thread, we need a higher number of collision tests to saturate the GPU and achieve a high throughput. And with a growing group size $k$, we need successively more tests to perform in parallel. If we do not use the CSR pattern, it is not beneficial to use large group sizes. In this setup, we achieved the highest collision test throughput of our kernel when using a group size of 16 threads. For one million parallel collision tests, we could achieve a throughput that is comparable to kernel A in the engine scenario and even a higher throughput in the nails scenario. Utilization of the CSR pattern results in very good performance improvements as we have already seen in the bar charts in Fig. 2.17 and 2.18. Again, using larger group sizes requires more parallel jobs for a good device occupancy. But interestingly, we can achieve a higher peak throughput for large group sizes in this setup.
These kernels can perform even better if we have more than one million collision tests to be processed in parallel. CSR memory access can improve the performance of kernel B by a factor of 2 over the naive access pattern when we have a sufficiently high number of concurrent tests.
In Fig. 2.21, we compare our CUDA implementation of kernel B for different group sizes against the multi-core OBB based approach with a parent-relative traversal scheme.

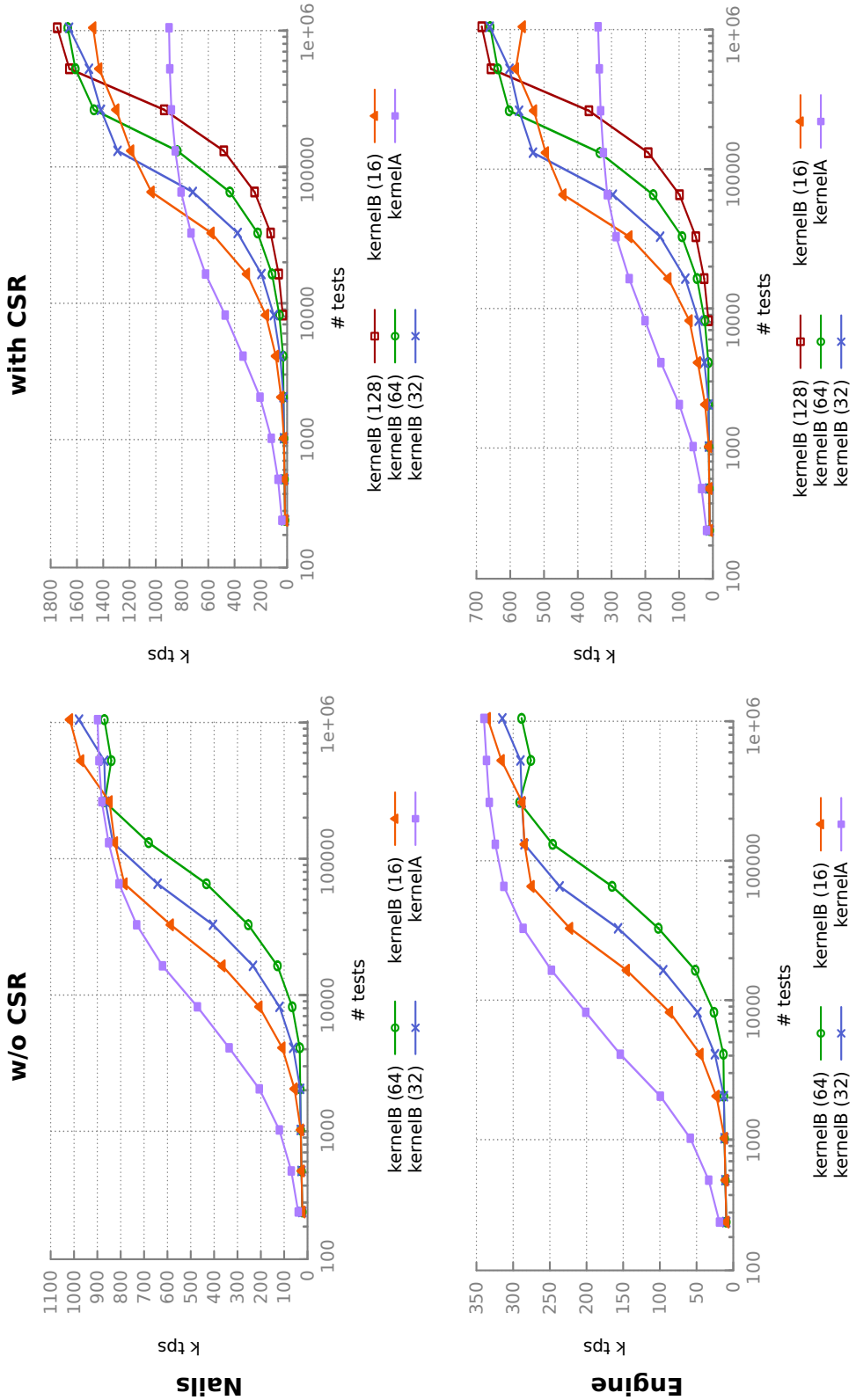


Fig. 2.20: Device Saturation: The figure shows the performance of an OBB-BVH based collision test against the number of parallel tests. We compare kernel A and kernel B for different group sizes.

As we can see, both the CPU and the GPU, need a certain number of tests to be fully occupied. The GPU is however much more sensitive to this effect. Let us have a look at the nails benchmark: Our multi-core CPU implementation runs at around 450k test per second when performing $2^{23}$ collision tests. This drops to about 90k tps or $1/5th$ of peak performance when running only 128 tests in parallel. But we already have 70% peak performance for just 1024 parallel tests! The CSR CUDA test has a peak of around 2M tests per second, which drops to only 25k tps ($\sim 1/80$ peak) when having only 1024 tests. In both benchmark scenarios, we observe that we need to have between 10k and 100k collision tests to have the GPU outperform our multi-core implementation. For less than 10k tests, we should prefer the CPU version. If we have several million collision tests we can process en block, we can achieve a speed-up factor of 4.5 - 6 when using the GPU compared to a 4-core CPU with hyper-threading.

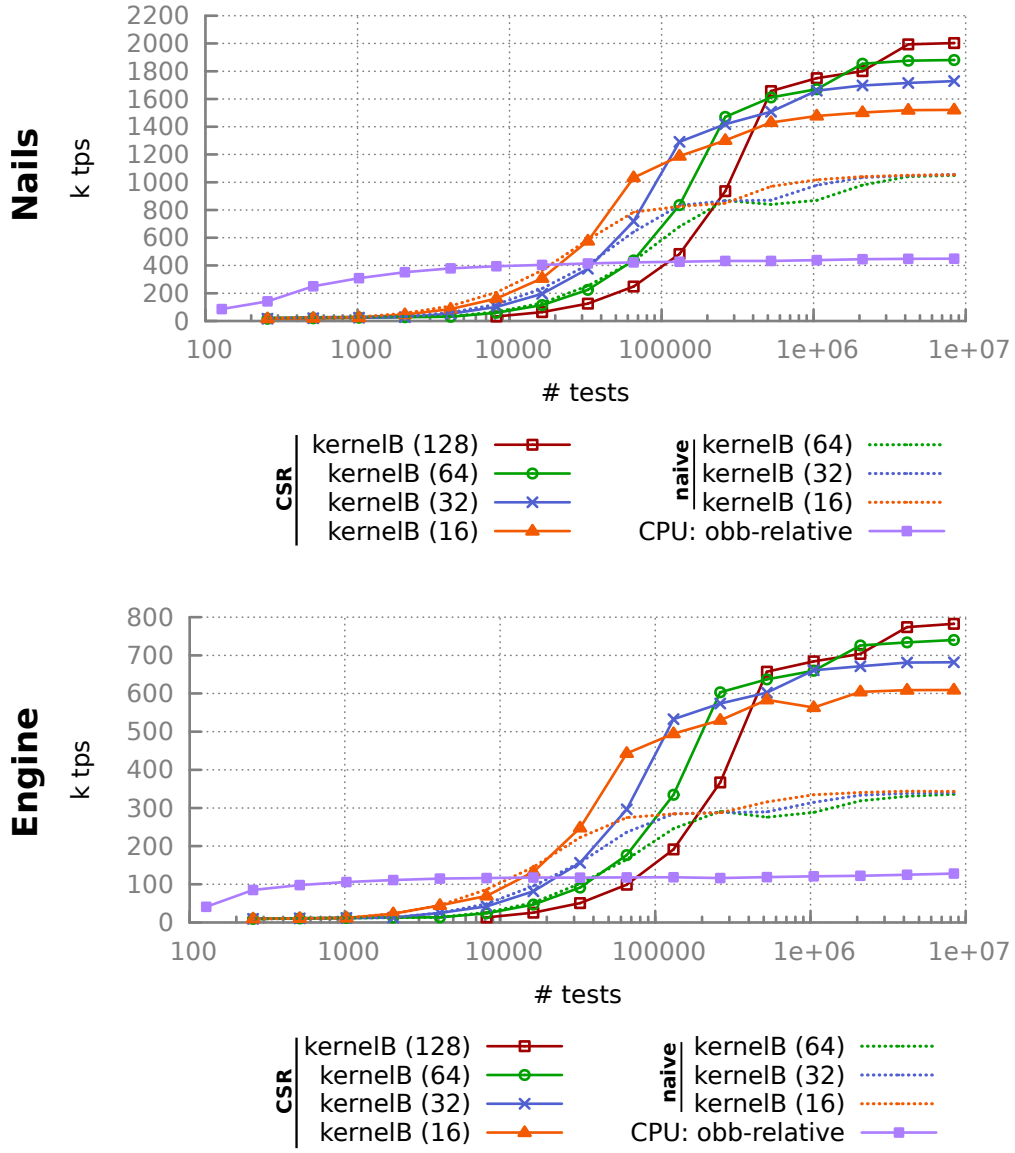


Fig. 2.21: Device Saturation. The figure shows the performance of an OBB-BVH based collision test against the number of parallel tests. We compare the CPU version and GPU kernel B.



### 2.9.6 Approximate Collision Tests with BVHs

With bounding volume hierarchies it is easy to implement an approximate collision test by simply dropping the elementary triangle-triangle intersection tests for intersecting bounding volume leaf nodes like we have already described for the CUDA kernels. The approximate algorithm already reports a collision whenever it finds two intersecting leave nodes. While this can surely improve performance due to code coherence, saving the time for the triangle tests and probably an earlier break of the traversal, it is still a conservative test. It can never miss collisions but only deliver false positive results.
Fig. 2.22 shows the effect that skipping the triangle tests has on the running time of our OBB-based collision test on the CPU. We can record a maximal performance improvement of around 25% for the engine bay benchmark and 10% for the nails benchmark.
Of course, an interesting aspect is how the use of an approximate test affects the correctness of the result. In Fig. 2.23, we can see how the triangle refinement influences the number of falsely detected collisions. Interestingly, the number of detected collisions is not necessarily a strictly monotonous function of the refinement rate. It is possible that decreasing the maximal edge length can increase the number of false positives. This is due to the fact that for the refined triangle set, the resulting OBB leaf nodes define a different covering of the original un-refined triangles. However, in the long term, the refined version tends to approximate the enclosed triangles more closely and the number of falsely detected collisions decreases. It is interesting that for our randomly chosen configurations, the actual number of falsely detected collisions is not too high. For the engine bay benchmark, choosing a moderate refinement rate of around 50mm results in only 4500 false positives (0.45%). For the nails benchmark, we have around 3500 incorrect decisions for a refinement of 1mm to 2mm. Choosing a more acute refinement, we can further reduce these numbers to 1300 and 750. The number of incorrect decisions would of course increase, if we tested predominantly collision free but almost colliding positions. However, in applications where we have to test randomly distributed configurations or if we can accept performing approximate collision tests, we can benefit from

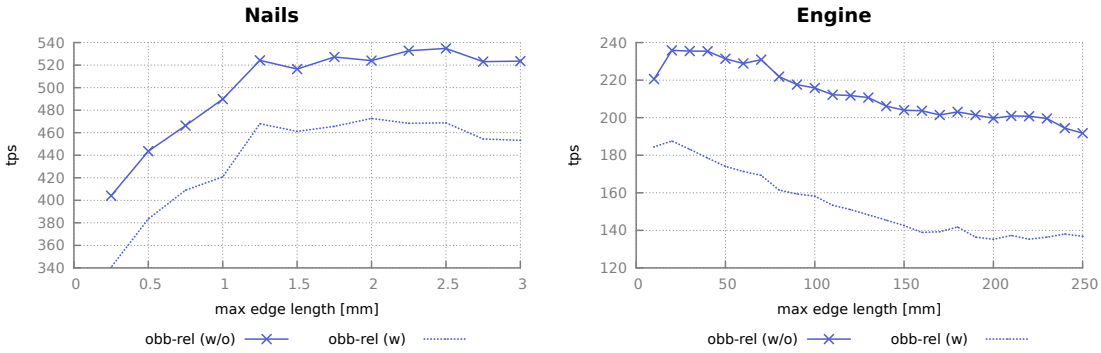


Fig. 2.22: Approximate BVH-based collision tests. The plots compare the versions with and w/o triangle tests for different refinement rates of the input triangles.

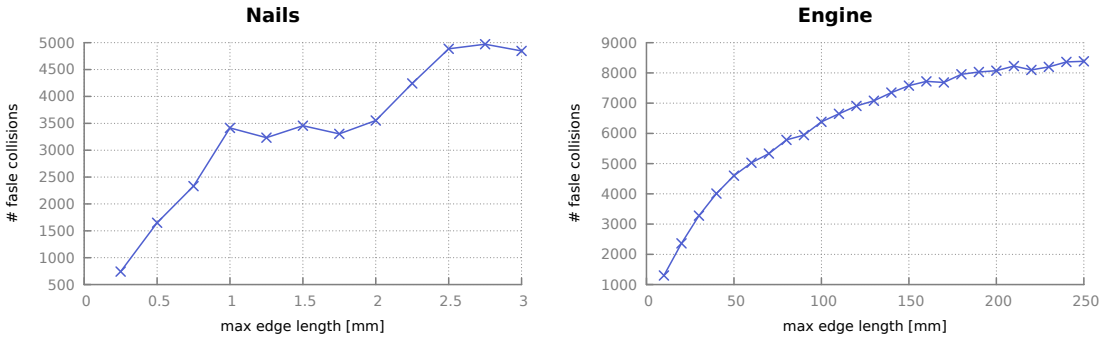


Fig. 2.23: Approximate BVH-based collision tests. The plots show the refinement of the input triangles against the number of falsely detected collisions.

the "no-triangle-version" of bounding volume hierarchy traversal. The idea produces absolutely no implementation overhead. It should also be noted that it is possible to estimate the maximal approximation error by the Hausdorff distance between the set of bounding volume leaf nodes and the bounded triangle set though we will not go into further detail about this.



### 2.9.7 Grid-based Collision Tests

To analyze the grid-based collision test described in Sec. 2.7, we first examine how the cell size influences performance. A very low grid resolution results in a poor approximation quality of the grid which means there is typically a high number of intersecting grid cells that do not induce triangle triangle intersection. Therefore, a possibly high number of grid cells has to be examined to determine the collision state of the underlying triangle sets. Increasing the resolution always increases the approximation quality of the grid. However, it also increases the number of triangle references in the grid cells. This can have a negative effect on the overall performance, as it can lead to repetitive execution of the same triangle intersection tests. This is especially true for non-colliding near contact situations. Moreover, the increased number of grid cells can have a negative effect on the running time of the algorithm as more and more cells have to be touched to detect a collision. Fig. 2.24 shows our experimental results for the same setup as with the BVH-based tests described earlier. We can see that as the cell size decreases the throughput of the algorithm increases. For very high grid resolutions the performance begins to drop again. It is interesting to compare this with the performance of the box-tree where the effect is more distinct (cf. Fig. 2.12 on page 57). The box-tree implementation has its performance peak at a cell size of around 0.5mm for the nails benchmark and 8.0mm for the engine bay benchmark. From this point on, the negative effects of increased performance cost and reference doubling exceed the positive effect of improving the approximation quality. For the grid-based approach, the performance begins to drop at a comparably small cell sizes of about 0.15mm and 4mm. This is because of different reasons: First of all, for the grid-based approach, a higher number of cells does not affect the traversal cost as much as it does for the box-tree. A single grid lookup is much cheaper than a box-box-intersection test. And during grid traversal, we do not have to keep record of all *pairs* of intersecting cells as is the case for bounding volume hierarchy traversal but only of the conflicting voxels of one object. Secondly, as the cells of the box-tree fit all triangles they contain and do not align exactly to the split positions of a spatial subdivision scheme, we can expect the box-tree to have a better approximation quality as a grid with the same cell size. This also implies that peak performance will be reached for larger cell sizes as with a grid-based approach.

### 2.9.8 Approximate Collision Tests with Grids

As with bounding volume hierarchies, we can drop all triangle-triangle tests when performing a collision test and define the collision state of the queried objects solely based on the objects' approximation as a grid. Whenever we find two intersecting grid cells at the lowest level grid, we declare the objects as colliding. This results in an approximate, yet conservative collision test.
The representation of the objects as a grid gives rise to a number of other, non-conservative approximation techniques as well. We recall from Sec. 2.7 how the basic grid-based collision test works: First, we compute a voxelization for the triangle set A with cell-size

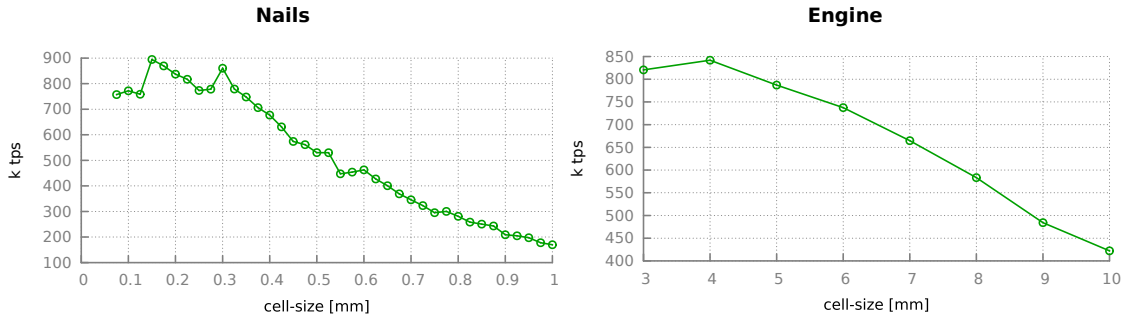


Fig. 2.24: Grid cell-size: The plots show the performance of our grid-based collision test for different cell-sizes.

