Universität Bremen

Fachbereich 3: Mathematik und Informatik
Computer Graphics

# Master Thesis

## A Volume-Based Penetration Measure for 6DOF Haptic Rendering of Streaming Point Clouds

## Ein auf Volumen basiertes Eindringmaß für 6DOF Haptisches Rendern von wechselnden Punktwolken

Maximilian Kaluschke

Matrikel Nr. 2475447

23th of December, 2016

(Revised 26th of January, 2017)

**Primary Reviewer**: Prof. Dr. Gabriel Zachmann
**Secondary Reviewer**: Dr. René Weller
**Supervisor**: Dr. René Weller

Universität Bremen

Nachname **Kaluschke** Matrikelnr. **2475447**

Vorname/n **Maximilian**

Diese Erklärungen sind in jedes Exemplar der Bachelor- bzw. Masterarbeit mit einzubinden.

## Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Stellen, die ich wörtlich oder sinngemäß aus anderen Werken entnommen habe, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Datum Unterschrift

## Erklärung zur Veröffentlichung von Abschlussarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten.
Archiviert werden:

1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils der ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.

- (●) Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ( ) Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ( ) Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Datum Unterschrift

## ABSTRACT

I present a novel method to define the penetration volume between a surface point cloud and arbitrary 3D CAD objects. Moreover, I have developed a massively-parallel algorithm to compute this penetration measure efficiently on the GPU. The main idea is to represent the CAD object's volume by an *inner bounding volume hierarchy* while the point cloud does not require any additional data structures. Consequently, my algorithm is well suited for streaming point clouds that can be gathered online via depth sensors like the Kinect. I have tested our algorithm in several demanding scenarios and our results show that our algorithm is fast enough to be applied to 6-DOF haptic rendering while computing continuous forces and torques.



## ZUSAMMENFASSUNG

Ich stelle eine neuartige Methode zur Feststellung des Eindringvolumens zwischen einer einfachen Punktwolke und beliebigen 3D CAD Objekten vor. Außerdem habe ich einen massiv-parallele Algorithmus zur effizienten Berechnung dieses Eindringmaßes auf der GPU entwickelt. Die Kernidee ist, dass man das Volumen des 3D CAD Objekts durch eine Bounding Volume Hierarchie die das Objekt von inner annähert, während für die Punktwolke keine weitere Datenstruktur gebraucht wird. Folglich ist mein Algorithmus gut geeignet für wechselende Punktwolken welche man in Echtzeit von einer Tiefenbildkamera wie der Kinect auslesen kann. Ich habe meinen Algorithmus in verschiedenen anspruchsvollen Szenarien getestet und die Ergebnisse zeigen, dass mein Algorithmus schnell genug ist für 6-DOF haptisches Rendern und liefert zugleich stetige Kräfte und Drehmomente.

# ACKNOWLEDGEMENTS

This thesis is not only attributed to me but to many people that helped me over the course of the last year.

Firstly, I want to thank my parents Liane and Martin for their support and for allowing and encouraging me to keep learning.
I owe a lot to my advisor René for always finding time for me when I had a problem and for letting us discuss many of our ideas with one another.
I also want to thank my professor Gabriel for giving me this opportunity and for always offering me his insights and critical feedback.

# CONTENTS

# 1 INTRODUCTION

Collision detection is a fundamental problem that arises in all tasks involving the simulated motion of objects that are not allowed to penetrate each other. For instance, it is necessary for interactive physically-based simulations that are widely used in computer graphics and VR, but also in robotics or the simulation of molecular dynamics. Usually the virtual objects are represented by an abstract geometric model like a triangle mesh of the surface or mathematical functions like non-uniform rational B-spline. In addition of *finding* collisions between such object representations, we also have to *resolve* them in a physically plausible way. Especially in time critical real-time simulations, e.g. when haptic interaction is included, we typically use a penalty-based approach for this collision handling. This means, we allow a small interpenetration of the objects and apply an appropriate force and torque to separate them in the next simulation step.

Obviously, resolving such a situation requires additional information about the amount of interpenetration. Basically, there exist three kinds of contact information for resolving collisions: We can try to find the exact time of impact between two consecutive simulation steps. This is computationally very expensive. Or we can define a minimum translational vector to separate the objects. This is also hard to compute and even worse, it may lead to discontinuities in case of heavy interpenetrations. Finally, we can use the complete *penetration volume*. This penetration measure has been called "the most complicated yet accurate method" [15] to define the extend of interpenetration for a pair of objects.

While general collision detection has been a research topic since more than three decades, the first algorithms to compute the penetration volume for arbitrary CAD objects were developed just a few years ago [13], [57]. However, they support only collision detection between pairs of watertight 3D CAD objects that must have a certain volume. Actually, today virtual environments do not only consist of watertight volumetric objects. Often they contain two-dimensional parts, like thin sheets in virtual prototyping tasks of the automotive industry, or even dimensionless points that form a point cloud. Such point cloud data often appears in case of tracking data, e.g. from body trackers relying on depth sensors like the Kinect or hand tracking devices like the Leap Motion. Collision detection with such tracking data is crucial because normally we add it to a scene to apply interactions with other objects in the virtual environment.