$\delta_A$. We want to represent the static triangle set $B$ as a grid $\mathcal{G}$. Because we want to perform containment tests for this grid with the center points of the voxels, we mark a cell $c \in \mathcal{G}$ as occupied iff there exists a triangle $t \in B$ so that

$$d(c,t) \leq \frac{\sqrt{3}}{2} \cdot \delta_A,$$

where $\sqrt{3}/2 \cdot \delta_A$ is half the diameter of the voxels of $A$. This way, we account for shrinking the voxels of $A$ to points by performing an offset on the grid representation of $B$. In our implementation, we simplify and over estimate this offset voxelization. We first set the cell size of $\mathcal{G}$ to

$$\begin{aligned} \delta_B &:= \frac{\sqrt{3}}{2} \cdot \delta_A \\ &:= r \cdot \delta_A \end{aligned}$$

and mark all occupied grid cells. Then, we add all grid cells that have an occupied cell in their 27-neighborhood. We refer to this approach as building an offset of the grid representation. The *ratio* $r$ between the cell sizes $\delta_B$ and $\delta_A$ is chosen so that the "offset grid" conservatively accounts for the voxel shrinking.
Having this in mind, it is possible to generate an approximate collision test by omitting the grid-offset or by using a cell size ratio smaller than $\sqrt{3}/2$. In both cases, we can either perform triangle tests for the lowest grid level or skip the triangle tests. All together, this results in the following tests:

**wt:** with grid offset, with triangle tests,

**nt:** with grid offset, no triangle tests,

**nowt:** no offset, with triangle tests,

**nont:** no offset, no triangle tests.



**Adjusting the cell-size.** Fig. 2.25 shows how omitting the offset or the triangle tests affects performance and correctness of the resulting algorithm. The plots show the achieved tests per second and the number of falsely detected states for both benchmark scenarios. The third plot in a row shows a magnification for "nowt".
We can see that omitting the triangle tests can drastically improve the throughput of the collision test. However, this approach causes a high number of falsely detected collisions. If we have a look at the nails benchmark, we can see that for a moderate cell size of 0.5mm, the approximate collision query can perform up to 2 million tests per second which is an improvement by a factor of 4 compared to the standard (correct) test. In this case, the test produces around 60k false positive results. Reducing the cell size to the peak of the standard test, which was around 0.15mm, we still have 1.560k compared to 895k tests per second and the number of false results reduces to 20k. Again, we want to point out that this test is conservative. It will never miss a colliding configuration. This is not true, if we omit the grid offset.
The nont-test runs at 1.540k tps and produces 29k false results for a cell size of 0.5mm. This drops to 1.270 tps and 10k mistakes for a smaller cell size of only 0.15mm.
It is interesting that, compared to the other tests, the nowt-version produces a very small number of errors. If we look at the same cell sizes for the nails benchmark as above, we only have 27 mistakes for a cell size of 0.5mm and 10 mistakes at 0.15mm. At 0.2mm and 0.175mm we have 3 falsely detected configurations! This high improvement of the correctness comes from the fact that performing triangle collision tests for the intersecting grid cells rules out the false positive results and only leaves the small fraction of false negatives as mistakes. All errors that are illustrated in the plots for the nowt-version of the algorithm are false negative results. The throughput of this test is not as high as for the versions that skip the triangle tests, but it does not tend to drop to the original wt-test when the cell size increases. This way, we can achieve a relatively good performance improvement with a low error rate at moderate cell sizes. If we once more look at the example of the nails benchmark, we observe that for a cell size of 0.6mm, the performance can be improved by a factor of 3 compared to the wt-test. The approximation produces only 69 false results. Besides, the nowt-test at 0.6mm consumes only about $\frac{1}{45}th$ of the memory that the wt-version uses with its best resolution of 0.15mm.
We have only discussed the nails benchmark as an example, but the reader may convince himself that similar observations hold for the engine bay benchmark as well.

**Adjusting the cell-size ratio.** As we have described earlier, we can also influence the performance of the algorithms by using a different ratio $r$ between the cell-size of the grid representation of B and the voxelization cell-size of A. We have iteratively decreased the value of $r$ from $\sqrt{3}/2$ in small steps. In this test, we have used a fixed cell-size of 0.15mm and 4mm for the grid representation of the fixed object in the nails and engine scenario, respectively. Fig. 2.26 shows the results of our experiment. The reader should note that for a value of $r$ smaller than $\sqrt{3}/2$ this are all non-conservative approximate tests. We can see that for smaller ratios, the performance of all four algorithms increases. In the pictured range, we see a performance improvement of a factor of around 2 for the nails

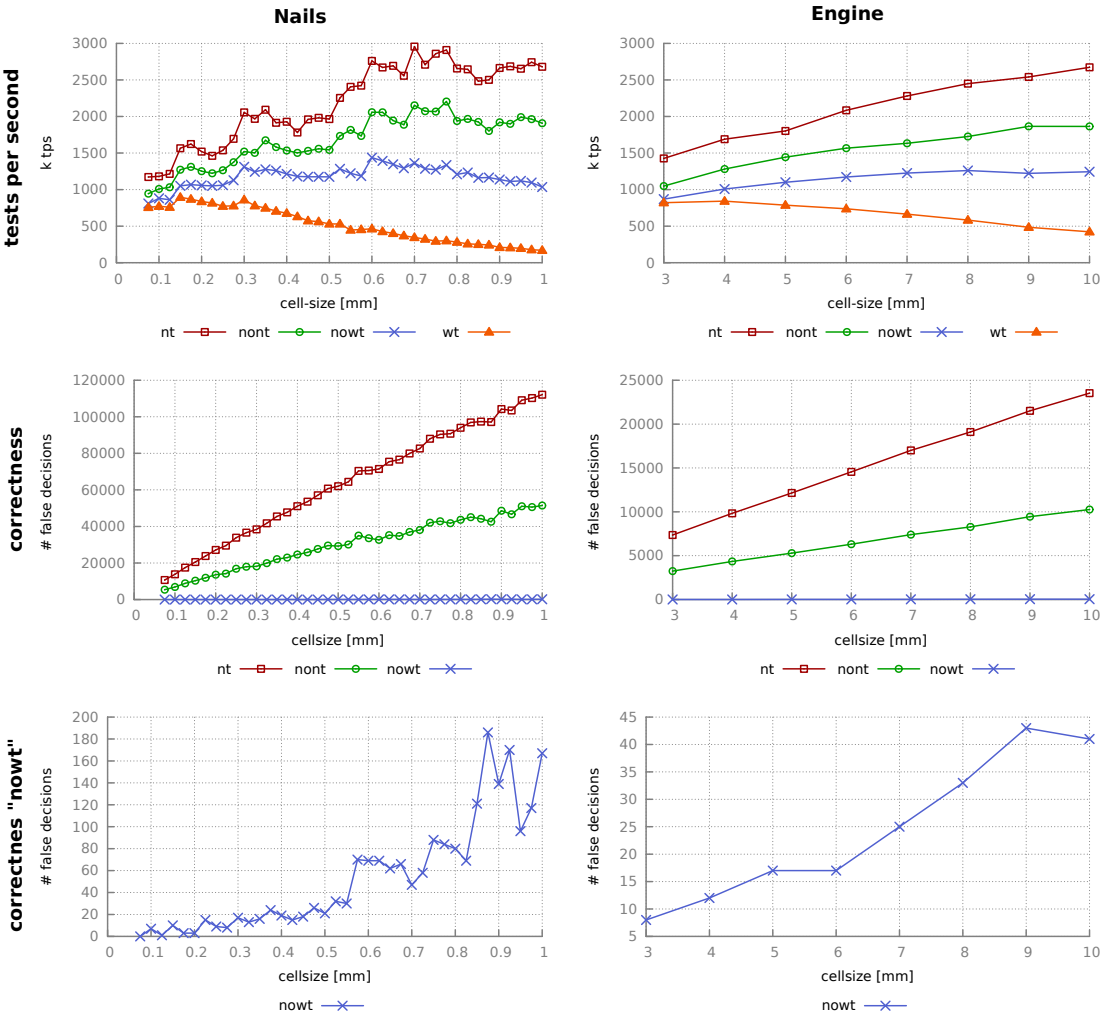


Fig. 2.25: Approximate collision tests with grids. By omitting the grid offset or the triangle tests, we arrive at different approximate collision tests. The plots compare the performance and correctness of these tests to the cell size of the underlying grid. The bottom row shows a magnification of nowt.



benchmark. Simultaneously, the number of wrong decisions decreases. This is because for smaller ratios, the number of falsely detected collisions dramatically decreases. On the other hand, the number of missed collisions can increase as we see in the third row of the figure for the nowt- and wt-versions. Again, by eventually performing triangle intersection tests, the false positive results are ruled out, leaving only the missed collisions as errors. Still, while allowing relatively high performance improvements, these two versions only produce a moderate number of a few dozen wrong decisions out of one million random collision tests.
Like with the approximate collision tests based on bounding volume hierarchies, it would be possible to bound the maximal error in terms of the maximal separation distance that two falsely positive rated objects can have and the maximal penetration depth that two falsely negative rated objects can have. These bounds could be given à priori based on the cell-size and cell-size ratio. We do not want to do this here, but the reader should note that these errors are bounded. This makes the tests an interesting choice for applications where small object interpenetrations are acceptable.

## 2.10 Conclusions and Future Work

We have implemented and compared different parallel collision detection algorithms for multi-core CPUs as well as many-core GPUs. We assumed to always have a high number of independent collision tests to be performed for the same two query objects but in different relative positions. On the GPU-side, we have implemented and compared four traversal schemes for OBB trees with a different granularity of work distribution amongst the parallel threads. Interestingly, our experiments have shown that the simpler implementations that do not split one collision test over several threads could outperform more sophisticated schemes in our benchmarks. This is far from obvious and we see this as an interesting result of this chapter. In addition to finding the best suited distribution of the parallel work amongst all threads, we could further improve the performance of our BVH-based OBB trees by a factor of two when using the novel CSR memory access pattern to load bounding volumes during the traversal.
On the CPU-side, we have analyzed four different BVH trees. We have shown that even though we have to keep track of the current transformation matrix on the traversal stack, it is beneficial to use a parent-relative traversal like proposed in the PQP library due to the reduced cost of the OBB intersection tests. We have compared the OBB trees with an AABB tree and what we called a box-tree, which is an overlap free version of a standard AABB tree. AABB trees have shown to be problematic in that they can require a high refinement of the input triangles for good performance. Axis aligned structures in the geometry of the query objects can lead to non-monotonous behavior in the algorithms performance for progressively refined input and very high refinement rates can cause performance penalties. On the other hand, the construction of a box-tree is controlled by the maximally allowed size of a leaf volume. We could clearly determine an optimal leaf size for our benchmarks and achieved a performance that is comparable

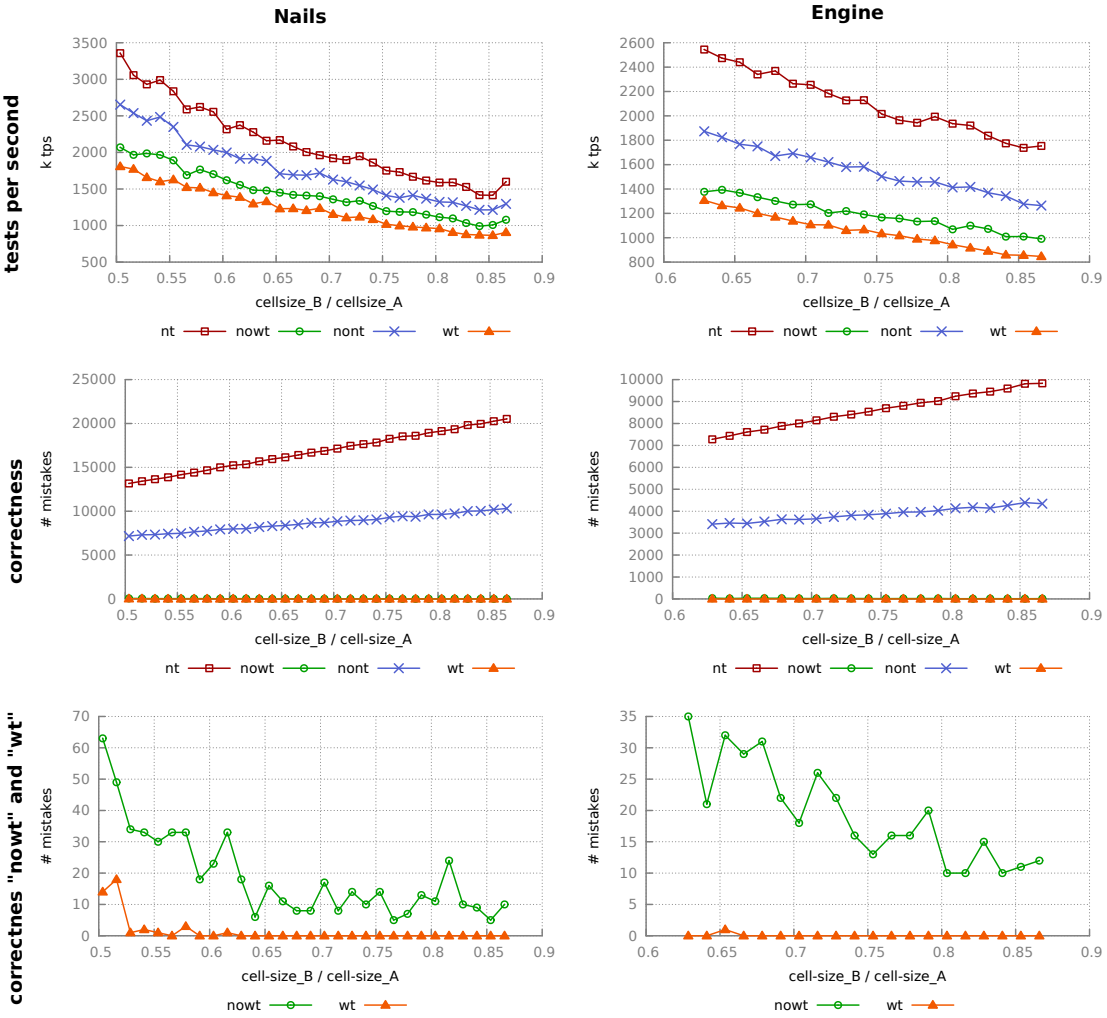


Fig. 2.26: Approximate collision tests with grids. We can additionally influence the performance of the algorithms by adjusting the ratio $r$ between the cell-size used for the voxelization of A and the grid representation of B. The plots show the performance and correctness for different ratios less than $\sqrt{3}/2$. We use a fixed grid size of 0.15mm for the nails and 4mm for the engine scenario.



to our parent-relative OBB implementation.
All CPU versions have been parallelized by statically distributing the individual collision tests to different threads with OpenMP. The reader might argue that we did not take as much effort as with our GPU implementations here, but we want to point out that even with this straightforward implementation, we could observe a speed-up factor of up to 7.5 on a quad-core machine with hyper-threading. This is close to optimal and so there is no need to further improve the implementation at this point.
We have compared the performance of our best CPU and GPU algorithms for different input sizes (i.e. number of parallel collision tests). It turned out that both, the GPU and the CPU, need a certain number of tests to be fully occupied. While for the CPU version a few thousand tests are sufficient, the GPU needs more than one million tests to reach peak performance. Depending on the kernel version used, the cross-over is at an input size between 10k and 100k configurations. For a sufficiently high number of configurations, we could achieve a speed-up factor of 4 to 5 for the GPU over the multi-core CPU version of our code. This is a factor of roughly 30 to 40 when we compare the GPU to a single-core CPU.
As an alternative implementation on the CPU, we have also proposed a hierarchical extension of the known voxmap-pointshell algorithm. We applied the same parallelization mechanism as with our BVH-based approaches. With an adequately tuned cell-size, we have achieved a two times speed-up over the parent-relative OBB tree. One problem with this implementation might be that big query objects in combination with relatively small cell-sizes might lead to a very high memory consumption. Still, we have not observed this in our benchmark scenarios. In practice, grid resolutions of up to $2^{10}$ are surely not problematic.
Another advantage of the grid-based approach is that it offers plenty of different possibilities for approximate collision tests and we have seen that approximate tests are a good choice to further improve the efficiency of the algorithm if we can accept some incorrect results in our application.

The reader may have noticed, that we did not present GPU versions for all the algorithms we have presented in this chapter. All GPU versions are built around a standard OBB traversal. Considering that we could achieve such good results with parent-relative OBB trees, box-trees and grids on the CPU, the reader might ask why we did not present the corresponding implementations for the GPU as well. The answer to this question is that it is a lot more involved to design an efficient GPU implementation using CUDA then it is to implement an equivalent CPU version. Moreover, it is not clear that the ideas that work well on the CPU also have to work for the GPU. As we have stated above, for parent-relative traversal, we have to keep track of the current transformation on the traversal stack. This shifts the instruction to memory operation ratio which can has a very negative effect for an execution on the GPU. Likewise, it is not trivial to come up with an efficient CUDA implementation for the box-tree traversal. Here, the CPU version of the code can profit from the precalculation of the projections to the 15 constant separating axis directions. In our experiments, this did not pay off on the GPU side. Finding an efficient GPU implementation for these algorithms is an interesting

and highly non-trivial avenue for future work. Still, it is not clear that if a performance improvement similar to the CPU-side can be achieved.
In the introduction of this chapter, we mentioned a much wider range of proximity queries than just boolean collision tests. The algorithms and data structures described in this chapter can also be used to handle more sophisticated proximity queries. It is an interesting question how the resulting approaches can be optimized and parallelized, respecting the limitations that are induced by the hardware capabilities. Again, this is especially true for parallel GPU implementations.

Of course, an other question that we are particularly interested in is how we can apply the algorithms we have developed in this chapter to the problem of sampling-based motion planning. We will further investigate this in the following chapter.



# 3 Sampling-based Motion Planning

## 3.1 Overview and Problem Statement

In this chapter, we want to use the parallel collision tests we have implemented in chapter one in the context of sampling based motion planning.
The term *planning* is used with a different meaning in a number of different fields. An excellent overview over the different types of planning algorithms along with theoretical background information, many interesting applications and specific implementation concerns is given in the book *"Planning Algorithm"* by Steven M. LaValle [29]. For our purpose, we want to use the term *motion planning problem* in a less general way. We want to assume that we have two 3D rigid body objects, that we refer to as *robot* $\mathcal{R}$ and *obstacle* $\mathcal{O}$. While the obstacle is a static object, the robot can move and its position in space at any given time is defined by a *configuration* $q$. Depending on the situation, $q$ can simply be the robot's position and orientation in space. If we think of the robot as a kinematic chain of numerous rigid bodies, connected at several joints, (e. g., a model of an industrial robot), $q$ could also describe the joint parameters of the chain. In either case, we call the set of all possible configurations $q$ the *configuration space* $\mathcal{C}$. Clearly, the complexity of the configuration space highly depends on the geometry of the robot. Specifying the configuration of a highly complex kinematic chain can require more parameters than the six degrees of freedom to determine the position and orientation of a rigid body in $\mathbb{R}^3$. The reader should also note that the structure and topology of $\mathcal{C}$ can vary with the representation. In the case of a rigid body in 3D, it would be convenient to set $\mathcal{C} := \mathbb{R}^3 \times SO(3)$. While it is easy to represent an element of $\mathbb{R}^3$ with three real-valued numbers, it is a non-trivial task to represent an element of $SO(3)$. There are different possibilities ranging from Euler Angles, axis-angle representation to quaternions and rotation matrices. Not only do these choices influence the number of required parameters (and thus the dimension of $\mathcal{C}$), but they have also implications on the topology of $\mathcal{C}$ that may or may not be desirable.

Every point $q \in \mathcal{C}$ brings the robot to a certain place $\mathcal{R}(q)$ in $\mathbb{R}^3$, the robot's *work space*. Not all configurations are feasible, though. Along with the robot, the obstacle lives in the work space, too. Clearly, we want to avoid that robot and obstacle are colliding or even interpenetrating. This introduces constraints to the feasible regions of the configuration space and partitions the configuration space into the free space $\mathcal{C}_{free}$ and the obstacle

space $\mathcal{C}_{obst}$:

$$\mathcal{C}_{free} := \{\ q \in \mathcal{C} \mid \mathcal{R}(q) \cap \mathcal{O} = \emptyset\ \},$$
$$\mathcal{C}_{obst} := \mathcal{C} \setminus \mathcal{C}_{free}.$$

Having this said, we can state the motion planning problem in the following way: Find a path $\alpha : [0,1] \to \mathcal{C}$, that lies totally in $\mathcal{C}_{free}$ and connects a given start position $q_0 \in \mathcal{C}_{free}$ to a set of goal positions $\mathcal{G} \subset \mathcal{C}_{free}$:

$$\alpha([0,1]) \subset \mathcal{C}_{free},\ \alpha(0) = q_0,\ \alpha(1) \in \mathcal{G}.$$

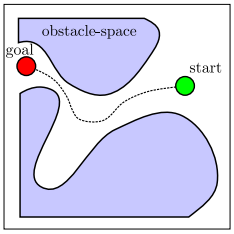


Fig. 3.1: Find a path from a start to a goal position that avoids the obstacle-space.

The set of goals could possibly only consist of a single configuration: $\mathcal{G} = \{q_1\}$. In practice, it is often prohibitive to explicitly construct the free space. Complex environments (i.e. complex robot or obstacle geometry) and high dimensional configuration spaces can lead to a high description complexity of the free space. In these cases, it becomes highly desirable to find a simpler representation of $\mathcal{C}_{free}$.

In the last decades, sampling based motion planners have been used successfully to solve motion planning problems with many degrees of freedom [26, 23, 15]. These methods represent the free space with a finite number of sample points. The set of sample points define the vertices of a graph. To capture the topological structure of the free space, two *nearby samples* are connected with an edge whenever they can be connected with a *local path*. In this manner, the structure of $\mathcal{C}_{free}$ is represented by a graph. This approach is advantageous in the following respects: (a) It avoids the explicit construction of the free space. (b) It divides the complex problem of path planning into smaller sub-problems, namely: Sampling of points in free space and planning of local paths between nearby points. The family of sampling based planners is, however, reliant on some concept of visibility. The basic idea is that two samples can be connected by a local path if they can "see one another", i.e., if there exists a straight line in $\mathcal{C}_{free}$ connecting them. This can cause difficulties with cluttered environments and *narrow passages*. A narrow passage can be hard to find for a sampling based planner and cluttered environments with fine structures are only vaguely rendered by the graph. This behavior is desirable in many cases, as it reduces the description complexity, but it can be problematic when the solution path has to pass such regions of the free space. There is previous work that addresses the problem of visibility in free space and its effect on the success of sampling based planners more precisely [3, 16]. And there has been substantial work on how to handle the narrow passage problem [16, 14, 51, 17]. This is also an issue we have addressed in the present work.

Every sampling based planner is to a certain degree composed of the following basic components: sample generation, determination of nearby samples and local planning. The design choices for these parts and how they work together make up the whole planner. In this paragraph, we want to give a brief sketch of these parts.

### Sampling Strategy

The first question that might come to mind when thinking about a sampling strategy is: How can we actually create a sequence of sample points in $\mathcal{C}_{free}$ if we have no representation of the free space? The idea here is to sample $\mathcal{C}$ and to discard all samples that belong to the obstacle space. This can be achieved by utilizing a collision test between robot and obstacle in the work space. The reader can already imagine that it can be very difficult to "accidentally" find a point that is inside a narrow passage of the free space.
Once we can generate a sequence of points in $\mathcal{C}_{free}$, we might think of other nice characteristics our sequence should have. In our case, $\mathcal{C}$ is an uncountably infinite set, but we can only create a finite number of points. To cover the configuration space tightly, we want the sequence to be a *dense* subset of $\mathcal{C}$. The sequence can be carefully chosen to be dense when using a *deterministic* sampling scheme. When using a *random* sequence of samples, we would want it to be "dense with probability one". The order in which the points appear in the sequence is also important. Typically, we want a nice distribution of the first $n$ points of the sequence (for any choice of $n$). As $n$ increases, the sequence should fill up the configuration space more densely, but it should not prefer some regions over others. However, there can be good reasons not to aim at a uniform distribution of the samples. For example, we could want to achieve a higher sampling rate in difficult regions of the free space.
For most configuration spaces, it is not very complicated to generate well behaved samples, but it can be challenging to sample $SO(3)$, and we will look into this in Sec. 3.2 in more detail.

### Finding Nearby Samples

In order to characterize the spatial proximity of two samples, and to be able to identify nearby samples, we have to define a metric on the configuration space, thus turning $\mathcal{C}$ into a metric space. There are different alternatives when it comes to choose an appropriate metric and it is not always clear what is the best choice. When constructing a metric, we have to consider the topological structure of $\mathcal{C}$. In addition, we would like our metric to be translation invariant. More general, if $\mathcal{C}$ has a group structure, we would like that the metric respects this structure in that it is invariant under the actions of the group elements. However, this is not absolutely necessary. It could even be reasonable not to use a metric but a pseudo metric on $\mathcal{C}$. For example, we could think of situations where it is easier for the robot to travel from configuration $q_0$ to $q_1$ than it is to travel from $q_1$ back to $q_0$. In this case, we could drop the requirement for our pseudo metric to be symmetric.
If we already had defined a meaningful metric $d_1$ for $\mathbb{R}^3$ and $d_2$ for $SO(3)$, what would be a good metric for $\mathcal{C} := \mathbb{R}^3 \times SO(3)$? A common choice is to use a weighted normal of these two metrics: $d := \sqrt{\alpha\ d_1^2 + \beta\ d_2^2}$. But another interesting opportunity is to use an "object metric". Instead of inspecting just two points $q_0$ and $q_1$ in the configuration



space, these approaches try to measure the spatial relationship between the corresponding objects $\mathcal{R}(q_0)$ and $\mathcal{R}(q_1)$ in work space [60].

Nearest Neighbor Finding

The choice of an appropriate metric can influence the overall planning result and one should carefully consider the different choices. Using a particular metric, the overall motion planning algorithm repeatedly determines candidate pairs of configurations to pass to the local planning subroutine. This requires an efficient handling of nearest neighbor queries. A naive approach to find the nearest neighbors for $n$ sample points has a time complexity in $O(n^2)$. Thus, if the algorithm needs a high number of samples, it is likely that the nearest neighbor queries will consume a substantial part of the running time. To avoid this, the samples are typically organized in an adequate data structure. The most common choice here is to use a kd-tree [58] though other approaches like bounding volume hierarchies [18] or locality sensitive hashing [41, 40] have also been used successfully.

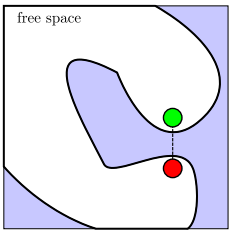


Fig. 3.2: A good metric on $\mathcal{C}$ does not induce a good metric on $\mathcal{C}_{free}$.

Unfortunately, despite all this effort, there remains one conceptional problem: Instead of having a metric on $\mathcal{C}$, we would rather like to have a metic on $\mathcal{C}_{free}$. In many cases this is no problem and we can expect that our metric will work well locally. But as we do not consider the topological structure of the free space, there can always be samples that are near in $\mathcal{C}$ but not in $\mathcal{C}_{free}$ (see Fig. 3.2).

## Local Planning

Let us assume that $q_0$ and $q_1$ have been identified as nearby samples. How can we efficiently find a collision free path that connects the samples? The most basic idea is to simply test the linear interpolation between $q_0$ and $q_1$ and to check if this path lies entirely in the free space. There are two main approaches to implement the validation of a candidate path. The first one is to sample the path with a high number of points and to perform discrete collision tests for the samples. We call this *sampling based path validation*. The second alternative is to utilize a continuous collision test to conservatively test the whole trajectory. We want to refer to this as *continuous path validation*. The main advantage that continuous tests have over sampling-based tests is that they can reliably eliminate "tunneling effects" and avoid collisions. For wide, open free space regions, it can also be very efficient to perform a continuous collision test as opposed to densely sample the trajectory and to perform many discrete collision tests. But typically, continuous tests are more expensive and more complex to implement compared to discrete tests.

To probe the linear interpolation as a candidate path is fast and easy to implement, but it does not consider any information about the environment. These approaches tend to work well in wide and unconstrained environments but they can have problems in highly constrained regions of the configuration space. Together with the sampling strategy, this is one of the main issues that causes the narrow passage problem with sampling based motion planners. More sophisticated local planners aim at using local workspace information to influence the construction of a local path [44, 46, 61].

Single query vs. Multi-query

Sampling, nearest neighbor finding and local planning are the basic components, that have to be addressed by virtually every sampling based motion planner. Furthermore, on a higher level, these planners can be roughly categorized in two classes: single query planners and multi-query planners. The first class of planners attempts to find a single path from start to goal without any preparation. It tries to only explore parts of $\mathcal{C}_{free}$ that are necessary to discover the solution path. Prominent representatives of this class are Rapidly-exploring Random Trees (RRT) and Expansive Space Trees (EST) [23, 15]. Multi query planners work in two phases. They first spend a substantial amount of time on the exploration of he free space as preparatory work. The prepared graph is then used later on to process different queries and the construction time is amortized over many queries. The most prominent representative of this class is the Probabilistic Roadmap family of motion planners (PRM) [26].

## Application and Problem Formulation

We are particularly interested in applying our motion planning algorithms to automatically compute collision free disassembly paths of rigid body objects. A practical application for this is, for example, maintainability studies with digital mock-ups in the field of CAD/CAM. In this field of application, we usually want the robot to escape a highly constrained region of the configuration space to reach a fairly unconstrained outer region (e. g., remove an engine from an engine bay). In addition, the robot potentially has to pass several narrow passages along a collision free disassembly path. Although very interesting in practical applications, this makes the problem hard to solve for sampling-based motion planners.
To test our algorithms especially with respect to highly restricted motion planning problems with many narrow passages, we consider a family of complex "motion planning puzzles", including the famous alpha puzzle (Fig. 3.3). To aim at the application of automatic part disassembly, we are only giving a dedicated starting position to our motion planner. The set of goal positions is defined implicitly as the set of all configurations that separate the bounding boxes of robot and obstacle for the corresponding relative placement.

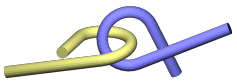

Fig. 3.3: The famous Alpha1.0 motion planning puzzle.

## Contribution

We have developed a novel single query sampling-based motion planner that is specially designed to tackle the narrow passage problem. The planner works in two phases: In phase I, we apply a retraction-based local planner that implicitly uses work space information via sample retraction. This helps the planner to find local paths in restricted areas of the free configurations space and thus improves the effectiveness of the local planning mechanism. In addition, we allow to optionally shrink the robot before the planning process in order to slightly dilate the free space. In the second phase, we use a novel type of planner (Locally Biased RRT, LBRRT) to locally re-plan the path from



phase I with enhanced precision. We utilize the parallel compute capabilities of modern multi-core CPUs to parallelize the local planning procedure which consumes the lion's share of the overall running time.
In the next sections, we want to discuss the concepts we have used and choices we have made in the design of our planning algorithms. We have found similar ideas in related work on the topic. Partly our ideas have been inspired by this previous work, partly we arrived at similar results independently. We will address this topic as we go and refer to previous work. Many of the basic concepts of our work are not fundamentally new, but the details of our implementation and especially their combination to an overall planning algorithm are. We have carefully considered the design of the components described above and arrived at a novel motion planning algorithm that is capable to solve very complex motion planning benchmarks with highly complex configuration spaces and many narrow passages in reasonable time.

## 3.2 Uniform Sampling of SO(3) Computing Random Rotation Matrices and Unit Quaternions

The configuration space for our rigid body motion planning problems is $\mathcal{C} = \mathbb{R}^3 \times SO(3)$ and our planner will have to create random samples in the configuration space. It is intuitively clear, that we can generate uniformly distributed random samples in $\mathbb{R}^n$ (or an axis aligned box $[0,1]^n$) by generating independent uniform samples for every component but not so clear how to generate random, uniformly distributed samples in SO(3). In this section, we want to describe how we can achieve this to obtain uniformly distributed rotations/orientations.
2D rotations can by characterized with a single angle and it is not surprising that uniformly distributed rotation angles lead to uniformly distributed rotations. But things are more complicated in the 3D case. A 3D rotation can be represented by an axis of rotation and a rotation angle. Now, one could come to the conclusion that choosing a random axis and a random angle with a uniform distribution would lead to uniformly distributed rotation. However, this is not the case (see [48] for example).
What do we even mean when we are speaking of a uniform distribution in the rotation group SO(3)? As SO(3) is a lie-group, we can define a distribution to be uniform, if it does not change under the action of the group elements. This is closely related to the definition of the Haar measure, which is the unique (up to scale) measure on SO(3) that is invariant under rotation (cf. [57]).
It is known that the unit quaternions form a double cover of the group SO(3) and that quaternion multiplication corresponds to the concatenation of rotations under this projection. Thus, we can obtain uniform random rotations by generating random quaternions, i.e. points, that are uniformly distributed on the sphere $S^3$.

There are several ways to generate uniformly distributed random unit quaternions. One way is to let the components of a quaternion be four Gaussian distributed random

variables and to normalize the resulting quaternion [48]. Another easy way is to generate uniform random samples in $[-1,1]^4$ and to discard all samples not in $S^3$. If we normalize the remaining samples, we also obtain uniformly distributed unit quaternions. However, as the unit sphere in $\mathbb{R}^4$ has a volume of $\frac{\pi^2}{2}$ and the cube $[-1,1]^4$ has a volume of 16, we had to discard about 70% of the samples.

An interesting alternative to generate uniform random samples in SO(3) that is exploited by Shoemake [48] and Arvo [1] is to make use of the nested group structure of the group of unit quaternions, $S^3 \cong S^2 \otimes S^1$, and to use the so called subgroup algorithm.[1] Shoemake describes it as follows: For every choice of a fixed axis, the planar rotations around this axis form a subgroup of the group of rotations in space. The coset of this subgroup can be represented by a rotation that brings the fixed axis to an other direction. Now, the basic idea behind the subgroup algorithm is that if we first choose a uniform random element from the subgroup and multiply it with a uniform random coset representative, we will get a uniform random element of the complete group.[2] To achieve this, Shoemake multiplies a quaternion that performs a random rotation around the $z$-axis, passing through the north pole of the unit sphere, with a rotation that randomly places the north pole at an arbitrary point on the sphere. He concludes with the following algorithm to generate random, uniformly distributed points on $S^3$:

a) let $x_0, x_1, x_2 \in [0,1]$ be independent and uniformly distributed

b) compute the uniformly distributed angles $\theta_1 = 2\pi\ x_1$ and $\theta_2 = 2\pi\ x_2$

c) and their sines and cosines $s_1, c_1, s_2, c_2$

d) also compute $r_1 = \sqrt{1 - x_0}$ and $r_2 = \sqrt{x_0}$

e) return the unit quaternion $q = (s_1 r_1,\ c_1 r_1,\ s_2 r_2,\ c_2 r_2)$

In [1], Arvo uses the same strategy to generate uniform random rotation matrices. He also combines a rotation around the $z$-axis with a rotation of the z-axis that brings the point $z = (0,0,1) \in S^2$ to a random point $p \in S^2$. He points out that this second transformation can be elegantly achieved by a Housholder reflection

$$H = I - 2vv^{\mathrm{T}},$$

with $v$ being a unit vector in the direction of $\overline{zp}$. To turn this into a rotation, he makes use of the odd dimensionality of $\mathbb{R}^3$ and scales with -1. He arrives at the uniform random rotation matrix

$$M = -HR.$$

[1] This notation shall indicate that the sphere $S^3$ is made up of fibers, where each fiber is a circle $S^1$ and there exists one fiber for every point of $S^2$. Locally, $S^3$ looks like the product $S^2 \times S^1$.

[2] For a more detailed description and the proof of this statement please refer to the original work by Shoemake.



Here, R is a rotation matrix around the z-axis and the vector $v$ that defines the Householder reflection is carefully chosen to result in uniformly distributed points $p$:

$$R = \begin{pmatrix} cos(2\pi x_1) & sin(2\pi x_1) & 0 \\ -sin(2\pi x_1) & cos(2\pi x_1) & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
$$v = \begin{pmatrix} \sqrt{x_3} \cdot cos(2\pi x_2) \\ \sqrt{x_3} \cdot sin(2\pi x_2) \\ \sqrt{1 - x_3} \end{pmatrix}.$$

$x_1, x_2, x_3 \in [0, 1]$ are again assumed to be uniformly distributed and independent random variables. With the vector $v$ from above, the corresponding Householder transformation $H$ takes the point $z := (0, 0, 1)^\top$ to a random point in $p \in S^2$:

$$\begin{aligned} p = Hz &= z - 2vv^\top z \\ &= \begin{pmatrix} -2\sqrt{x_3(1 - x_3)} \cdot cos(2\pi x_2) \\ -2\sqrt{x_3(1 - x_3)} \cdot sin(2\pi x_2) \\ 2x_3 - 1 \end{pmatrix}. \end{aligned}$$

We can see that if $x_2$ and $x_3$ are uniformly distributed, the point $p$ has a uniformly distributed $z$-coordinate and azimuth angle. It may be surprising, but this leads to an uniform distribution of the point $p$ in $S^2$. This can be motivated by understanding that the area of any spherical segment[3] of height $h$ is simply $2\pi h$; the area only depends on the height of the spherical segment. Thus, if we cut the unit sphere into slices of equal thickness, all slices have the same area. A random point in the sphere can then be generated by first choosing a slice at random and then choosing a random point in this particular slice. The interested reader may check this result by verifying that when using the $z$-coordinate and the azimuth angle $\vartheta$ to integrate over $S^2$, the surface element has the form $dA = dz\ d\vartheta$.