Obviously, we could reconstruct 3D meshes from the point cloud data and then apply the aforementioned collision detection techniques. Unfortunately, mesh reconstruction is time consuming [41] and moreover,

the collision detection methods often require additional time consuming pre-processing that can be hardly performed in real time.

In this thesis, I present a novel method for collision detection between virtual environments that are modelled by arbitrary 3D CAD objects and unstructured point cloud data. The only pre-condition is that the CAD models have to be watertight and that the points in the point cloud represent a surface and have consistent normal information.

In detail, I contribute the following novel ideas to the field of collision detection:

- A volumetric penetration measure for point clouds and CAD models.
- A massively-parallel algorithm that computes this penetration volume efficiently on the GPU.
- A novel penalty-based collision response method relying on the volumetric intersection data that computes continuous forces as well as torques for full 6-DOF phyiscally-based simulations.

The main idea is to represent the volume of the CAD object by simple volumetric primitives and distinguish between parts of those primitives that are inside and outside of the point cloud, based on normal information. To do that, I use, similar to the inner sphere trees (ISTs) described in [57], a polydisperse sphere packing for the CAD object. Additionally, I compute a neighborhood graph to identify spheres that are completely located inside the point cloud. The application of traditional data structures for CAD vs CAD collision detection, like ISTs, has several advantages: First, the re-usage of well known technology simplifies the implementation and reduces errors and second, it is straight forward to add multiple CAD objects to the same scene that can interact with each other in a physically-plausible way, without the need to maintain different data structures for CAD vs CAD and CAD vs pointcloud tests.

My algorithm is easy to implement and handles multiple contacts automatically in a physically plausible way. The results show that my algorithm can perform collision queries at haptic rates for reasonable point cloud sizes. As a use case I present the application of my algorithm to 6-DOF haptic rendering for streaming point clouds that are gathered live via a Kinect. To do that, I additionally present a novel method for an online pre-filtering of the point cloud and a new approach to handle gaps produced by large differences in the depth values.

# 2 PREVIOUS WORK

Haptic rendering often incorporates multiple tasks to offer a complete solution. I will outline the different tasks at hand and give a brief overview of the state-of-art.

## 2.1 COLLISION DETECTION

The topic of collision detection is an essential part in most interactive simulations and computer graphics and it has been extensively researched in the literature. Usually, 3D objects in these scenarios are represented by polygonal meshes. Hence, most work on collision detection has been spent to accelerate queries for this kind of object representation. Often, some kind of bounding volume hierarchy (BVH) is used in order to early prune parts of the geometry that can not collide. Such hierarchies have been described for different bounding volumes that all have their unique strengths and weaknesses, including axis aligned bounding boxes (AABBs) [5], orientated bounding boxes [16], spheres [22] or discrete oriented polytopes [66].

### 2.1.1 *Parallelization of Collision Detection*

All these approaches were designed for sequential processors. Implementations that use parallel CPU instructions like OpenMP [67] or SSE give considerable speedups of around 2.7 compared to sequential algorithms, but there is more potential in modern GPUs. For triangle mesh representations there already exist a few approaches that make use of massively parallel processing of GPUs. For example [25] used the graphics card for collision detection between multiple objects with a single common object. Lauterbach et al. [33] implemented a distance computation using OBB trees on the GPU. Some methods have been described that do not require BVHs: for instance Faure et al. [13] used layered depth images, Mainzer and Zachmann [39] proposed a parallel sweep-and-prune approach and Weller et al. [55] showed an approach that is based on hierarchical grids.

[65] introduce $p$-partitioned fronts, which help to simplify load balancing for parallel collision detection implementations between two BVHs. Recently, neural networks have been used to achieve fixed time collision detection between two polyhedra [26].

### 2.1.2 *Collision Detection on Point Clouds*

Compared to mesh representations, the literature on collision detection for point clouds is relatively sparse. One of the first approaches to detect collision between point clouds was developed by [29]. They use a BVH in combination with a sphere covering of parts of the surface. [30] proposed an interpolation search approach of the two implicit functions in a proximity graph in combination with randomized sampling. [12] support only collisions between a single point probe and a point cloud. For this, they fill the gaps surrounding the points with AABBs and use an octree for further acceleration. [14] used R-trees, a hierarchical data structure that stores geometric objects with intervals in several dimensions [18], in combination with a grid for the broad phase. [44] described a stochastic traversal of a bounding volume hierarchy. By using machine learning techniques, their approach is also able to handle noisy point clouds.

Another approach is approximating the point cloud by a set of axis-aligned cubes, called a collision map [46, 69].

### 2.1.3 *Inner Sphere Trees Datastructure*

Inner sphere trees are an object representation to have a close approximation to the ground truth for geometric queries, such as collisions [61] or haptic rendering [59]. The big difference to other approximations is that the volume is approximated from the inside instead of from the outside.

I will use an algorithm that is similar to one I presented in [24] to register collisions between point clouds and inner sphere trees in a massively-parallel fashion on a GPU. However, since I have no use for measuring proximity, I will be using a simplified version of the algorithm. I just need a boolean collision detection, so the computation will be a lot faster in the uninteresting case of no collision, but otherwise the computational times will be comparable.

## 2.2 PENETRATION DEPTH

Penetration depth is the measurement of how far two objects are colliding with each other and can be segmented in two categories, translation penetration depth ($PD^t$) and generalized penetration depth ($PD^g$).

### 2.2.1 *Translational Penetration Depth*

$PD^t$ describes the magnitude of the minimal translation of one of the objects that resolves the collision between the two objects. The concept was introduced by [4], which shows algorithms for calculating the $PD^t$ between two convex primitives using Minkowski sum. There have been many improvements published after that, mostly on more general data structures.

A translational solution is of course only useful for translational haptic rendering like 3-DOF, whereas 6-DOF requires torques to be calculated as well. Although there are ways to combine multiple translational results to approximate the rotational part [36], they do not offer accurate results.

### 2.2.2 *Generalized Penetration Depth*

$PD^g$ is the generalized penetration depth, measuring the minimal transformation applied to one of the two colliding objects to resolve the collision. The metric to compare the trajectory of general transformations has to formulated by some metric, like [62, 63].

[64] introduced the concept by solving $PD^g$ between two polyhedral models.

## 2.3 SURFACE ESTIMATION

The topic of surface estimation has been researched for a long time, even before range scanning cameras were common tools.

[21] first introduced the least-squares approach to estimate normals at a specific point by looking at it's neighbour points. Like here, the application often does not require real-time ready computations. In robotics [38] for example you can scan the whole environment that will be traversed and compute the surface for the composition of all scans. So their focus is mostly on quality and computation times often are in the order of seconds to analyze one point cloud [7, 40, 54], some don't mention computation times at all [37, 43].

### 2.3.1 *Voronoi-Based Estimation*

Visualization of point clouds is another application where most algorithms are some form of Voronoi-based or Delauny-triangulation method [1, 2, 10, 50]. [9] compares multiple of approaches that fall into this category with approaches that use least-squares based methods, which are shown to be considerably faster, more accurate, but less reliable under noise. Voronoi-based algorithms are geared more towards use-cases with static scenes, where quality is significantly more important than speed.

One of the reasons these solutions take so much time is that they operate on unorganized point clouds. However, in this work I will be using a Kinect to generate point clouds, so I will take advantage of the fact that they are organized point clouds, simplifying the surface estimation.

### 2.3.2 *Estimation Based on Covariance Matrices*

There have also been studies on simpler and therefore faster algorithms that are more fit my use case. For example [53] compares the accuracy

of several very basic approaches. Most of the simpler approaches use principal component analysis (PCA) to fit a plane through a neighbourhood of points [3, 20, 53]. [23] compares various modifications of PCA, singular value decomposition (SVD) and similar algorithms under different circumstances in detail, which shows plane fitting via PCA with normalization to be most accurate and fastest on average. [28] illustrates and compares different approaches in artificial and real point clouds.

I will be using a simplified version of plane fitting via PCA in my implementation.

### 2.3.3 *Estimation by Averaging Triangles*

[28] also include another category of estimation methods that are based on averaging triangles formed by neighbouring points. This type of technqiue is also utilized in [6]. [28] comes to the conclusion that they offer less quality normals, especially under noisier data, so they are not fit for my use-case. Their experiments show results similar to [23], which are that plane fitting via PCA & SVD to be most accurate and fastest.

Other simpler approaches are based on calculating the crossproduct of tangential vectors on the point that is being analyzed, most prominently used in robotics in unknown terrain [19, 20, 32].

## 2.4 HAPTIC RENDERING

Haptic rendering is a notion that describes a complete solution to perform haptic rendering, which includes solution to the problems previously mentioned in this chapter.

Haptic rendering solutions are either penalty-based or constraint-based. Penalty-based algorithms continuously check how far the virtual tool has penetrated the environment and calculates an appropriate penalty which is then applied to the physical haptic device. These algorithms are usually easier to implement but have trouble with the virtual tool popping through thin parts of the environment.

Constraint based algorithms usually use some form of virtual-coupling, which means there is an additional virtual object in the scene which can't penetrate objects. All the movement of the haptic device is applied to the virtual object, but under the mentioned constraint. To calculate the output force to be applied to the haptic device, a virtual spring is span from the haptic device position to the virtual object.

### 2.4.1 *Three Degrees of Freedom*

This topic has been well-explored, the most notable approaches are the constraint-based god-object [68] and proxy-method [45], both operate on environments in polygonal data.

[11] introduce an algorithm to perform haptic rendering on an unorganized point cloud by pre-computing knowledge about their neighbouring points into each point. [35] construct an implicit surface by defining meta-balls at each point in the point cloud, resulting in a smooth surface. Both of these have pre-computation times of about a second, so are not fit to work on changing point clouds in real-time.

Another approach pre-computes a regular grid over the point cloud [51]. This enables very fast neighbourhood searches, but again will only be feasible for static scenes.