## 3.3 Collision Response
## Resolving Collisions Using Contact Simulation

In this section we want to describe our basic retraction algorithm for two triangle meshes. The approach uses an underlying collision detection algorithm to determine a number of collision points. Based on these reference points, we apply a small retractive impulse that aims at resolving the object interpenetration. We then iterate this process until we manage to resolve the interpenetration or a maximal number of iterations has been reached. In this section, we describe the approach when using bounding volume hierarchies as an acceleration data structure for collision detection, but the same concept can

[3] A spherical segment of $S^2$ results from cutting the sphere with two parallel planes.

easily be used together with other collision detection algorithms that use a hierarchical approximation to localize collision points. For example we have also implemented it together with our collision detection algorithm based on hierarchical grids as described in Sec. 2.7.

From an abstract point of view, we use a collision detection mechanism to localize a finite number of collision points $p_0, \ldots, p_{k-1}$ on the surface of our robot. We assume that the points have well defined surface normals $n_0, \ldots, n_{k-1}$. With this information, we compute a small translational and rotational momentum

$$P = \sum_{i=0}^{k-1} n_i \text{ and}$$
$$L = \sum_{i=0}^{k-1} (p_i - c) \times n_i,$$

where $c$ is the center of mass of our robot. Based on these quantities, we perform a small translational and rotational movement of the robot to resolve the interpenetration. We have observed that this heuristic works well in our context and it can help to resolve the small penetrations that occur in the local planning process. The resulting algorithm can be easily implemented with only small changes from an existing collision test.

**A BVH-based Retraction Algorithm.** Alg. 7 shows how we have modified a regular bounding volume hierarchy traversal scheme to sum up the contact momenta. The resolve algorithm performs a conventional BVH traversal and tests bounding volume pairs (r,o) for intersection. If the objects $\mathcal{R}$ and $\mathcal{O}$ are not colliding, the algorithm just reports their collision status and returns *false* to indicate that no retraction step has been performed. In case we find two intersecting leaf nodes, we apply a small retractive momentum $p$ and torque $l$ that would locally resolve the interpenetration. These local quantities are added to a global impulse and angular momentum that will be used to affect the configuration $q$ of the robot. We do not intend to produce physically accurate behavior, neither do we expect the approach to be always able to separate the colliding objects. So we simply use $P$ and $L$ to update the robot's configuration according to a fixed rotational and translational step size. We move $\mathcal{R}$ with a step size of $c_t$ in the direction of $P$ and rotate it around the axis defined by $L$ with a fixed rotation angle of $c_\varphi$.

The pseudocode of Alg. 7 shows how this can be implemented by just adding a few lines of code to a BVH-based collision test. Although it is very instructive, this simple implementation has one main drawback. It does not stop the while-loop on the first intersecting triangle pair but it determines all pairs of intersecting triangles. This has of course a drastic effect on the running time of this algorithm compared to a simple collision test. On the other hand, considering only one collision point in our computation of $P$ and $L$ can lead to undesired effects when the robot is in collision at different places. To improve the efficiency of the algorithm, we decided to cut off the top of the hierarchy



**Algorithm 7** RESOLVE

```
bool resolve(
   Configuration q [in,out],
   Bvh R [in],
   Bvh O [in] )
{
   P = ( 0,0,0 ); //total linear momentum
   L = ( 0,0,0 ); //total angular momentum
   c = center_of_mass( R );

   S = { (R.root, O.root) }
   while( S ≠ ∅ )
   {
      (r,o) = S.pop();
      if( !intersect( r(q), o ) ) continue;

      if( !leaf(r) || !leaf(o) )
      {
         S.push( child_pairs( r,o ) );
      }
      else
      {
         if( {bounded triangles intersect} )
         {
            p = o.normal − r.normal; // direction of the impulse,
            l = (r.center − c) × p; // cf. Fig. 3.4

            P += p;
            L += l;
         }
      }
   }

   P.normalize();  P *= c_t;
   L.normalize();  L *= c_φ;

   q.trans += P; // translate by c_t in direction of P
   q.rot   *= L; // rotate by angle c_φ around axis L

   return ( P ≠ 0 || L ≠ 0 );
}
```

for the robot at a user defined level so that the truncated hierarchy now consists of a number of root nodes. For every such root note, we attempt to find at most one collision point. This way, we first split $\mathcal{R}$ into distinct regions and we then compute the collision points for these regions separately. This approach can also be applied to a grid-based collision test.

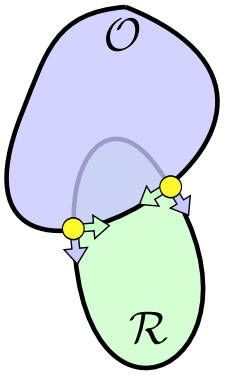


Fig. 3.4: Retractive impulses are applied near the line of intersection (yellow spots).

**Direction of the impulses.** For the direction of the small impulses, we use the sum of the surface normals of $\mathcal{O}$ and the inward facing surface normal of $\mathcal{R}$ at a the contact points (Fig. 3.4). Both directions can be heuristically motivated: the inward facing surface normals of the robot correspond to a pressure that acts on the surface of $\mathcal{R}$ as it penetrates $\mathcal{O}$; the outward facing normals of the obstacle help to move the intersecting parts of $\mathcal{R}$ to the outside. There are situations where these simple heuristics can fail to resolve an object interpenetration. But for small object interpenetrations, as they are evoked by a small step in a local planning algorithm, it will usually work fine.
In our experiments, we found that it is usually a good choice to use the average of an inward normal of $\mathcal{R}$ and an outward normal of $\mathcal{O}$ as a direction to apply the local contact impulses. One should keep in mind that depending on the situation, it could also be beneficial to use other weighting factors. If, for example, one of the objects does not provide meaningful surface normals, we could decide to use only the normals of the other object.

## 3.4 Local Planning Using Contact Information

We are now going to discuss how we have utilized the resolve algorithm in a retraction based local planner. In the local planning process, we attempt to connect two nearby samples, $q_{start}$ and $q_{goal}$, with a collision free local path. The easiest family of local planners just probes if the straight line that connects $q_{start}$ and $q_{goal}$ in the configuration space is collision free. They sample this line with a finite number of configurations according to some maximally allowed step size and perform a rigid body collision test for every sample. In our case, we have $\mathcal{C} \cong \mathbb{R}^3 \times S^3$ and so, instead of performing a simple linear interpolation between start and goal configuration, we use a linear interpolation (lerp) for the translational part and a spherical linear interpolation (slerp)[4] for the rotational $S^3$ part:

$$lerp(t, a, b) := (1 - t) \cdot a + t \cdot b \text{ and}$$
$$slerp(t, u, v) := \frac{\sin[(1-t)\Omega]}{\sin \Omega} \cdot u + \frac{\sin[t\Omega]}{\sin \Omega} \cdot v,$$

where $a, b \in \mathbb{R}^3$, $u, v \in S^3$ and $\Omega$ is the angle subtended by the arc, so that $\cos \Omega = u \cdot v$.

The routine approach in Alg. 8 shows how we use (spherical) linear interpolation to move the start configuration one step closer towards the goal configuration. It makes

[4] Slerp interpolation for quaternions was proposed by Shoemake in [47]. The interested reader may consult the original work for further information.

**Algorithm 8** APPROACH

```
Configuration approach(
    Configuration start [in],
    Configuration goal  [in] )
{
    double dt = distance( start.t, goal.t );
    double dq = distance( start.q, goal.q );

    tq = clamp( c_φ/dq, [0,1] );
    tt = clamp( c_t/dt, [0,1] );

    t = min( tt, tq );
    inter.q = slerp( t, start.q, goal.q );
    inter.t =  lerp( t, start.t, goal.t );

    return inter;
}
```

use of two different metrics for the translational and rotational part of the configuration space. For the translational part, we use the Euclidean distance in $\mathbb{R}^3$ and we use the orthodromic distance between two points in $S^3$. The step size for interpolation is limited according to the translational and rotational motion bounds $c_t$ and $c_\varphi$. This makes it easy to give an error bound for the motion of the robot between two consecutive time steps. We assume that the robot moves with a constant velocity $v$ and angular velocity $\omega$ between two consecutive discrete positions. The magnitude of these quantities is limited to the aforementioned motion bounds: $|v| \leq c_t$ and $|\omega| \leq c_\varphi$. For any point $p(t) := c(t) + R(t) \cdot p_0$ of the robot, we can then easily bound the maximal distance $l$ that this point can travel in between two consecutive time steps:

$$
\begin{aligned}
l &= \int_0^1 |\dot{p}(t)| \,\mathrm{d}t \\
&= \int_0^1 \left| \frac{\mathrm{d}}{\mathrm{d}t}(c(t) + R(t) \cdot p_0) \right| \mathrm{d}t \\
&= \int_0^1 |v + \omega \times R(t) \cdot p_0| \,\mathrm{d}t \\
&\leq |v| + \int_0^1 |\omega \times R(t) \cdot p_0| \,\mathrm{d}t \quad {}^{(5)} \\
&= |v| + |\omega \times R(0) \cdot p_0| \\
&= |v| + |\omega \times p_0|.
\end{aligned}
$$

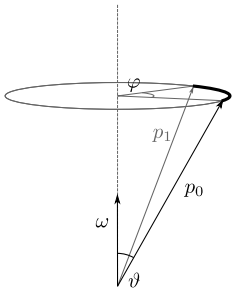


Fig. 3.5: $|\omega \times R \cdot p_0|$ is const for $t \in [0, 1]$.

Thus, we can bound the maximal distance $\mu$ that *any* point of $\mathcal{R}$ can travel in between

[5] Note that $|\omega \times R(t) \cdot p_0|$ is constant for $t \in [0, 1]$ because $|\omega|$, $|R(t) \cdot p_0|$ and the angle $\vartheta = \measuredangle(\omega, R(t) \cdot p_0)$ are constant (cf. Fig. 3.5).

**Algorithm 9** CONNECTLINEAR

```
bool connectLinear(
    Configuration start [in,out],
    Configuration goal  [in] )
{
    for( int i=0; i<i_max; i++ )
    {
        inter = approach( start, goal );

        if( collision( inter, R, O ) == true ) break;
        else start = inter;
    }

    return( i>0 );
}
```

two time steps by

$$\begin{aligned} \mu &\leq |v| + \max_{p\in\mathcal{R}} |\omega \times p| \\ &\leq |v| + |\omega| \cdot \max_{p\in\mathcal{R}} |p| \qquad (\star) \\ &= c_t + c_\varphi \cdot \max_{p\in\mathcal{R}} |p|. \end{aligned}$$

This expression allows us to choose an adequate translational and rotational step size for a given global error bound $\mu$. However, since this is not a tight bound, the approximation is usually very pessimistic and we have to use very small step sizes to guarantee a small error bound.[6]

A basic, yet commonly used local planning approach would now repeatedly approach the goal configuration and perform a collision test for every intermediate position. Alg. 9 gives pseudocode for such a "straight line local planner". In this implementation, we iteratively follow the (spherical) linear interpolation between start and goal with a fixed step size until we determine a collision or run out a maximum number of steps $i_{max}$. The routine returns *true*, if we could perform at least one successful step towards the goal. In this case, the parameter *start* holds the last collision free sample that was found along the path.

The structure of our retraction-based local planner is very similar to this approach. But instead of performing a collision test for a newly generated sample on the path, we try to resolve intermediate penetrations with the resolve routine described earlier. Only if we fail to repair a collision that was caused by a small step towards the goal, we interrupt the advancement. The pseudocode for the connectResolve approach is given in Alg. 10.

[6] We could also use an adequate estimation for every individual motion. In fact, many continuous collision tests use a motion bound according to the distance of robot and obstacle. The bound can be used to repeatedly advance the robot. These techniques are known as *conservative advancement*.



Just like with the connectLinear routine, we return *true* if we could perform at least one successful step. The parameter start always holds the last collision free sample.

**Local Planning and Constrained Optimization:** One interesting idea to look at the problem of local planning from a different point of view is to show its similarities to solving a constrained optimization problem. When we try to connect $q_{start}$ and $q_{goal}$, we cannot expect to find a collision free path. Possibly, we do not even know if $q_{goal}$ is collision free. It would then be desirable to move $q_{start}$ as close as possible to $q_{goal}$ while avoiding any collision with the obstacles. Following this logic, a local planner should aim at solving the optimization problem

$$\min\{\ \delta(q, q_{goal}) \mid q \in \mathcal{C}_{free}\ \},$$

where $\delta$ is some adequate metric in $\mathcal{C}$. We had to start the optimization process at $q_{start}$ and ensure that we can track the minimal argument on a collision free path.
From this point of view, connectResolve can be interpreted as a rudimentary hill climbing optimization algorithm. It minimizes the distance between start and goal configuration along the straight line that connects both configurations. And it maintains the non-collision constraint by using the resolve mechanism to push conflicting configurations back to $\mathcal{C}_{free}$.

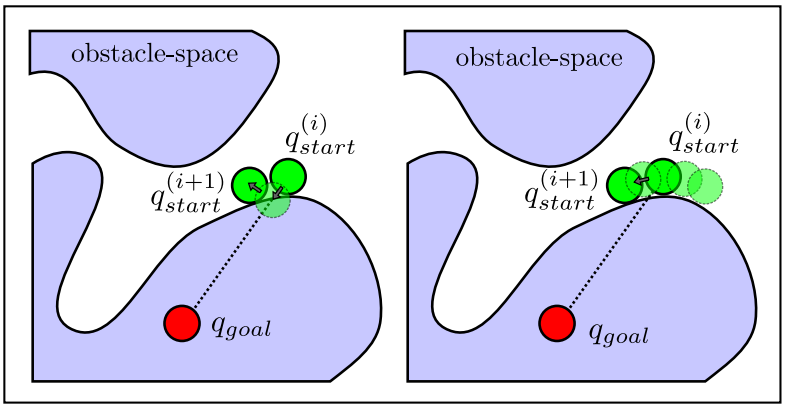


Fig. 3.6: Our retraction-based resolve mechanism (left) vs. optimization based approach by Zhang *et al.* (right). Our approach allows small temporal interpenetrations and tries to push the conflicting positions into free space. Zhang *et al.* [61] sample a local representation of the contact space and determine the sample that minimizes the distance to $q_{goal}$.

Recently, Zhang *et al.* [61] have proposed an optimization-based retraction mechanism that is based on this idea. They first attempt to connect start and goal along a straight line. If this fails, they determine the last collision free sample on this line. Based on an analysis of the contact points in this position, they compute a local approximation of the free-space border at this point [44]. They then generate samples on this local approximation and determine the collision free sample that minimizes the distance to $q_{goal}$ (see Fig. 3.6). This process is iterated to construct a local path.

**Algorithm 10** CONNECTRESOLVE

```
bool connectResolve(
    Configuration start [in,out],
    Configuration goal  [in] )
{
    for( int i=0; i<i_max; i++ )
    {
        n=0;
        inter = approach( start, goal );
        while( resolve( inter, R, O ) && n < n_max ) { n++; }

        if( collision( inter, R, O ) == true ) break;
        else start = inter;
    }

    return( i>0 );
}
```

Compared to this approach, our method is very straightforward and easy to implement. It requires no analysis of contact points to construct a local contact space approximation. It relies only on the resolve mechanism that, as we have seen, can be implemented as a straight forward extension of many existing collision tests.

We note that this technique cannot guarantee that a local path is completely collision free. It is possible for points of the robot to "tunnel" small distances through the obstacle (Fig. 3.7). We could attempt to estimate the maximal error that can occur as a function of the step sizes used for interpolation and retraction as we have sketched on page 91 ($\star$). However, these estimations would probably be too pessimistic in many cases and in order to *guarantee* small object interpenetrations, we had to use very small step sizes. In practice, we found that even with larger step sizes the errors that occur in the planning process are typically much smaller than a conservative estimation would imply. We therefore conceptually allow this source of error in the first planning phase, as allowing small interpenetrations in the local planning phase slightly dilates the free space and can help planning a path through narrow passages. We will then repair and validate the path in a second planning phase (cf. Sec. 3.8).

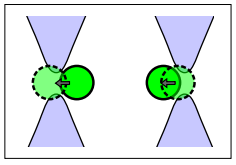

Fig. 3.7: By using the retraction-based resolve mechanism, the robot can possibly tunnel through small corridors.

## 3.5 Expansive Space Trees

There are many different types of sampling-based motion planning algorithms that have been developed in the last decades. Amongst all these techniques, Probabilistic Roadmaps (PRMs) [26] and Rapidly-exploring Random Trees (RRTs) [23] have been successfully used to solve a wide range of different motion planning problems. They have probably become the most famous motion planning algorithms. One problem of the original methods is that they do not use workspace information to guide the sampling strategy or local planning phase which leads to the narrow passage problem as we



have already motivated in the introduction of this chapter.
A third, less popular class of sampling-based motion planners is know as Expansive Space Trees (ESTs) as first described by Hsu *et al.* [15]. ESTs weight the samples that have already been produced with the sampling density in this region. Not only does this lead to a uniform distribution of all samples in unconstrained free space regions, but it can also help to sample difficult regions of the free space more densely than regions that are easily accessible (cf. Fig. 3.8). This makes them an ideal candidate for our application with highly constrained motion planning problems.

The basic EST algorithm iteratively grows a tree $\mathcal{T}$ in free space that is rooted at a given start configuration. Every node $q$ in the tree is assigned a weight $w(q)$ that counts how many other nodes are present in a certain neighborhood $\mathcal{N}(q)$:

$$w(q) := |\, V(\mathcal{T}) \cap \mathcal{N}(q) \,|\,,\ q \in V(\mathcal{T}),$$

where $V(\mathcal{T})$ denotes the set of all nodes in $\mathcal{T}$. At every iteration, the standard EST algorithm picks a node $q_{start}$ from $V(\mathcal{T})$ with a probability proportional to $1/w(q_{start})$. It then attempts to connect this node to a random sample $q_{rand}$ with a collision free local path. The sample $q_{rand}$ is randomly chosen from a (probably different) neighborhood $\mathcal{N}_2(q_{start})$. Hsu *et al.* argue that in expansive[7] configuration spaces, the tree will eventually converge to a uniform sampling of the free space [15].
We use a slightly modified version of this scheme to let the algorithm explore the free space more greedily. Instead of choosing a new node with a probability that is reciprocally proportional to the weight of the node, our planner attempts to grow the $L$ nodes with the smallest $w$-values at every iteration. The value of $L$ is a parameter of our approach. The expansion of these $L$ nodes can be easily distributed over multiple CPU cores. Very small values for $L$ result in a poor utilization of the multiple CPU cores and slows down the parallelization of the local planning. For large values, the algorithm tends to generate lots of samples in the EST which also slows down the exploration of the free space. We will discuss the choice of this parameter in more detail in Sec. 3.9.1.

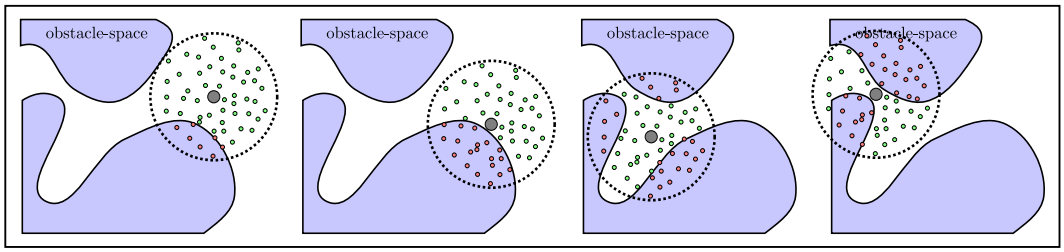


Fig. 3.8: Basic idea of the EST weight function: In order to rate the gray sample, we count the number of samples in a certain neighborhood. If the neighborhood is almost completely in $\mathcal{C}_{free}$, it will contain many free samples. It contains less free samples, if the gray sample is near the border of the free space, in a highly constrained region or inside a narrow corridor.

[7] The term *expansive* is defined in their original work. We do not need this term in the present work and do not go into further detail, here.

**Two different regions:** In addition to the value of $L$, we also have to describe two regions $\mathcal{N}(q)$ and $\mathcal{N}_2(q)$ that define a neighborhood of a point $q$ in the configuration space. Clearly, these regions influence the rating and local exploration behavior of the EST algorithm and are also important parameters.

For a point $q_0 = (c_0, u_0) \in \mathbb{R}^3 \times \mathbb{H}$ we define

$$\mathcal{N}(q_0) := \left\{ (c, u) \in \mathcal{C} \;\middle|\; |c - c_0| \leq r_c \text{ and } |u \mp u_0| \leq r_q \right\}.$$

Here, $r_c$ and $r_u$ define two different radii for the translational and rotational part of a configuration. The two different signs ($\mp$) for the quaternion part of a configuration reflects the fact that two antipodal points of $S^3$ represent the same orientation. Hence, it holds that

$$\mathcal{N}(q_0) = [B_{r_c}(c) \times (B_{r_u}(u) \cup B_{r_u}(-u))] \;\cap\; \mathcal{C}$$

is the product of a ball $B_{r_c}(c)$ in $\mathbb{R}^3$ and the union of two balls in $\mathbb{R}^4$. The definition of $\mathcal{N}(q_0)$ makes it easy to determine if a point belongs to this region or not. Moreover, we can easily utilize a kd-tree as an acceleration structure to find all points $q \in V(\mathcal{T}) \cap \mathcal{N}(q_0)$. This allows an efficient implementation of the weight function $w$. In our implementation, we use a 7d-tree to efficiently perform the required range queries in the configuration space.
To generate random samples for the local exploration of a configuration $q$, we use the same type of region $\mathcal{N}_2$ but with comparatively large radii.

**Parallelization:** In the local planning phase of the algorithm, numerous collision tests have to be performed. Performing a huge number of collision tests is a comparatively expensive operation and so a big fraction of the running time is spent on local planning. The $L$ individual calls to the local planning routine in every iteration of the procedure are completely independent which makes them an ideal candidate for parallel execution. According to the experiences we have made in Chap. 2 when designing parallel collision tests, we can expect a good speed-up by distributing the moderate number of parallel local planning tasks among all available CPU cores. The weighting of the individual nodes in the tree and the associated range queries happen in a serial branch of our code.

**Termination:** As we have mentioned earlier, we define the set of goal positions implicitly by all configurations that separate the bounding boxes of robot and obstacle. Thus, we do not guide the planning by giving a dedicated goal position. To prevent the algorithm from running forever when it cannot find a solution, we limit the maximal execution time and the number of generated samples in the tree.

## 3.6 Collision Detection and Shrinking the Robot

According to our preparatory work on collision detection in Chap. 2, it would make sense to use either bounding volume hierarchies or our extended voxmap pointshell algorithm



(grid-based approach) to perform the collision tests that occur in the local planning phase. Both approaches have shown to scale well with parallelization over multiple CPU cores and both approaches can be extended to an implementation of the resolve algorithm.
One advantage of representing the objects as a union of voxel cells is that we can slightly shrink the robot by removing one layer of voxels. Thus, we can realize a non-conservative approximate collision test. Not only does this improve the efficiency of collision testing (cf. Sec. 2.9.8), but shrinking the robot also results in a dilation of the free space which can help planning paths through narrow passages. This is why we decided to use the grid-based approach to implement the collision checking / resolve mechanism for our local planner.

## 3.7 Non-conservative Retraction-based EST

We have combined the ideas described above to a non-conservative retraction-based EST planner as sketched in Alg. 11. The routine grows a tree $\mathcal{T}$ of rated nodes. It is initialized with a predefined start position. For every node $q$ that is added to $\mathcal{T}$, we update the rating of all nodes in $V(\mathcal{T}) \cap \mathcal{N}(q)$. The evaluation of the weight function $w$ is implemented using a kd-tree on $\mathbb{R}^7 \supset \mathbb{R}^3 \times \mathbb{H}$.
At every iteration of the main loop, we extract the $L$ nodes with smallest weights, $q_{start}[i]$. For all these $L$ nodes, we randomly choose a configuration $q_{rand}[i] \in \mathcal{N}_2(q_{start}[i])$. We then try to find local paths between these pairs of configurations by invoking the connectResolve local planner. The $L$ local planning tasks are completely independent and can be easily distributed over multiple CPU cores. Eventually, all configurations that could be successfully moved towards their goal configurations are added to $\mathcal{T}$. The process terminates when we happen to find a new configuration that separates the bounding boxes of robot and obstacle.

If we are using a grid-based collision test together with watertight models for robot and obstacle, we can optionally dilate the free space by slightly shrinking the robot (or the obstacle or even both) before the planning process and using approximate collision tests. But even if we do not shrink or use approximate tests, the connectResolve routine can cuase tunneling effects in the planning process. Both concepts cause a non-conservative behavior of the planning algorithm. But they also help planning paths in highly constrained environments and narrow passages. This is why we conceptually allow non-conservativeness together with our retraction-based EST algorithm.
Of course, if the planner grows a tree that contains a goal position, we cannot expect the corresponding path $\mathcal{P}$ that connects the start and the goal configuration to be entirely collision free. In the next section, we describe how we repair the path by locally regrowing a new tree.

**Algorithm 11** REST PLANNER

```
Tree growREST( Configuration start [in] )
{
   Tree T;
   T.insert( (start,1) );

   do
   {
      int N = max( T.size(), L );
      q_start[0..N-1] = get_N_best( T );
      for(int i=0; i < N; i++)
         q_rand[i] = randomConfig( N_2(q_start[i]) );

      bool valid[0..N-1] = false;
      for(int i=0; i < N; i++) in parallel {
         valid[i] = connectResolve( q_start[i], q_rand[i] );
      }

      for(int i=0; i < N; i++)
      {
         if( valid[i] )
         {
            T.insert( q_start[i], w(q_start[i]) );
            updateRating( T, N(q_start[i]) );
         }
      }
   }
   while ( time < t_max && T.size() < n_max &&
      {OBBs of robot and obstacle are not disjoint} )

   return T;
}
```



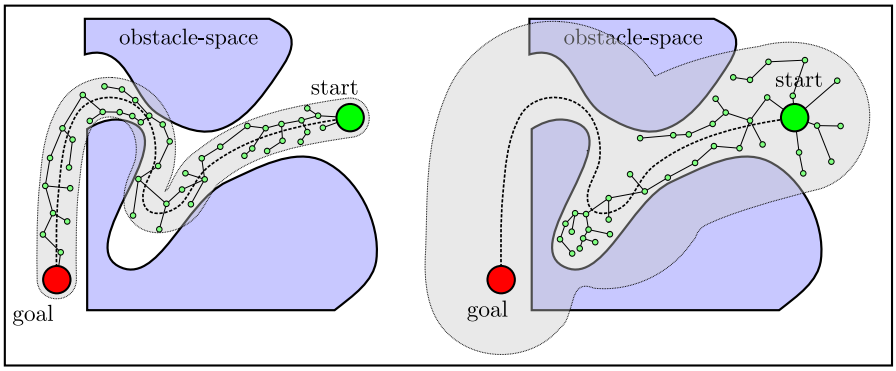


Fig. 3.9: The figure shows how we grow an LBRRT in a neighborhood of a path from start to goal. If the size of the neighborhood is too big, the algorithm can suffer from "local minima problems" (right picture).

## 3.8 Local Re-planning of the Solution Path

In this section, we want to assume that we already have computed a path $\mathcal{P}$ that connects start and goal position. $\mathcal{P}$ results from growing a retraction-based EST as described in the previous section. It is not guaranteed to be absolutely collision free, but we want to use this path to guide a second planning phase that uses more accurate collision checking. To achieve this, we assume[8] that a valid solution path exists in some tube shaped region around $\mathcal{P}$. We grow a new tree $\mathcal{T}_{loc}$ rooted at the start position of the path $\mathcal{P}$ in the following manner: The algorithm weights all points $q \in \mathcal{P}$ (not $\mathcal{T}_{loc}$ !) by the number of nodes in $\mathcal{T}_{loc}$, that lie in a certain neighborhood $\mathcal{N}(q)$:

$$v(q) := |\, V(\mathcal{T}_{loc}) \cap \mathcal{N}(q) \,|\,, \; q \in \mathcal{P}.$$

At every iteration, we determine the $K$ points $q_0, \ldots, q_{K-1}$ in $\mathcal{P}$ with the smallest weights for further exploration. We do not want points near the tail of the path to influence the early sampling. So we consider only points with a strictly positive weight. For every such point $q_i$, we choose a random sample $q_{rand}$ in a neighborhood $\mathcal{N}_2(q_i)$. We let $q_{near} \in V(\mathcal{T}_{loc})$ be the nearest neighbor of $q_{rand}$ and try to extend the path along the straight line that connects $q_{near}$ and $q_{rand}$ using the connectLinear local planner described in Alg. 9. In this way, we grow a tree similar to a RRT but we restrict the sampling domain to a tube shaped neighborhood of our first guess solution path $\mathcal{P}$ (cf. Fig. 3.9). We refer to this exploration scheme as Locally Biased RRT (LBRRT). By biasing the sampling with the weight function $v$, we encourage the algorithm to put the same number of samples along the path. Again, this helps to sample the narrow free space-regions more densely than the unconstrained regions. Pseudocode for the LBRRT algorithm is given in Alg. 12. The size of the region $\mathcal{N}_2$ should not be chosen too big for

[8] We do not give proof that this assumption is always correct. In fact this does not have to be the case. However, we could always repair the paths $\mathcal{P}$ from the first planning phase in plase II in all our benchmarks.

**Algorithm 12** LBRRT PLANNER

```
Tree growLBRRT( Path P [in] )
{
   Tree T;
   T.insert( P.start );
   updateRating( P, N(start) );

   do
   {
      q0[0..K-1] = get_K_best( P );
      for(int i=0; i < K; i++)
      {
         q_rand[i] = randomConfig( N2(q0[i]) );
         q_near[i] = T.getNN( q_rand[i] );
      }

      bool valid[0..K-1] = false;
      for(int i=0; i < K; i++) in parallel {
         valid[i] = connectLinear( q_near[i], q_rand[i] ) );
      }

      for(int i=0; i < K; i++)
      {
         if( valid[i] )
         {
            T.insert( q_near[i] );
            updateRating( P, N(q_near[i]) );
         }
      }
   }
   while ( time < t_max && T.size() < n_max &&
      {OBBs of robot and obstacle are not disjoint} )

   return T;
}
```

this algorithm. For one thing, we want to have a rather small tube around $\mathcal{P}$ in order to restrict the search space, for another thing, we found that very big diameters of $\mathcal{N}_2$ (and $\mathcal{N}$) can cause the algorithm to be stuck in local minima as is shown at Fig. 3.9 on the right. In the indicated case, the algorithm attempts to reach the goal configuration too early and does not consider to take the detour around the obstacles. We handle this by careful choice of the parameters $\mathcal{N}$ and $\mathcal{N}_2$. In case the algorithm gets stuck in a dead end, we simply restart the algorithm. See Sec. 3.9.6 for an analysis of phase II.

## 3.9 The Benchmarks

We have implemented our algorithm and tested it on a Core i7 Intel Xeon 4-core CPU with 2.4GHz. To highlight the strength of our approach, we benchmarked it in a number



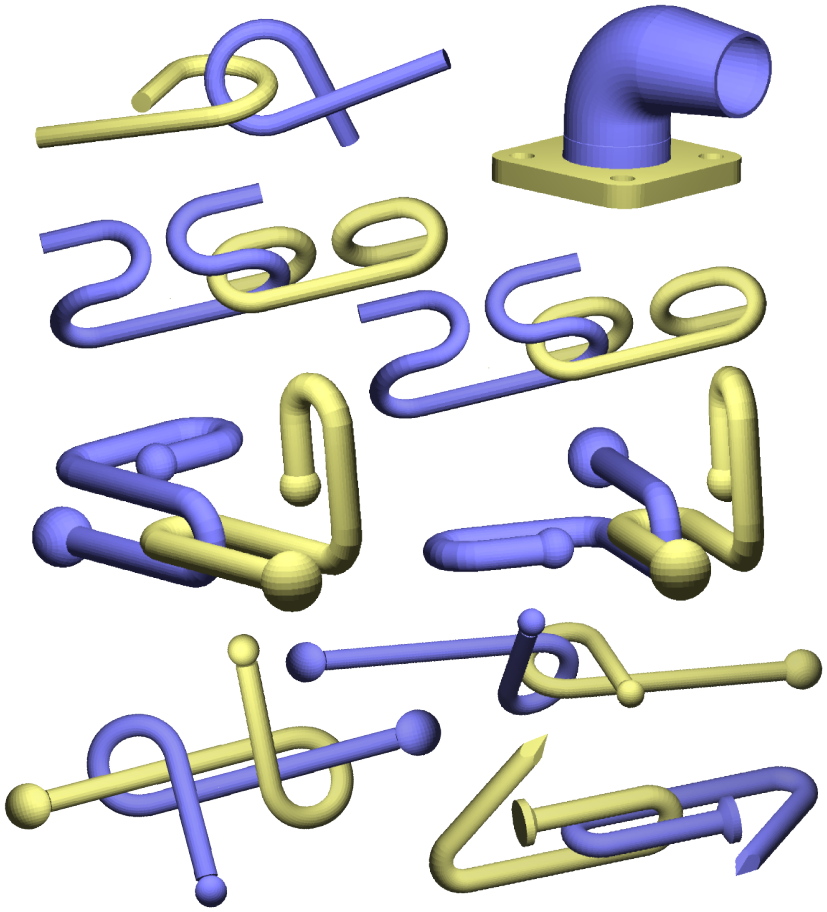

Fig. 3.10: The different puzzles used in our benchmarks (top left to bottom right): Alpha1.0, Flange, XPuzzle225, XPuzzle210, TwistedArcs(a)/(b), AlphaNails(a), AlphaNails(b) and Nails. We show the obstacle (yellow) and robot in start position (blue).

of very difficult and highly constrained disassembly problems with many narrow passages (see Fig. 3.10). In addition to the well known Alpha1.0 and the Flange benchmark scenarios, we have modeled and tested a number of difficult new metal puzzles.

The complexity of many motion planning problems can change significantly with the specified start and goal positions of the robot. For example, if we put the start position near the narrow passage of the Alpha1.0 puzzle and the goal position is on the other side of that passage, a bi-directional tree planner can perform much better than in an arbitrary case. To account for this, we use start positions for our benchmarks that are not in close proximity to the narrow passages (cf. Fig. 3.10). We do not use a dedicated goal configuration to guide the planning process but the objective is to separate the bounding boxes of robot and obstacle. This makes the planning process more difficult, as no bi-directional approaches or biasing towards a goal position can be applied. But

| | Alpha1.0 | Nails |
|---|---|---|
| L = 32 | 9.59 / 2.51 | 33.35 / 12.15 |
| L = 64 | 8.57 / 2.04 | 30.81 / 10.12 |
| L = 128 | 9.25 / 2.31 | 30.34 / 9.78 |
| L = 256 | 9.50 / 2.27 | 28.10 / 7.20 |
| L = 512 | 9.80 / 1.79 | 31.41 / 9.57 |
| L = 1024 | 14.38 / 1.60 | 39.39 / 9.66 |

Table 3.1: Running times for different values of the parallel growth width L. Mean value / standard deviation over 50 runs in seconds.

it also makes it more general and applicable in cases, where we do not know a good goal position in advance.

### 3.9.1 Parameter Analysis: Phase I

We now want to analyze the behavior of our algorithm for different choices of parameters. The two main parameters of our approach are the number of nodes $L$ that are concurrently expanded at every iteration of the main loop and the size of the neighborhood $\mathcal{N}$ that is used for the evaluation of the weight function $w$. We want to examine the first and the second phase separately. Let us first have a look at phase I.
All timings in this section are taken on a Core i7 Intel Xeon 4-core CPU with 2.4GHz. For all tests, we shrank the robot by one voxel layer and used approximate collision tests. The influence of parallelization and approximate collision checking will be discussed later.

**Parallel Growth Size $L$.**
Table 3.1 shows the running times and standard deviation of the growREST algorithm for different sizes of $L$ for the Alpha1.0 and the Nails benchmark. For all values of $L$ we use $\mathcal{N}(c, u) = B_{7.0}(c) \times B_{0.1}(\pm u)$. The timings are given in seconds.
We can see that in both scenarios, the number of nodes that are concurrently expanded in the EST affects the running time of the algorithm. The qualitative behavior for both scenarios shows that as $L$ decreases, the running time starts to decrease as well. For too small values of $L$ however, the algorithm starts to consume more time again.

If we compare the running times to the average number of samples created in the EST, as shown in Table 3.2, we can see that regardless of the running times, the number of configurations always decreases with the value of $L$. Clearly, a very high value of the parameter $L$ leads to a broader exploration strategy and a higher number of configurations being created in the EST in total. On the other hand, from what we have learned in Chap. 2, a small number of concurrent tests leads to a smaller throughput of parallel tasks on the CPU. This means that even though we have to perform fewer local planning



| | Alpha1.0 | Nails |
|---|---|---|
| L = 32 | 55,443 / 12,139 | 154,232 / 40,074 |
| L = 64 | 57,436 / 11,750 | 160,534 / 39,931 |
| L = 128 | 63,574 / 12,610 | 173,919 / 43,771 |
| L = 256 | 68,820 / 12,689 | 175,259 / 35,618 |
| L = 512 | 80,416 / 11,411 | 199,591 / 44,019 |
| L = 1024 | 116,982 / 10,525 | 253,076 / 46,248 |

Table 3.2: Number of configurations for different values of the parallel growth width L. Mean value / standard deviation over 50 runs in seconds.

tasks and thus fewer collision tests for $L = 32$ then for $L = 64$ or $L = 128$, we need more overall planning time because of the smaller utilization of the parallel CPU cores.