There are also approaches that operates directly on live-streaming point clouds from devices like a Kinect. [47, 49] extend the classic proxy-method from [45] to work on changing point clouds retrieved from a Kinect by fitting a plane through all points that are inside the spherical proxy, similarly to [31, 51].

### 2.4.2 *Six Degrees of Freedom*

One of the early real-time capable solution here was [42], which voxelized the virtual tool and represented the environment as point shells with inwards facing normals. The environment point shells are transformed by the inverse orientation of the virtual tool, but in itself is static.

There have been different solutions to 6-DOF haptic rendering on static scenes since [8, 17, 27, 36], however the field is not as well explored 3-DOF haptic rendering, because of the increased complexity.

Solutions that work on point clouds also exist already. [34] expand on the meta-ball algorithm from [35] to allow for 6-DOF haptic rendering of a static scene with the robot's gripper as the virtual tool.

[49] started by extending the classic proxy-method to work on streaming point clouds, but in this first version they supported only 3-DOF haptic rendering because the haptic probe was represented by a single point. The same author later introduced a method for 6-DOF haptic rendering on streaming point clouds [48]. This is the closest publication to what I want to achieve, however for slightly different data representations. They rely on the classic VPS algorithm The running time are dependent on the number of points in the pointshell.

Most implementations are available only for the CPU and hence, the number of supported points in the pointshell is restricted. Moreover, none of these extensions was able to overcome the huge memory-footprint of the voxmap and the need for different data structures for moving and fixed objects. Additionally, the resulting forces and torques are very noisy [56].

# 3 CONCEPT

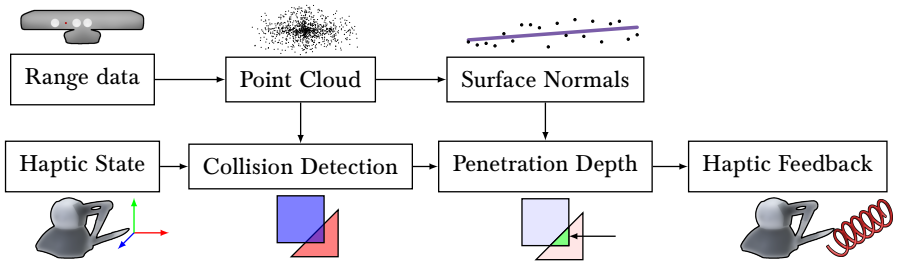



**Figure 3.1**: My haptic rendering pipeline.

## 3.1 CHALLENGE

As previously mentioned, haptic rendering includes multiple subtasks that need to be performed. I illustrated an overview of these subtasks of my application in Figure 3.1 in the form of a pipeline.

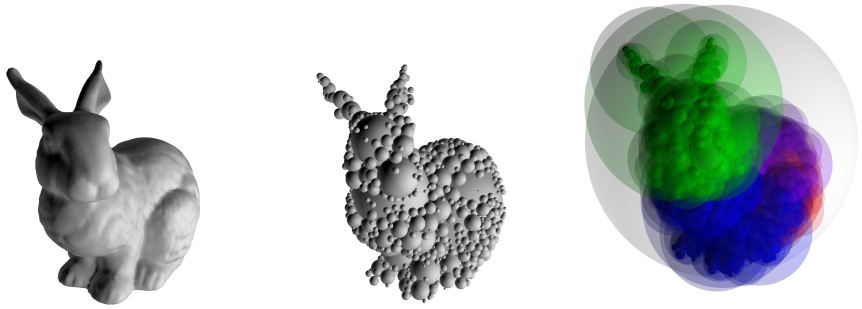
### 3.1.1 *Inner Sphere Trees*

One of the datastructures I deploy are inner sphere trees (ISTs) introduced in [58] for the representation of the virtual tool. I included one example inner sphere tree construction with its bounding volume hierarchy of the Stanford bunny in Figure 3.2.

ISTs are poly-disperse sphere packings of the CAD model, which can be generated by the Protosphere algorithm [60]. This algorithm produces space-filling sphere packings for almost any 3D object representation, including polygonal meshes, CSG and NURBS. The virtual tool is assumed to be rigid, so it suffices to pre-compute an IST once with the desired precision.

## 3.2 VOLUMETRIC PENETRATION MEASURE

The core issue and contribution of my work is the design of a volumetric penetration measure between point clouds and 3D CAD models approximated by ISTs. The CAD model that is used as a virtual tool is required to be water-tight and additionally, each point in the point cloud needs to have a corresponding normal point towards the outside of the approximated environment. Many depth sensors provide these normals automatically. However, the Kinect does not provide them, so we calculate our own surface approximation as I later describe in section 3.4.



**(a)** The original mesh as ground truth.

**(b)** The sphere packing that represents the volume.

**(c)** The bounding volume hierarchy to accelerate the traversal.

**Figure 3.2**: Polygonal mesh and it's IST representation.

A point cloud that intersects an inner sphere tree basically divides the spheres into three different parts:

- **Boundary**: Spheres that are intersected by at least one point.
- **Outside**: Spheres that are outside of the implicit surface generated by the point cloud.
- **Inside**: Spheres that are located completely inside the point cloud surface.

The *penetration volume* consists of the volume of all inside spheres and the inside part of all boundary spheres. In case of a single intersecting point in a boundary sphere, the intersection volume is simply the spherical cap defined by the plane consisting of the point $p$, its normal $n$ and the boundary sphere $s$ with radius $r$. The volume of the spherical cap is given by

$$V = \frac{1}{3}\pi h^2(3r - h) \tag{3.1}$$

where $h$ is the height of the spherical cap, i.e. $h = r - d$ with $d$ being the distance of the plane to the center $c$ of $s$ given by

$$d = \hat{n} \times (c - p) \tag{3.2}$$

The actual algorithm has to consider several other cases, which is why I define it in its completeness in Algorithm 3.1. Unfortunately, this creates a possible place where some discontinuity is introduced. Since in the worst case a point can enter a sphere from the backside, effectively adding almost all of the sphere's volume to the penetration volume. I did not solve this problem yet, but have come up with an idea that counteracts it. However, I did not implement this yet, so no experiments will be given to verify this proposal.

In order to avoid discontinuities that may appear from the previously mentioned single point that enters a sphere and directly adds the complete intersection volume $V_{cap}$, we add a weighting factor $w$ that takes the

**Algorithm 3.1**: intersectedVolume($s$ Sphere, $p$ Point, $\vec{n}$ Normal)

$c_s \leftarrow$ center of $s$
$r_s \leftarrow$ radius of $s$
$d \leftarrow \vec{n} \cdot c_s - (\vec{n} \cdot p)$
**if** $|d| < r_s$ **then**
    $h \leftarrow r - d$
    **return** $\frac{1}{3}\pi h^2(3r - h)$
**if** $d \geq 0$ **then**
    **return** 0  `// Sphere completely in-front of plane`
**return** $\frac{4r^3\pi}{3}$  `// Sphere completely behind plane`

distance of the point to the center into account $d_p = |p - c|$. Moreover, we have to consider, whether the point enters an inside or an outside sphere. In case of outside spheres, we simply set $w_{out} = d_p/r$, in case of inside spheres as $w_{in} = r - d_p/r$. We switch $w$ between $w_{out}$ and $w_{in}$ in case that $w_{out}V = 1 - w_{in}V$ and the points switched between the respective hemisphere.

In case that several points $p_i$ and their normals $n_i$ with $i \in \{1, ..., N\}$ hit the same boundary sphere, we simply compute the normalized average point $p_s$ of all points $p_i$ and the corresponding averaged normal $n_s$ as

$$p_s = \frac{\sum_{i=0}^{N} p_i}{N} \qquad n_s = \frac{\sum_{i=0}^{N} n_i}{N} \tag{3.3}$$

We can then substitute $p$ for $p_s$ and $n$ for $n_s$ in Equation 3.2 to approximate the total penetration volume of $s$ as described in Algorithm 3.1.

### 3.2.1 *Boundary Spheres Collision Detection*

This task is closely related to traditional collision detection methods. Hence, I use a very similar approach. In a pre-processing step I compute a sphere packing as previously mentioned.

However, I additionally create a wrapped *Inner Sphere Tree* hierarchy based on this sphere packing to accelerate collision queries, similar to [57] (see Figure 3.2c). I use a typical recursive traversal scheme to find the boundary spheres (see Algorithm 3.2). This can be easily performed for all spheres in parallel. A similar approach has been used for distance computations between CAD objects and point clouds in [24].

Obviously, in addition to simply marking the spheres we can directly sum up the collided points and normals per sphere (as described in Equation 3.3), which are required for later collision response calculations.

**Algorithm 3.2**: traverseIST($s$ Sphere, $p$ Point)

```
1: if s is leaf then
2:     mark s as boundary sphere
3: forall children s_j of s do
4:     if p inside s_j then
5:         traverseIST(s_j, p_i)
```

### 3.2.2 *Finding Inside Spheres using Brute Force*

After determining the boundary spheres while performing collision detection, the next step is to determine inside spheres. Outside spheres do not need to be explicitly identified, since they have no effect on the total penetration volume in any case.

The simplest approach to identify inside spheres and compute their respect penetration volumes is a brute force substitution approach. The only difference between boundary spheres to all other spheres is that they collided with one or more points, meaning they have collision information which enables us to compute an exact penetration volume. The goal of my brute force approach is to substitute the missing collision information by borrowing it from suitable boundary spheres.

In Algorithm 3.3, I outline the basic principle. We initialize the global total sums with zeros and start a massively-parallel array of threads, one for each leaf sphere in the inner sphere tree. Now we have two basic cases, either the sphere is a boundary sphere, so it has a collision plane from which we simply calculate the penetration volume. In the other case, the sphere has no collision plane, which we substitute by iterating over all available boundary spheres and calculating the penetration volume with regards to every of those boundary spheres' collision plane. We calculate the overall average of those, weighted by their *priority* in regards to the current sphere. Afterwards, we normalize the weighted average and add it to the global sum. There is no need for explicitly storing the penetration volume, since it will automatically be the magnitude of the force vector.