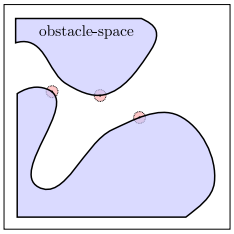


Fig. 3.11: Very small regions $\mathcal{N}$ do not help to identify narrow passages.

**Size of the Neighborhood $\mathcal{N}$.**
The size of the neighborhood can also affect the behavior of the planning algorithm because it influences the weight function $w$. Intuitively, the size of $\mathcal{N}$ should somehow be related to the tightness of the narrow passages that we want to classify. Too small regions cannot differentiate between a narrow corridor and any other place near the free space boundary (cf. Fig. 3.11). On the other hand, if we use too large regions to evaluate $w$, we cannot see structures that are relatively small. Additionally, performing range queries with very large regions can become expensive as the size of the output typically adds to the running time of the query.
In our benchmarks, we used regions $\mathcal{N}$ of the form

$$\mathcal{N}(c, u) := B_{r_0}(c) \times B_{r_1}(\pm u)$$

for different radii $r_0$ and $r_1$. We let $r_0$ and $r_1$ vary from 3.5 to 14.0 and from 0.1 to 0.4 respectively and recorded the running times and standard deviations over 50 runs for the Alpha1.0 and the Nails puzzle in Table 3.3. To get an impression of the size of the different regions, Fig. 3.12 shows the swept volumes that are formed when the robot nail moves in $\mathcal{N}$. We timed out the construction of the EST after 120 seconds for the Alpha1.0 benchmark and after 180 seconds for the nails benchmark.
For varying size of $\mathcal{N}$, the growREST algorithm was not always able to successfully reach a goal configuration within the predefined time limit. The settings for which this was the case are marked with an asterisk in Table 3.3. Table 3.4 shows the number of successful planning attempts for the different sizes of $\mathcal{N}$. We can see that a small neighborhood has a negative effect on the overall planning time of the algorithm. But it does not affect the ability of the algorithm to find a valid goal configuration. For increasing sizes of $\mathcal{N}$, the performance first starts to improve before it gets worse again. This is because for very large neighborhoods the algorithm has a smaller success rate. As all failed planning attempts add to the average running time with $t_{max}$ (which is 120s or 180s for the Alpha1.0 and Nails benchmark, respectively), this has a significant negative effect. Interestingly, if we ignore the unsuccessful queries and take the mean only over the successful queries,

| **Alpha1.0** | 3.5 | 7.0 | 10.5 | 14.0 |
|---|---|---|---|---|
| 0.1 | 42.12 / 10.27 | 8.44 / 1.83 | 5.32 / 1.06 | 6.10 / 1.72 |
| 0.2 | 13.06 / 3.34 | 6.10 / 2.01 | 4.95 / 1.44 | 6.50 / 5.14 |
| 0.3 | 8.74 / 2.53 | 6.61 / 3.29 | 17.61 / 28.62* | 58.27 / 52.61* |
| 0.4 | 13.23 / 9.80 | 21.29 / 34.04* | 91.70 / 47.93* | 108.43 / 34.76* |

| **Nails** | 3.5 | 7.0 | 10.5 | 14.0 |
|---|---|---|---|---|
| 0.1 | 54.34 / 19.80 | 26.88 / 10.13 | 77.79 / 66.47* | 126.60 / 69.63* |
| 0.2 | 57.86 / 37.38* | 28.38 / 31.78* | 51.78 / 62.08* | 101.97 / 79.11* |
| 0.3 | 78.32 / 57.38* | 48.99 / 57.75* | 61.97 / 69.83* | 96.66 / 78.89* |
| 0.4 | 106.20 / 66.81* | 112.38 / 73.33* | 102.53 / 77.72* | 111.93 / 74.85* |

Table 3.3: Timings for different sizes of the C-space region $\mathcal{N}$. L is set to 128. Mean value / standard deviation over 50 runs in seconds. For the cells marked with an *, not all planning queries could successfully reach a goal configuration and timed out. They add with 120s (180s) to the results of the Alpha1.0 (Nails) benchmark.

the running times are not very much affected (see timings in Table 3.5).

Roughly condensed we can say that the choice of the parameter $\mathcal{N}$ is critical for the performance of the algorithm. Very small values lead to high success rates but can also lead to high running times. Larger regions lead to better running times but can have a negative effect on the success rate of the algorithm. This is because if $\mathcal{N}$ is too big, we cannot see small structures like narrow passages in the configuration space. If these passages are not being traversed in an early planning phase, they vanish in the mass of densely distributed configurations and can hardly be recognized by the algorithm later on. This parameter should be carefully chosen to fit the planning problem. If the success rate of the algorithm goes down, $\mathcal{N}$ is probably too big. The motion planning puzzles we have used as benchmark scenarios all have more or less the same size. In the following tests, we have always used $(r_0, r_1) = (7.0, 0.1)$. As indicated by our experiments with the Alpha1.0 and the Nails puzzle, this seems to be an adequate size and it worked out fine for the other puzzles as well.



| **Alpha1.0** | 3.5 | 7.0 | 10.5 | 14.0 |
|---|---|---|---|---|
| 0.1 | 50 | 50 | 50 | 50 |
| 0.2 | 50 | 50 | 50 | 50 |
| 0.3 | 50 | 50 | 47 | 30 |
| 0.4 | 50 | 45 | 13 | 5 |

| **Nails** | 3.5 | 7.0 | 10.5 | 14.0 |
|---|---|---|---|---|
| 0.1 | 50 | 50 | 37 | 19 |
| 0.2 | 48 | 49 | 42 | 25 |
| 0.3 | 41 | 43 | 38 | 28 |
| 0.4 | 30 | 26 | 26 | 24 |

Table 3.4: Number of successful planning queries for different sizes of the C-space region $\mathcal{N}$. L is set to 128 and we have performed 50 queries in total.

| **Alpha1.0** | 3.5 | 7.0 | 10.5 | 14.0 |
|---|---|---|---|---|
| 0.1 | 42.12 / 10.27 | 8.44 / 1.83 | 5.32 / 1.06 | 6.10 / 1.72 |
| 0.2 | 13.06 / 3.34 | 6.10 / 2.01 | 4.95 / 1.44 | 6.50 / 5.14 |
| 0.3 | 8.74 / 2.53 | 6.61 / 3.29 | 11.08 / 12.64* | 17.10 / 19.40* |
| 0.4 | 13.23 / 9.80 | 10.32 / 9.17* | 11.08 / 7.27* | 4.14 / 0.79* |

| **Nails** | 3.5 | 7.0 | 10.5 | 14.0 |
|---|---|---|---|---|
| 0.1 | 54.34 / 19.80 | 26.88 / 10.13 | 41.87 / 31.78* | 39.47 / 22.68* |
| 0.2 | 52.77 / 28.42* | 25.28 / 23.49* | 27.35 / 29.32* | 23.93 / 18.42* |
| 0.3 | 56.00 / 35.32* | 27.66 / 25.08* | 24.70 / 25.04* | 31.18 / 36.98* |
| 0.4 | 57.00 / 37.24* | 49.94 / 47.11* | 31.01 / 30.01* | 38.18 / 34.81* |

Table 3.5: Timings for different sizes of the C-space region $\mathcal{N}$. L is set to 128. Mean value / standard deviation over 50 runs in seconds. For the cells marked with an *, some planning queries timed out and have been ignored in the results shown.

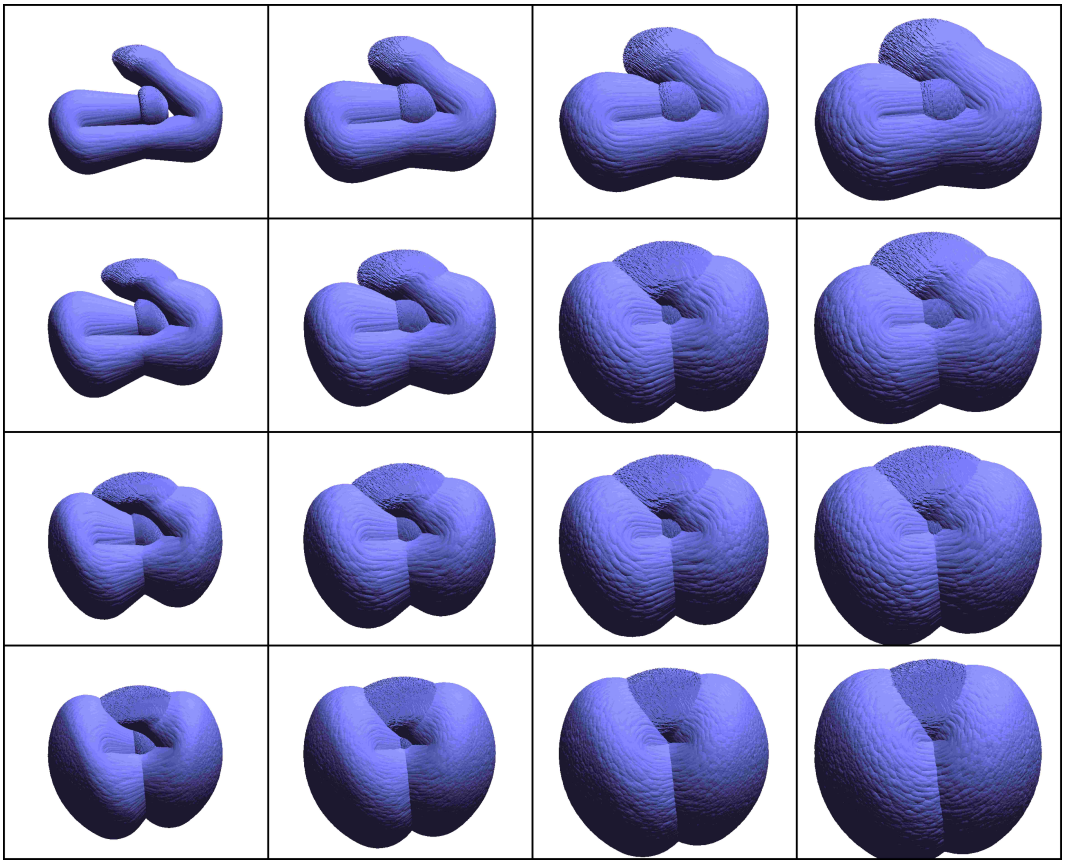

Fig. 3.12: Swept volumes of the nail moving in differently sized regions $\mathcal{N}$. The swept volumes gives an impression of the different sizes of $\mathcal{N}$ used in tables 3.3, 3.4 and 3.5. From left to right, $r_0 = 3.5, \ldots, 14.0$; from top to bottom, $r_1 = 0.1, \ldots, 0.4$.



| | 1 - Core | | | 4 - Core | | |
|---|---|---|---|---|---|---|
| | Local Planning | Rest | Total | Local Planning | Rest | Total |
| Alpha1.0 | 42.22 | 0.79 | 43.0 | 8.39 | 0.86 | 9.25 |
| Nails | 131.79 | 2.62 | 134.41 | 26.78 | 3.56 | 30.34 |

Table 3.6: Parallelization: The tables compare the same code running on just a single CPU core and 4 cores with hyper-threading. Mean value over 50 runs in seconds.

### 3.9.2 Parallelization

As we have described in earlier, our algorithm can easily be parallelized by distributing the independent local planning tasks that result from the concurrent expansion of the best nodes in our EST over multiple CPU cores.
It is worth mentioning that in this manner, we only parallelize a certain fraction of the main loop of our algorithm. The sample generation, the range queries to evaluate our weight function $w$, the insertion of new nodes to our tree and the check of the termination criterion happen in a serial branch of the code. According to Amdhal's Law, this can have severe implications on the maximal achievable speed-up if the serial part of the code consumes a considerable fraction of the running time.

Table 3.6 shows the running times for the Alpha1.0 and Nails benchmark when the algorithm runs on just a single CPU core and compares it to a quad-core version. The tables show a breakdown of the overall planning time into local planning and the serial part of the algorithm. In both scenarios, the local planning consumes more than 98% of the running time.
This motivates why we have decided to parallelize only the local planning part of the code. The individual local planning task are completely independent, which allows to distribute them to different cores with very little implementation overhead. On the other hand, to parallelize read-write operations on the kd-tree we used to implement the evaluation of the weight function $w$ is a highly non trivial task. Luckily, as this part of the code accounts for barely 2% of the overall running time, there is not much to be gained from a parallel implementation.
The running times were taken for parameter $L = 128$. In both scenarios, we could achieve a 5 times speed-up of the local planning phase compared to the single-core version. This leads to an overall speed-up factor of around 4.5 for phase I of our planning algorithm.

### 3.9.3 Why Not Use Standard EST?: Greedy EST vs. Standard EST.

The standard EST algorithm as originally proposed by Hsu *et al.* [15] picks a node for exploration with a probability that is proportional to $1/w$ for a weight function that is defined similarly to the function $w$ in Sec. 3.5. Opposed to this, we have decided to always use the best rated nodes for exploration. We could have chosen $L$ nodes

| | EST $\sim 1/w$ | EST $\sim 1/w^2$ | EST $\sim 1/w^3$ | Greedy EST |
|---|---|---|---|---|
| Alpha1.0 | $> 120^*$ / - | 36.55 / 3.92 | 20.52 / 2.69 | 8.39 / 2.07 |
| Nails | $> 140^*$ / - | 95.23 / 18.11 | 59.66 / 16.69 | 26.78 / 8.60 |

Table 3.7: Running times of the standard EST algorithm for Phase I with shrink. We used L=128. Timings are given for different distribution functions. For the first, second and third column of the table, the algorithms picks a sample for expansion with a probability that is proportional to $1/w$, $1/w^2$ or $1/w^3$, respectively. Timings show *local planning time only*. Mean value / standard deviation over 50 runs in seconds. For cells marked with a $^*$, the algorithm could not find a solution within the restricted tree size of one million configurations.

for concurrent exploration according to the originally proposed distribution function just as well. However, we found that in our scenarios with many narrow passages, our greedy, deterministic exploration scheme could perform better than the standard EST approach. To quantify this, we used a reference implementation that picks $L$ random samples at every iteration according to different probability distributions and compared the results to our Greedy EST planner. To mask out the effect of the sample generation and evaluation of $w$, we compare only the time spent for local planning. As the different algorithms all use the exact same local planning mechanism, comparing the local planning time needed per query gives information of the abilities of the global planning mechanism. Table 3.7 shows the results of this experiment for the Alpha1.0 and Nails benchmarks. For all tests, we used $L = 128$ as parallel growth width. We performed 20 individual runs and present the mean and standard deviation of the running times we have measured. When using a probability distribution function that picks every node in the EST with a probability $\sim 1/w$, the algorithm was killed after it had created more than one million configurations in the tree. By this time, it had spent more than 120s for *local planning only*. It was never able to find a solution path within the tree size limit. To promote a greedier behavior of the algorithm, we used distributions $\sim 1/w^2$ and $\sim 1/w^3$ to pick better samples with higher probability. Following this approach, the algorithm was able to find a solution for the Alpha1.0 puzzle in 36.55 seconds and 20.52 seconds average local planning time. The Greedy EST consumed only 8.39 seconds of local planning time on average. In other words, the greedy exploration scheme of always picking the $L$ best nodes for exploration was 2 to 4 times as efficient as choosing these nodes probabilistically. We have made similar observations for the nails benchmark as well.

### 3.9.4 Shrinking the Robot

So far, we always used a shrunken robot and approximate collision tests. How does not shrinking the robot and using accurate collision tests between the triangle sets of the objects influence the performance of the first phase of our motion planner? Table 3.8



| **no shrink** | Local Planning | kd-Tree | Rest | Total | # Configs |
|---|---|---|---|---|---|
| Alpha1.0 | 27.55 | 2.01 | 0.18 | 29.74 | 95586 |
| Nails | 109.29 | 6.84 | 0.92 | 117.05 | 214231 |

| **shrink** | Local Planning | kd-Tree | Rest | Total | # Configs |
|---|---|---|---|---|---|
| Alpha1.0 | 8.39 | 0.77 | 0.09 | 9.25 | 63574 |
| Nails | 26.78 | 3.09 | 0.47 | 30.34 | 173919 |

Table 3.8: Comparison of running times with and without shrinking the robot. The tables show a breakdown of the running time in seconds and the number of nodes in the resulting trees. Mean value / standard deviation over 50 runs in seconds.

compares the running times for the Alpha1.0 and the Nails scenario with and without shrinking the robot.

We see that shrinking the robot before the first planning phase can improve the performance by a factor of $\sim 3$ for the Alpha1.0 and even $\sim 3.8$ for the Nails scenario. If we also take the number of generated samples in the trees into account, we can conclude that this improvement is to some extent caused by the reduced cost of the collision tests: using a shrunken robot in the Nails scenario, for example, consumes roughly 25% of the time but still generates 80% of the samples compared to the "no shrink version". Still, the reduction of generated samples indicates that slightly shrinking the robot can help the algorithm to grow the tree through narrow passages. This is more obvious for the Alpha1.0 benchmark. Here, the "shrink version" of the algorithm consumes around 30% of the running time and produces only 66% of the nodes.

As shrinking the robot dilates the free space, planning with a shrunken robot can affect the quality of the resulting path. In the worst case, it can change the connectivity of thin-walled free space regions and create new passages. So even though this strategy has shown to work well in the first planning phase, it can probably complicate or prevent the successful execution of the second phase. We will examine this in more detail in the next section.

### 3.9.5 Retraction-based Local Planning

In addition to shrinking the robot, we also use a retraction-based local planning technique (connectResolve, Algo. 10) to enhance the local path planning mechanism in narrow passages or otherwise restricted regions of the free configuration space (cf. Sec. 3.4). Like we already said earlier, this can have implications on the quality of the resulting path, as sample retraction influences the distance between two adjacent samples on a local path. Thus, it is probably harder to repair a path in the second planning phase. We will further discuss this problem in Sec. 3.9.7.
The idea behind a retraction-based local planning mechanism is that hopefully, we can

| **no shrink** | Local Planning | kd-Tree | Rest | Total | # Configs |
|---|---|---|---|---|---|
| Alpha1.0 | 169.18 | 155.40 | 34.95 | 359.53 | 1102440 |
| Nails | 562.06 | 229.36 | 195.64 | 987.06 | 1362470 |

| **shrink** | Local Planning | kd-Tree | Rest | Total | # Configs |
|---|---|---|---|---|---|
| Alpha1.0 | 64.99 | 95.70 | 18.28 | 178.97 | 899784 |
| Nails | 100.79 | 116.54 | 62.45 | 279.78 | 1041110 |

Table 3.9: Comparison of running times with and without shrinking the robot *when using a straight line local planner*. The tables also show the number of nodes in the resulting trees. Mean value / standard deviation over 50 runs in seconds.

improve the efficiency of the overall planning algorithm due to a more effective local planner. On the other hand, a simple straight line local planner (like connectLinear shown in Algo. 9) is obviously more efficient and we can check a higher amount of local paths in the same amount of time than with a retraction-based planner. In this section, we have a look at how the choice of the local planner influences the performance of the overall planning algorithm. Table 3.9 shows the running times for the Alpha1.0 and the Nails puzzle when using a straight line local planner.