#### 3.2.2.1 *Simplification*

One simplified variant of this algorithm that I tested was to substitute missing collision information only with the most fitting boundary sphere instead of taking the weighted average. One still has to iterate over all possible boundary spheres and calculate the priority to them, however the more time intensive computations would only have to be done once. I described the whole procedure in Algorithm 3.4.

**Algorithm 3.3**: volumeBFSub($\mathcal{I}$ Inner Sphere Tree)

1: $F_{\text{total}} \leftarrow (0,0,0)^T$
2: $\tau_{\text{total}} \leftarrow (0,0,0)^T$
3: $C_I \leftarrow$ center of mass of $\mathcal{I}$
4: **forall** Leaf Sphere $s \in \mathcal{I}$ **do in parallel**
5: **if** $s$ is Boundary Sphere **then**
6: $p_s \leftarrow$ averaged collision point of $s$
7: $\vec{n}_s \leftarrow$ averaged collision normal of $s$
8: $V \leftarrow \text{intersectedVolume}(s, p_s, \vec{n}_s)$
9: $F \leftarrow \vec{n}_s V$
10: $\tau \leftarrow F \times (p_s - C_I)$
11: $F_{\text{total}} \overset{*}{\leftarrow} F_{\text{total}} + F$
12: $\tau_{\text{total}} \overset{*}{\leftarrow} \tau_{\text{total}} + \tau$
13: **else**
14: $F_{s'} \leftarrow (0,0,0)^T$
15: $\tau_{s'} \leftarrow (0,0,0)^T$
16: $w_{\text{denom}} \leftarrow 0$
17: **forall** Leaf Sphere $s'$ in Boundary of $\mathcal{I}$ **do**
18: $w \leftarrow \text{priority}^{\dagger}$ of $s'$ in relation to $s$
19: $V \leftarrow \text{intersectedVolume}(s', p_{s'}, \vec{n}_{s'})$
20: $F \leftarrow \vec{n}_{s'} V$
21: $\tau \leftarrow F \times (p_{s'} - C_I)$
22: $F_{s'} \leftarrow F_{s'} + wF$
23: $\tau_{s'} \leftarrow \tau_{s'} + w\tau$
24: $w_{\text{denom}} \leftarrow w_{\text{denom}} + w$
25: **if** $w_{\text{denom}} \neq 0$ **then**
26: $F_{\text{total}} \overset{*}{\leftarrow} F_{\text{total}} + F_{s'}/w_{\text{denom}}$
27: $\tau_{\text{total}} \overset{*}{\leftarrow} \tau_{\text{total}} + \tau_{s'}/w_{\text{denom}}$

*Atomic add, since this is a concurrency hazard because $F_{\text{total}}$ and $F_{\text{total}}$ are global variables.

**Algorithm 3.4**: volumeBFSubSimple($\mathcal{I}$ Inner Sphere Tree)

1: $F_{\text{total}} \leftarrow (0,0,0)^T$
2: $\tau_{\text{total}} \leftarrow (0,0,0)^T$
3: $C_I \leftarrow$ center of mass of $\mathcal{I}$
4: **forall** Leaf Sphere $s \in \mathcal{I}$ **do in parallel**
5: $\quad w_{\text{best}} \leftarrow -\infty$
6: $\quad s_{\text{best}} \leftarrow nil$
7: $\quad$ **forall** Leaf Sphere $s'$ in Boundary of $\mathcal{I}$ **do**
8: $\quad\quad w \leftarrow 0$
9: $\quad\quad$ **if** $s = s'$ **then** $w \leftarrow \infty^{\ddagger}$
10: $\quad\quad$ **else** $w \leftarrow$ priority$^{\dagger}$ of $s'$ in relation to $s$
11: $\quad\quad$ **if** $w > w_{\text{best}}$ **then**
12: $\quad\quad\quad w_{\text{best}} \leftarrow w$
13: $\quad\quad\quad s_{\text{best}} \leftarrow s'$
14: $\quad p_{s_{\text{best}}} \leftarrow$ averaged collision point of $s_{\text{best}}$
15: $\quad \vec{n}_{s_{\text{best}}} \leftarrow$ averaged collision normal of $s_{\text{best}}$
16: $\quad V \leftarrow \text{intersectedVolume}(s', p_{s_{\text{best}}}, \vec{n}_{s_{\text{best}}})$
17: $\quad F \leftarrow V\vec{n}_{s_{\text{best}}}$
18: $\quad \tau \leftarrow F \times (p_{s_{\text{best}}} - C_I)$
19: $\quad F_{\text{total}} \xleftarrow{+}{}^{*} F$
20: $\quad \tau_{\text{total}} \xleftarrow{+}{}^{*} \tau$

*Atomic add, since this is a concurrency hazard because $F_{\text{total}}$ and $F_{\text{total}}$ are global variables.

I compared both approaches by evaluating the same recorded haptic movement and Kinect environment with both approaches.

As expected, the computation time is reduced drastically, a speedup of about 1.7 on average is achieved (see Figure 3.3). This can be explained firstly by the reduced divergence, since we do not have to tread boundary spheres explicitly different, instead they are just assigned the maximum priority, which is a huge benefit for the GPU's SIMD architecture. Another reason is the reduction in memory accesses, since we do not need to read as much data on every single sphere, just the best fitting one. Global memory writes are as well reduced, to one instance of several writes instead of an instance of writes per boundary sphere.

Surprisingly, I also found that in some scenarios, the simplified approach produces smoother force feedback behaviour. The resulting force feedback of both approaches can be seen in Figure 3.4 when using inverse linear distance as priority. The simplified brute force produces noticeably smoother feedback, with less discontinuous spikes. Whereas with collision count as priority (see Figure 3.5), the more expensive weighted average brute force approach shows visibly better quality with none of the errors the simplified version shows. Overall though, the more expensive weighted average brute force produces better results in most cases.

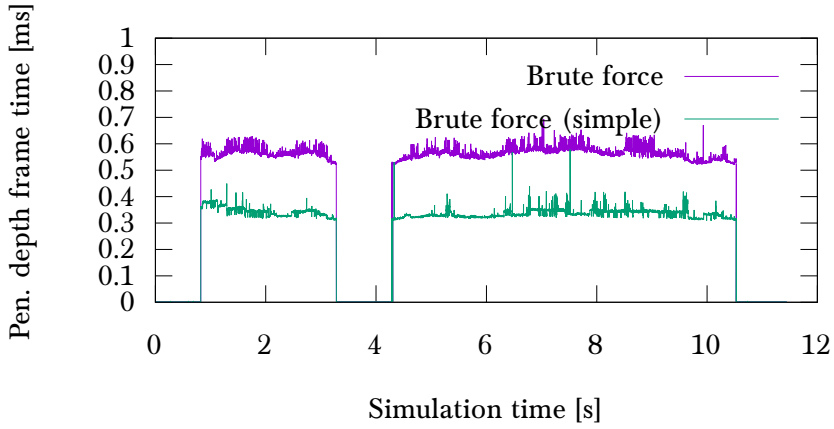
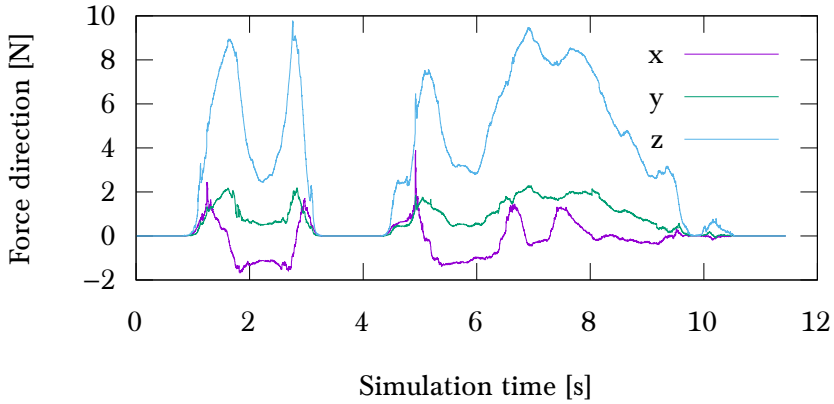
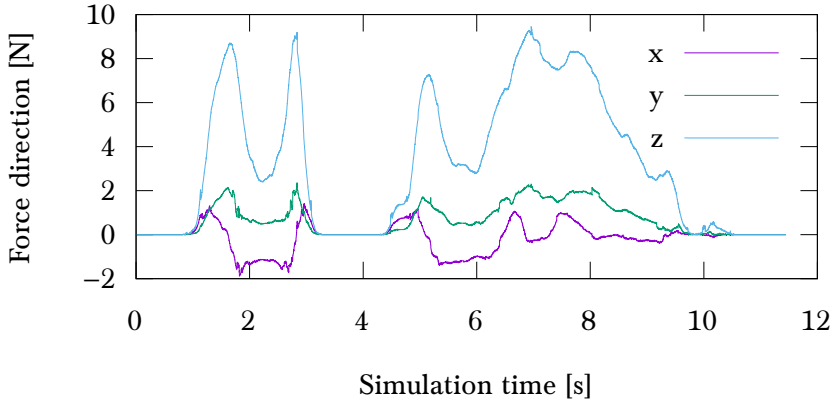



**Figure 3.3:** Interaction scenario with a realistic point cloud to compare computational effort of both algorithms. The simplified variant is significantly faster with a speedup of about 1.7.