We can see that shrinking the robot has a positive effect similar to what we have observed in the previous section. It improves the running times for the Alpha1.0 and the Nails puzzle by a factor of $\sim 2$ and $\sim 3.5$, respectively while still generating around 80% of the nodes in our trees. Again, this indicates that not only the dilation of the free space, but also the use of relatively faster, approximate collision tests influences the running time when shrinking the robot. Together, these two factors result in the observed speed-up.

If we compare the running times in table 3.9 with the timings in table 3.8, we see that we can achieve a nice speed-up when using a retraction-based local planner rather than a straight line local planner. When we did not shrink the robot in advance to the planning, we have observed a 12 (8.5) times performance improvement when using the retraction-based planner in our two benchmark scenarios. The factors improved to 19 for the Alpha1.0 and 9.2 for the Nails benchmark for the same test with a shrunken robot. Even though a single call to the retraction-based planner is more expensive than a single call to the linear planner, it was beneficial to use the informed rather than the fast but uninformed local planning mechanism.

The higher efficiency of the simpler local planner also reflects in the breakdown of the running times. When using the straight line local planner, the algorithm has to spend less time for local planning in every cycle. Consequently, the relative cost of performing range queries with our kd-tree, finding the L best rated samples and generating new random samples goes up. As these operations now take a considerable fraction of the overall running time, it could be well worth to think about parallelizing the respective branches of the code. If we don't want to or cannot use our retraction-based planning



mechanism (for example if robot and obstacle don't have meaningful surface normals), this could help to further improve the performance. However, this is beyond the scope of this work and we will not go into further detail about how these operations could be parallelized.

### 3.9.6 Parameter Analysis: Phase II

**Parallel Growth Size $K$.**
Table 3.10 compares the running times of the LBRRT algorithm in the second planning phase for different values of the parallel growth width $K$. The first and the second table show the running times for re-planning the paths produced by the growREST algorithm with approximate and accurate collision tests, respectively. Like we have already seen in the first planning phase, it should be avoided to expand a very high number of nodes concurrently. This results in an increased number of overall samples and does not speed up the planning process. On the other hand, smaller values of $K$ help to reduce the number of overall nodes which has a positive effect on the running time. This is true up to the point where the small number of parallel threads reduces the throughput of the parallel local planning tasks. In Table 3.10 on top, *"Phase I shrink"*, this is the case for $K = 8$ and we have achieved the best running time for $K = 16$. Compared to the first planning phase, this is only a small number of concurrently expanded nodes. The second table shows the results for the re-planning of the paths that have been constructed using exact collision checking in the first planning phase. Compared to the first table, we can achieve an improvement of the running times by a factor of more than 3 and also considerably smaller standard deviations. This is especially true for small values of $K$. We are going to use a value of $K = 32$ for all further benchmarks in this section.

**Size of the Neighborhood $\mathcal{N}_2$.**
Just like in the first planning phase, the LBRRT algorithm introduces a neighborhood $\mathcal{N}$ as a parameter. This region is used to evaluate our weight function $w$. As we have seen in the discussion of phase I, the size of the region should be chosen appropriately in order to properly detect narrow passages and achieve a good performance and success rate of our probabilistic motion planner. For the second phase, we use the region $\mathcal{N}$ from phase I with $r_0 = 7.0$ and $r_1 = 0.1$ as these values have shown to work well for our problems.
In phase I, we used a relatively large region $\mathcal{N}_2$ to generate random samples. But as we want to restrict the sampling domain in the second phase to a small tube around a path $\mathcal{P}$, we want to use a relatively small neighborhood $\mathcal{N}_2$ to generate our samples, here. In this paragraph, we are going to experimentally investigate, how the size of this region affects the performance of our re-planning algorithm. For the growREST algorithm, we have used neighborhood regions of the form

$$\mathcal{N}(c, u) := B_{r_0}(c) \times B_{r_1}(u) \cup B_{r_0}(c) \times B_{r_1}(-u)$$

| **Phase I shrink** | Alpha1.0 | Nails |
|---|---|---|
| K = 8 | 1.04 / 2.15 | 2.11 / 3.14 |
| K = 16 | 1.03 / 1.87 | 1.90 / 2.67 |
| K = 32 | 1.03 / 1.57 | 1.95 / 3.01 |
| K = 64 | 1.09 / 1.34 | 2.04 / 2.95 |
| K = 128 | 1.44 / 1.22 | 2.28 / 2.26 |
| K = 256 | 2.23 / 1.33 | 3.29 / 2.43 |
| K = 512 | 3.71 / 1.32 | 5.08 / 2.21 |

| **Phase I no shrink** | Alpha1.0 | Nails |
|---|---|---|
| K = 8 | 0.29 / 0.22 | 0.53 / 0.50 |
| K = 16 | 0.34 / 0.16 | 0.57 / 0.51 |
| K = 32 | 0.44 / 0.14 | 0.61 / 0.32 |
| K = 64 | 0.64 / 0.13 | 0.83 / 0.25 |
| K = 128 | 1.06 / 0.21 | 1.31 / 0.48 |
| K = 256 | 1.85 / 0.37 | 2.27 / 0.56 |
| K = 512 | 3.38 / 0.75 | 4.34 / 0.84 |

Table 3.10: PHASE II: Running time vs. the parallel growth width K. Mean value / standard deviation over 50 runs in seconds.

as these regions can be used directly for range queries with a kd-tree in a seven dimensional space. We use the same type of regions to sample random configurations in the second planning phase, but for compatibility reasons with our code basis, we have an other convention to specify the radii here. More specifically, for two radii $r_0$ and $\varphi_1$ we define

$$\mathcal{N}_2(c_0, u_0) := \left\{ (c, u) \in \mathcal{C} \;\middle|\; \|c - c_0\| \leq r_0 \text{ and } |u^\top u_0| \geq \cos(\varphi_1) \right\}.$$

The reader should note that $\mathcal{N}_2$ can be written as the product of a sphere in $\mathbb{R}^3$ and a double cone in $\mathbb{R}^4$. But since we already know that we are only dealing with unit quaternions for the $\mathbb{R}^4$-part of a configuration, this can also be written in the same form as $\mathcal{N}$, for a corresponding radius $r_1$. We could easily convert the radius $r_1$ to the aperture $\varphi_1$ of the cone and vice versa (cf. Fig. 3.13). It holds that $r_1/2 = \sin(\varphi_1/2)$.

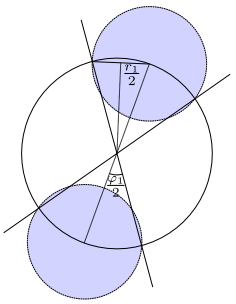


Fig. 3.13: The radius $r_1$ and the aperture $\varphi_1$ of the double cone can be easily converted.

Table 3.11 summarizes the results of our experiments for the Alpha1.0 and the Nails puzzle. For each of the 50 different paths that resulted from the first planning phase, we took 10 attempts to locally re-plan the path in the second phase. In this setup, we timed out the planning process after 20 seconds. The tables show the average running time over the 500 runs and the number of successful runs for different sizes of $\mathcal{N}_2$ (not $\mathcal{N}$!) . We only took the mean of the successful attempts. We can see that, unlike in the discussion of the parameter $\mathcal{N}$ of phase I, too big as well as too small sizes of $\mathcal{N}_2$ influence the success rate of the planning algorithm in phase II. If the regions get small, the algorithm is probably not able to find a path in the induced neighborhood of $\mathcal{P}$.



| **Alpha1.0** | $r_0 = 2.5$ | $r_0 = 5.0$ | $r_0 = 7.5$ | $r_0 = 10.0$ |
|---|---|---|---|---|
| $\varphi_1 = 2.5$ | 1.00 / 1.79 | 1.28 / 2.11 | 1.96 / 2.33 | 3.75 / 3.05 |
| $\varphi_1 = 5.0$ | 0.72 / 0.95 | 0.99 / 1.41 | 1.75 / 2.24 | 3.07 / 2.89 |
| $\varphi_1 = 7.5$ | 0.62 / 0.28 | 0.85 / 1.07 | 1.50 / 1.98 | 2.64 / 2.31 |
| $\varphi_1 = 10.0$ | 0.74 / 0.39 | 0.76 / 0.42 | 1.36 / 1.40 | 2.76 / 2.26 |

| **Alpha1.0** | $r_0 = 2.5$ | $r_0 = 5.0$ | $r_0 = 7.5$ | $r_0 = 10.0$ |
|---|---|---|---|---|
| $\varphi_1 = 2.5$ | 460 | 440 | 421 | 398 |
| $\varphi_1 = 5.0$ | 499 | 488 | 477 | 461 |
| $\varphi_1 = 7.5$ | 500 | 500 | 494 | 485 |
| $\varphi_1 = 10.0$ | 500 | 500 | 499 | 492 |

| **Nails** | $r_0 = 2.5$ | $r_0 = 5.0$ | $r_0 = 7.5$ | $r_0 = 10.0$ |
|---|---|---|---|---|
| $\varphi_1 = 2.5$ | 2.00 / 3.07 | 2.26 / 2.90 | 4.33 / 4.43 | 5.97 / 4.40 |
| $\varphi_1 = 5.0$ | 1.37 / 2.20 | 1.92 / 2.68 | 3.01 / 3.43 | 4.63 / 3.84 |
| $\varphi_1 = 7.5$ | 0.91 / 0.65 | 1.26 / 1.39 | 1.96 / 2.12 | 3.64 / 2.90 |
| $\varphi_1 = 10.0$ | 1.07 / 1.09 | 1.06 / 0.66 | 1.92 / 1.94 | 3.65 / 2.87 |

| **Nails** | $r_0 = 2.5$ | $r_0 = 5.0$ | $r_0 = 7.5$ | $r_0 = 10.0$ |
|---|---|---|---|---|
| $\varphi_1 = 2.5$ | 371 | 351 | 324 | 288 |
| $\varphi_1 = 5.0$ | 493 | 472 | 443 | 416 |
| $\varphi_1 = 7.5$ | 500 | 497 | 477 | 468 |
| $\varphi_1 = 10.0$ | 499 | 496 | 493 | 468 |

Table 3.11: Phase II: Timings for different sizes of the C-space region $\mathcal{N}_2$. K is set to 32. Mean value / standard deviation over 10 runs for all 50 paths that have been planned in phase I. Not all planning queries could successfully reach a goal configuration and timed out after 20 seconds. The number of successful attempt is shown in the second table for both scenarios.

For large regions, we have to examine a bigger portion of the configuration space. We recall that the local planner in the second phase does not use sample retraction but we use a basic straight line local planner here. If we do not focus the search on a restricted portion of the configuration space, the success rate of this planner begins to drop. It is also possible that the LBRRT planner might suffer from a "local minima problem" like we have sketched in Fig. 3.9. We have observed this effect when traversing the first narrow passage of the nails puzzle. This explains the poor success rates in the Nails benchmark compared to the Alpha1.0 puzzle. In this case, it is advisable to restart the second phase multiple times to handle this problem. None the less, the reader should note that for an adequate choice of this parameter, we have been able to successfully re-plan all 50 paths of the first planning phase. If the size of $\mathcal{N}_2$ is carefully chosen, we even managed to successfully re-plan all 50 paths in all of the 10 attempts. In the following benchmarks, we are going to use the parameter values $(r_0,\ \varphi_1) = (2.5,\ 7.5)$ for the size of $\mathcal{N}_2$ in the second planning phase.

### 3.9.7 Total Running Time

Table 3.12 summarizes the running times for the different motion planning puzzles we have solved with our algorithms (cf. Fig. 3.10, page 100). The left side of the table shows the timings if phase I was performed without shrinking the robot in advance, the right side shows the effect that shrinking the robot has on the performance of the algorithm. Again, shrinking the robot does not only dilate the free configuration space but this also allows us to perform more efficient, approximate collision tests. This operation is only applicable to watertight models (this is always the case for the metal puzzles). The two sub-tables break the running time down into phase I, phase II and the total time. We see that in almost all our benchmark scenarios, it was beneficial to slightly shrink the robot. We can then achieve a speed-up from 2.5 to 3.5 times compared to the "no shrink" version. The speed-up is higher for more complex benchmarks with more then one narrow passage (like the nails and twisted arcs benchmarks) or if there is a very tight passage like in the flange scenario.
XPuzzle225 and XPuzzle210 differ in the radius of the respective tubes. For the latter, the objects consist of a thinner tube which makes the puzzle easier to solve. This was the only benchmark scenario, where removing one voxel layer of the grid representation of the robot resulted in changes of the free space topology. The algorithm was then able to trivially separate the two shapes. This is why we give no timings for the "shrink" version in this case. Typically, the quality of the paths that result from the first planning phase is better if we do not shrink the robot in phase I. This reflects in the fact that the second phase is 2 to 3 times faster in this case. However, since the second phase only takes a low fraction of the overall running time, this is not much of a problem. Most interestingly, throughout our benchmarks, despite of the fact that we did not exactly keep track of the error that was induced by the shrinking of the robot plus the retraction-based local planner, we did never run into a situation where we could not repair a solution path of the



first planning phase (except of the XPuzzle210 benchmark, of course). This motivates, why it can be a good idea to use a non-conservative planner in plahe I.

| | **no Shrink** | | | **Shrink** | | |
|---|---|---|---|---|---|---|
| | Phase I | Phase II | Total | Phase I | Phase II | Total |
| Alpha1.0 | 29.74 ( 7.64) | 0.44 (0.14) | 30.18 ( 7.64) | 9.25 ( 2.29) | 1.03 (1.57) | 10.28 ( 2.77) |
| Nails | 109.21 (42.69) | 0.61 (0.32) | 109.82 (42.69) | 30.34 ( 9.68) | 1.95 (3.01) | 32.29 (10.13) |
| AlphaNails(a) | 23.39 ( 7.79) | 0.30 (0.05) | 23.69 ( 7.79) | 8.39 ( 2.23) | 0.50 (0.70) | 8.89 ( 2.33) |
| AlphaNails(b) | 26.68 ( 6.52) | 0.48 (0.23) | 27.16 ( 6.52) | 10.10 ( 1.41) | 0.63 (0.42) | 10.73 ( 1.47) |
| XPuzzle210 | 18.37 ( 6.07) | 0.24 (0.12) | 18.61 ( 6.07) | * | * | * |
| XPuzzle225 | 26.10 ( 8.10) | 0.25 (0.18) | 26.35 ( 8.10) | 10.34 ( 3.73) | 0.27 (0.12) | 10.61 ( 3.73) |
| TwistedArcs(a) | 72.66 (23.09) | 1.24 (2.13) | 73.90 (23.18) | 15.89 ( 2.78) | 1.46 (2.24) | 17.35 ( 3.57) |
| TwistedArcs(b) | 286.17 (93.30) | 0.55 (0.54) | 286.72 (93.30) | 56.09 (18.82) | 0.51 (0.09) | 56.60 (18.82) |
| Flange | 24.01 ( 2.65) | 2.96 (3.13) | 26.97 ( 4.10) | 5.48 ( 0.46) | 2.34 (1.60) | 7.82 ( 1.66) |

Table 3.12: Breakdown of the running times for all benchmark puzzles with and without shrinking the robot in phase I. All timings show the mean / std deviation over 50 runs in seconds. *: Shrinking the robot for the XPuzzle210 changed the topology of the configuration space. We do not give timings, here.



| | Alpha1.0 | | | |
|---|---|---|---|---|
| | 3.5 | 7.0 | 10.5 | 14.0 |
| 0.1 | 24.24 | 4.21 | 2.87 | 2.04 |
| 0.2 | 7.40 | 2.89 | 2.56 | 3.00 |
| 0.3 | 4.49 | 2.81 | 2.49 | 3.37 |
| 0.4 | 3.73 | 2.53 | 2.73 | 3.01 |

| | Nails | | | |
|---|---|---|---|---|
| | 3.5 | 7.0 | 10.5 | 14.0 |
| 0.1 | 24.79 | 13.92 | 16.28 | 13.77 |
| 0.2 | 18.10 | 8.61 | 7.46 | 8.03 |
| 0.3 | 16.91 | 7.12 | 5.89 | 6.33 |
| 0.4 | 14.95 | 5.67 | 5.32 | 6.98 |

Table 3.13: Timings for different sizes of the C-space region $\mathcal{N}$. L is set to 128. The tables show the *minimal* running time out of 50 runs in seconds.

## 3.10 Conclusions and Future Work

We have described a novel sampling-based motion planning algorithm that works in two phases. In the first phase, we conceptually allow small robot - obstacle interpenetrations in order to speed up the planning process. This is achieved by (a) a non-conservative retraction-based local planner and (b) shrinking the robot before the planning process. Both techniques help to cope with narrow passages in the configuration space. Allowing small interpenetrations between robot and obstacle effectively dilates the free space and thus widens narrow passages. The retraction-based local planner uses local contact information to guide the local planning process. This can improve the effectiveness of local planning and thus the efficiency of the overall planner not only in narrow passages but also in otherwise restricted free space regions. We can shrink a watertight robot object by representing it with a grid (cf. Sec. 2.7) and removing one layer of voxels from this representation. This results in a non-conservative, approximate collision test. In addition to the reduced size of the robot model, the approximate tests are cheaper than exact collision tests, as we do not perform actual triangle triangle intersection tests at the bottom level of the hierarchy. The improved collision test throughput is also beneficial for the planning efficiency of the first phase.
As a basic planning scheme, we decided to use ESTs instead of the more popular RRTs. An EST expands its nodes with a probability that is reciprocally proportional to the number of samples in a neighborhood of that node. This results in a uniform sampling in unconstrained free space regions but also in a relatively higher sample density in constrained free space regions like narrow passages, free space borders or otherwise highly constrained regions (see Fig. 3.8, p. 94). In contrast to the standard EST scheme, we used a greedier strategy that expands the $L$ nodes with smallest weights at every iteration. This leaves us with a moderately high number of individual local planning tasks that have to be performed at every iteration of the algorithm's main loop. According to the results of Chap. 2, we decided to parallelize this step by distributing the local planning tasks evenly amongst the available CPU cores.
The paths that result from the first phase are probably not completely collision free. In order to repair a path, we add a second planning phase, that locally re-grows a tree in a small neighborhood of this path. The second phase uses no contact information for sample retraction but uses simple linear interpolation for local planning. Of course, we

do not shrink the robot in phase II, neither. In order to realize the localized planning around a solution path, we restrict the sampling domain of the planner for phase II to a tube shaped region around this path. A weight function that measures the distribution of samples along the path biases the sampling process. We call this new planning scheme LBRRT, Locally Biased RRT.
We have implemented the algorithm and tested it with a family of highly constrained motion planning problems with many narrow passages including the famous alpha puzzle and flange scenario (see Fig. 3.10). Opposed to commonly used strategies, we did not support the algorithm with a dedicated goal position. So the algorithm cannot bias the sampling towards this goal which could probably make it easier to plan through narrow passages, if the goal would be on the other side of such a passage. It also prevents us from easily implementing a bi-directional planning scheme. Instead, the set of goal configurations is implicitly defined by the set of all configurations that separate the bounding boxes of robot and obstacle.
We have tested the behavior of the algorithm for different choices of parameters, namely the sizes of the neighborhoods for the evaluation of our weight functions and the parallel growth width, and we have compared it against the standard EST scheme. As it turns out, it is not profitable to explore too many nodes in parallel (i.e. we should not use very big values for L and K), but the algorithms are not highly sensitive to this choice. The size of the neighborhood regions $\mathcal{N}$ and $\mathcal{N}_2$ has a severe effect on the performance and the success of the planning algorithm. Using too small regions can have a negative effect on the performance of the algorithm as this results in a very dense sampling of the unconstrained free space regions. If we use too big regions, the algorithm cannot classify constrained free space regions properly as it cannot "see" features that are relatively small compared to the size of $\mathcal{N}$. For example, imagine the region in Fig. 3.8 (page 94 was so big that it covered the whole configuration space window. Clearly, the size of the neighborhood has to be chosen according to the size of the structures and the dimensions of the free space. In this work, we have experimentally determined good values and thus adjusted these parameters manually.

In future work, it would be desirable to investigate if it was possible to automatically adjust these parameters for different planning scenarios. Maybe it would even be possible not to use a fixed size for these parameters, but to adaptively adjust the size of the neighborhoods according to different free space regions. Up to now, it is unclear how this should be realized in detail.
We can see in Table 3.4 that for big regions $\mathcal{N}$, the algorithm failed to find a solution path within a predefined time limit. Let us however have a look at the best running times out of the 50 runs we performed for the differently sized regions. The values are given in Table 3.13. We can see that the best running times are roughly 3 to 4 times better than the best average times. And, more importantly, the timings are not as sensitive to big sizes of $\mathcal{N}$, as we have seen above. Thus, it could be beneficial to run the same planning problem numerous times in parallel. All instances can then be terminated if the first solution was found. We have not looked into this in detail, but it is probably worthwhile to analyze this effect in future work, too. The parallel



instances are completely independent and could even run on distant devices with only a slow communication. Running multiple instances of the planner could be particularly interesting for the second planning phase that, as we have mentioned, can get stuck in "local minima" for the wrong sizes of neighborhoods.

In the second planning phase, we use accurate collision tests and use linear interpolation for local planning instead of our non-conservative retraction-based local planner. But still, we perform discrete collision tests for a finite number of points sampled at the line that connects two nodes of the EST. This is still a source of error. No matter how densely we sample the solution path, it is always possible for the robot to jump over a collision point and that the path is not continuously collision free. This is why recent work has focused on the discussion of continuous collision tests. Using these tests, it is possible to decide if a certain trajectory (like the linear interpolation between two configurations) is completely collision free. However, these tests are typically more expensive than a simple collision test. If we are interested to continuously verify our solution path, we propose to add a third phase to the planning process that uses continuous collision checking. In this case, the planning domain could probably be chosen to be only a small tube around the discrete solution path from phase II. As a proof of concept, we have prototyped a continuous collision checker based on a technique called *conservative advancement* [32, 62] and used it to re-plan the solution paths for the alpha puzzle in a third planning phase. We have tested the algorithm with the solutions of the Alpha1.0 puzzle. Continuous re-planning took around 30 seconds in average (with a standard deviation of 10s).
We have basically implemented this by shifting from an OBB hierarchy to a hierarchy of Rectangular Swept Spheres (RSS) [24] to allow distance computations and we have substituted the basic collision tests during BVH traversal with the distance tests sketched in Sec. 2.3. Most interestingly, this has already lead to promising results in our experiments. Virtually the whole running time of the third phase was spent on proximity queries in the local planning phase. So we can potentially achieve better results by an improved continuous collision detection algorithm. An analysis of the continuous collision detection mechanism and a more rigorous testing of the third planning phase are other interesting avenues for further work.

# 4 Conclusions and Future Work

In the present work, we have seen how we can design efficient parallel collision detection algorithms for multi-core CPUs as well as CUDA capable GPUs. As a single collision test needs only a rather limited number of instructions and as we aimed at an application to sampling-based motion planning, we decided to perform the parallelization over a high number of independent collision tests for the same two query objects. We have seen that in order to fully occupy the parallel device, we need hundreds of parallel collision tests on current CPUs and more than one hundred thousand tests on the GPU. We have focused solely on the analysis of collision tests, but the same ideas and techniques can also be applied to distance and tolerance tests with only small changes. When switching over to other proximity queries, like distance tests, or when computing the set of all intersecting triangle pairs, the individual tests become more expensive. It is then interesting to think about possibilities to parallelize a single (or a small number of) tests. This can be especially interesting in the context of interactive real-time applications.

As far as the underlying hardware is concerned, in the near past, there has been a rapid progress in the development of parallel compute units and new generations of parallel devices became available for the developer within short periods of time. While we wrote this work, the next generation of Nvidia GPUs, codenamed *Kepler*, became publicly available. On the CPU-side of the scale, Intel is making progress with its Many Integrated Core Architecture and announced the new *Xeon Phi* family of parallel devices. It would be an interesting question how our algorithms can be ported to the new hardware and how they scale with the number of available compute cores.

In the second part of this work, we have seen how we can apply parallel collision tests to sampling-based motion planning and achieve a high speed-up. Here, we have focused especially on the treatment of highly complex rigid body motion planning problems with six degrees of freedom. As we have stressed in the introduction of this work, motion planning has important applications when dealing with digital mock-ups in CAD/CAM. With the introduction of our metal puzzle benchmarks, we analyzed a very complex class of motion planning problems with many narrow passages. It would now be interesting, to test our algorithm in more practical scenarios with real world data. While the complexity of the motion trajectories is already pretty high for the metal puzzles, real world objects are typically a lot more detailed and can consist of hundreds of thousands of triangles. Unfortunately, it is not always easy to get hold of such data but at the time of writing, we have already had the opportunity to use our algorithm in one test scenario to plan a collision free disassembly path of an engine from an engine bay. Despite the high triangle count of robot and obstacle, this was no problem for our approach and

required no further modifications. We are looking forward to test our algorithm with further real-world data in the foreseeable future.
In this context, there are many interesting modifications and expansions to planning a path for a single, free floating body in 3D. One possibility would be to let an actual industrial robot grip the component we want to be disassembled and to compute an adequate motion for the kinematic chain of the robot arm. This could be achieved by either planning a six degrees of freedom free floating object like we have already done and using an inverse kinematic solver to validate the grip of the kinematic chain for every sample. Another possibility could be to switch the configuration space to the space of joint parameters of the chain. In the second case, it is an interesting question of how to adapt the different components of our algorithm to the new formulation of the configuration space. Here, one possibility to implement sample retraction together with kinematic chains could be to utilize Featherstone's algorithm to propagate the forces that act on the robot object through the chain. It is unclear what would be the best choice to port the high-level planning mechanism and if it would be profitable to use an EST strategy in this setup after all.
Another interesting expansion of the original problem formulation would be to plan paths for more than just one robot that can move simultaneously and share a common workspace. In this respect, the individual robots appear as dynamic obstacles for each other. Planning a path for two moving objects would then evoke a 12-dimensional configuration space and it is yet unclear if our algorithms can handle this without bigger modifications.

The reader can see that there are many avenues for future work. For now, we are proud to have achieved our goal to solve a family of very complex motion planning problems with many narrow passages. We are planning to make the new benchmark suite publicly available so that other groups can test their planning algorithms with our puzzles as well. We hope that the results of this thesis can help the community to make further progress in the fields of parallel computing and sampling-based motion planning.

# Bibliography


[1] J. Arvo. Graphics Gems III. chapter Fast Random Rotation Matrices, pages 117–120. 1992. 85

[2] G. Baciu, W. S.-K. Wong, and H. Sun. RECODE: An Image-Based Collision Detection Algorithm. In *Proceedings of Pacific Graphics '98*, 1998. 6, 12

[3] J. Barraquand, L. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, and P. Raghavan. A Random Sampling Scheme for Path Planning. *Int. Journal of Robotics Research*, pages 759–774, 1996. 80

[4] J. Damkjær and K. Erleben. GPU Accelerated Tandem Traversal of Blocked Bounding Volume Hierarchy Collision Detection for Multibody Dynamics. In *VRIPHYS*, pages 115–124, 2009. 6, 12

[5] D. H. Eberly. *3D Game Engine Design, Second Edition: A Practical Approach to Real-Time Computer Graphics (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., 2006. 15

[6] R. Erbes, A. Mantel, E. Schömer, and N. Wolpert. Parallel Collision Queries on the GPU – A Comparative Study of Different CUDA Implementations. In *Facing the Multicore-Challenge III*, volume 7686 of *LNCS*. Springer, 2012. 2

[7] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005. 9, 11, 42

[8] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space. *IEEE Journal of Robotics and Automation*, pages 193–203, 1988. 5, 15

[9] S. Gottschalk. *Collision Queries using Oriented Bounding Boxes*. PhD thesis, The University of North Carolina at Chapel Hill, 2000. 16, 17

[10] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Proc. of Conf. on Computer Graphics and Interactive Techniques*, pages 171–180, 1996. 6, 10, 11, 12, 25

[11] N. Govindaraju, S. Redon, M. Lin, and D. Manocha. CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*, pages 25–32, 2003. 6, 12

[12] B. Heidelberger, M. Teschner, and M. H. Gross. Real-Time Volumetric Intersections of Deforming Objects. In *Proc. of the Vision, Modeling and Visualization Conference*, pages 461–468, 2003. 6, 12

[13] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha. Memory-Scalable GPU Spatial Hierarchy Construction. *IEEE Trans. on Visualization and Computer Graphics*, pages 466–474, 2011. 11

[14] D. Hsu. The Bridge Test for Sampling Narrow Passages with Probabilistic Roadmap Planners. In *Proc. IEEE Int. Conf. on Robotics & Automation*, pages 4420–4426, 2003. 80

[15] D. Hsu, J. claude Latombe, and R. Motwani. Path Planning in Expansive Configuration Spaces. In *Int. Journal of Computational Geometry and Applications*, pages 2719–2726, 1997. 80, 83, 94, 106

[16] D. Hsu, L. E. Kavraki, J.-C. Latombe, R. Motwani, and S. Sorkin. On Finding Narrow Passages with Probabilistic Roadmap Planners. In *Workshop on the Algorithmic Foundations of Robotics*, pages 141–153, 1998. 80

[17] D. Hsu, G. Sánchez-Ante, H.-L. Cheng, and J.-C. Latombe. Multi-level Free-space Dilation for Sampling Narrow Passages in PRM Planning. In *IEEE Int. Conf. on Robotics and Automation*, pages 1255–1260, 2006. 80

[18] C. L. J. Pan and D. Manocha. g-Planner: Real-time Motion Planning and Global Navigation Using GPUs. In *Conf. on Artificial Intelligence*, 2010. 82

[19] M. W. Jones, J. A. Baerentzen, and M. Sramek. 3D Distance Fields: A Survey of Techniques and Applications. *Transactions on Visualization and Computer Graphics*, pages 581–599, 2006. 6

[20] D. Kim, J.-P. Heo, J. Huh, J. Kim, and S.-E. Yoon. HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs. *Computer Graphics Forum (Pacific Graphics)*, pages 1791–1800, 2009. 6, 12

[21] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-Dops. *IEEE Trans. on Visualization and Computer Graphics*, pages 21–36, 1998. 6, 10

[22] D. Knott and K. P. Dinesh. CInDeR Collision and Interference Detection in Real-time using Graphics Hardware. *Computer Graphics Forum*, 2003. 6, 12

[23] J. Kuffner and S. LaValle. RRT-connect: An Efficient Approach to Single-query Path Planning. *IEEE Int. Conf. on Robotics and Automation*, pages 995–1001, 2000. 80, 83, 93

[24] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha. Fast Distance Queries with Rectangular Swept Sphere Volumes. In *Proc. of IEEE Int. Conference on Robotics and Automation*, pages 3719–3726, 2000. 6, 10, 11, 18, 118

[25] T. Larsson and T. Akenine-Möller. Bounding Volume Hierarchies of Slab Cut Balls. *Comput. Graph. Forum*, pages 2379–2395, 2009. 10

[26] J.-C. Latombe, L. Kavraki, P. Svestka, and M. Overmars. Probabilistic Roadmaps for Path Planning in High-dimensional Configuration Spaces. *IEEE Transactions on Robotics and Automation*, pages 566–580, 1996. 80, 83, 93

[27] C. Lauterbach, M. Garland, S. Sengupta, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics*, pages 375–384, 2009. 6, 12

[28] C. Lauterbach and Q. Mo. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Computer Graphics Forum*, pages 419–428, 2010. 6, 12, 30

[29] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Also available at http://planning.cs.uiuc.edu/. 1, 79

[30] M. C. Lin and J. F. Canny. A Fast Algorithm for Incremental Distance Calculation. In *In IEEE Int. Conf. on Robotics and Automation*, pages 1008–1014, 1991. 5, 15

[31] W. A. McNeely, K. D. Puterbaugh, and J. J. Troy. Six degree-of-freedom Haptic Rendering using Voxel Sampling. In *Proc. of Conf. on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 401–408, 1999. 44

[32] B. V. Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley, 1996. 118

[33] T. Morvan, M. Reimers, and E. Samset. High Performance GPU-based Proximity Queries using Distance Fields. *Computer Graphics Forum*, pages 2040–2052, 2008. 6, 12

[34] K. Myszkowski, O. G. Okunev, and T. L. Kunii. Fast Collision Detection between Complex Solids using Rasterizing Graphics Hardware. *The Visual Computer*, pages 497–511, 1995. 6, 12

[35] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007. 44

[36] NVIDIA. *CUDA C Programming Guide, Version 5.0.* 19, 20, 38

[37] NVIDIA. *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi.* 19, 20

[38] S. M. Omohundro. Five Balltree Construction Algorithms. Technical report, International Computer Science Institute, Berkeley, California. 11

[39] S. Pabst, A. Koch, and W. Straßer. Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *Comput. Graph. Forum*, pages 1605–1612, 2010. 6, 12

[40] J. Pan. GPU-based Parallel Collision Detection for Real-Time Motion Planning. *Algorithmic Foundations of Robotics IX*, pages 211–228, 2011. 82

[41] J. Pan, C. Lauterbach, and D. Manocha. Efficient Nearest-Neighbor Computation for GPU-based Motion Planning. In *Int. Conf. on Intelligent Robots and Systems*, pages 2243–2248, 2010. 82

[42] J. Pan and D. Manocha. GPU-based Parallel Collision Detection for Fast Motion Planning. *The International Journal of Robotics*, 2012. 12, 29, 38

[43] S. Quinlan. Efficient Distance Computation between Non-Convex Objects. In *In Proc. of Int. Conf. on Robotics and Automation*, pages 3324–3329, 1994. 6, 10

[44] S. Redon and M. Lin. Practical Local Planning in the Contact Space. In *IEEE Int. Conf. on Robotics and Automation*, pages 4200–4205, 2005. 82, 92

[45] M. Renz, C. Preusche, M. Pötke, H.-P. Kriegel, and G. Hirzinger. Stable Haptic Interaction with Virtual Environments Using an Adapted Voxmap-Pointshell Algorithm. In *In Proc. Eurohaptics*, pages 149–154, 2001. 44

[46] S. Rodriguez and N. Amato. An Obstacle-based Rapidly-exploring Random Tree. *IEEE Int. Conf. on Robotics and Automation*, pages 895–900, 2006. 82

[47] K. Shoemake. Animating Rotation with Quaternion Curves. In *Proc. of Conf. on Computer Graphics and Interactive Techniques*, SIGGRAPH'85, pages 245–254, 1985. 89

[48] K. Shoemake. Graphics Gems III. pages 124–132. 1992. 84, 85

[49] M. Stich, H. Friedrich, and A. Dietrich. Spatial Splits in Bounding Volume Hierarchies. In *Proc of the Conference on High Performance Graphics*, HPG '09, pages 7–13, 2009. 50

[50] A. Sud, N. Govindaraju, R. Gayle, and D. Manocha. Interactive 3D Distance Field Computation using Linear Factorization. In *Proc. of Symposium on Interactive 3D Graphics and Games*, pages 117–124, 2006. 6, 12

[51] Z. Sun, D. Hsu, T. Jiang, H. Kurniawati, and J. H. Reif. Narrow Passage Sampling for Probabilistic Roadmap Planning. *IEEE Transactions on Robotics*, pages 1105–1115, 2005. 80

[52] M. Tang, D. Manocha, and R. Tong. MCCD: Multi-core Collision Detection between Deformable Models using Front-based Decomposition. *Graphical Models*, 2010. 6, 12

[53] G. van den Bergen. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *J. Graphics Tools*, 2, 1998. 27

[54] V. Volkov. Better Performance at Lower Occupancy. In *Proc. GPU Technology Conf.*, GTC, 2010. 21

[55] V. Volkov and J. W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proc. of Conf. on Supercomputing*, page 31, 2008. 20

[56] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, pages 52–69, 1984. 10

[57] A. Yershova, S. Jain, S. M. Lavalle, and J. C. Mitchell. Generating Uniform Incremental Grids on SO(3) Using the Hopf Fibration. *International Journal on Robotics Research*, pages 801–812, 2010. 84

[58] A. Yershova and S. M. LaValle. Improving Motion Planning Algorithms by Efficient Nearest-Neighbor Searching. *IEEE Transactions on Robotics*, pages 151–157, 2007. 82

[59] G. Zachmann. The BoxTree: Exact and Fast Collision Detection of Arbitrary Polyhedra. In *In SIVE Workshop*, pages 104–112, 1995. 27

[60] L. Zhang, Y. J. Kim, and D. Manocha. C-DIST: Efficient Distance Computation for Rigid and Articulated Models in Configuration Space. In *Proc. of the 2007 ACM Symposium on Solid and Physical Modeling*, SPM '07, pages 159–169, 2007. 82

[61] L. Zhang and D. Manocha. An Efficient Retraction-based RRT Planner. In *IEEE Int. Conf. on Robotics and Automation*, pages 3743–3750, 2008. 82, 92

[62] X. Zhang, M. Lee, and Y. J. Kim. Interactive Continuous Collision Detection for Non-Convex Polyhedra. *Vis. Comput.*, pages 749–760, 2006. 118