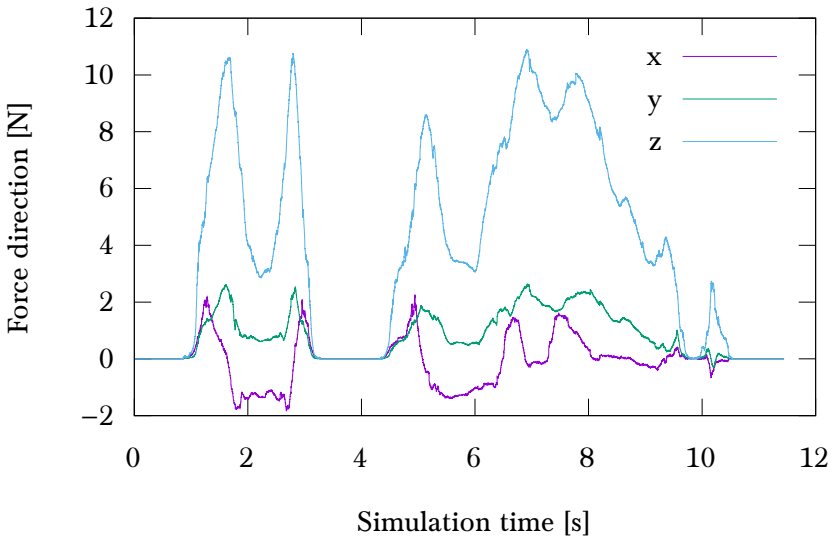



**(a)** Weighted Average Brute Force. Several noticeable spikes that deviate from the general path.

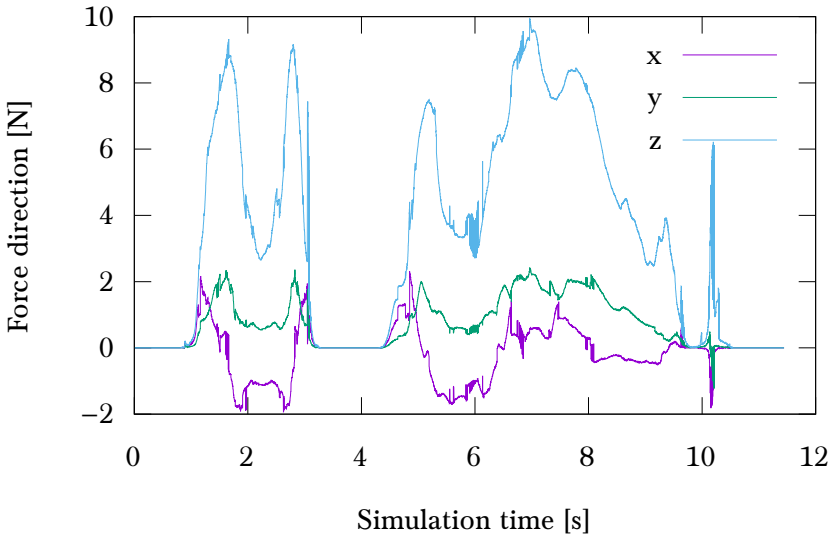



**(b)** Simplified Brute Force. Irregular spikes that are shown on the weighted average brute force are missing or signifcantly less pronounced here.

**Figure 3.4:** Haptic feedback quality comparison of the same run evaluated with the Brute Force and the Simplified Brute Force method. Priority chosen as inverse linear distance between spheres.




**(a)** Weighted Average Brute Force. This priority choice is generally more noise, because it does not take into account the relation of non-boundary spheres to the boundary spheres they are weighted by. Instead, every non-boundary sphere will weight the same boundary sphere exactly the same.




**(b)** Simplified Brute Force. Severe errors are visible, multiple spikes from the general path that no other evaluation method shows. This priority is extremely ill-fitting for the simplification, since this essentially means every non-boundary sphere will use the collision plane of the same single boundary sphere with the most collisions to substitute their missing collision data.

**Figure 3.5**: Haptic feedback quality comparison of the same run evaluated with the Brute Force and the Simplified Brute Force method. Priority chosen as collision count of the boundary sphere.

### 3.2.3 *Finding Inside Spheres using Leaf-Graph*

I also developed an alternative, graph-based algorithm to find inside spheres, that takes into consideration the physical connection between the individual spheres as presentations of volume. The brute force approaches check against any possible sphere that is in the boundary, this can lead to incorrect penetration volumes in some however uncommon cases. For example when the virtual tool is penetrating a point cloud at an edge where part of the virtual tool is lying in a gap in-between points (see Figure 3.6a). When I used the graph traversal to determine the inside spheres, I got the result shown in Figure 3.6b. It is arguable if this is the perfect solution, however I think it is at least much better, since the mannequin's point cloud would continue on that side, if it was actually that large as the inside volume seen in Figure 3.6a suggests.

## 3.3 LEAF-GRAPH

### 3.3.1 *Construction*

We construct the leaf-graph by creating a connected graph over all the leaves of the inner sphere tree. However, I came to the conclusion that we should not intend to construct a minimal graph. Instead, the graph should represent the physical connectivity of the object's volume as close as possible. If we were to construct a minimal graph, we might leave out edges in cases where a sphere has multiple very close neighbors. In Figure 3.7, I included a simplified sketch to illustrate the construction process.

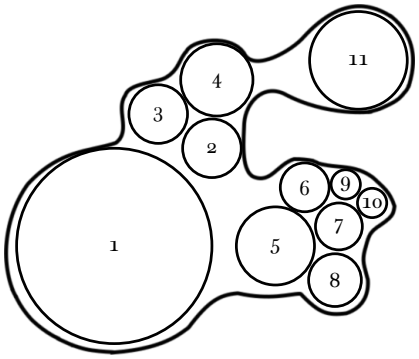
The procedure will be explained in the following. Given is an arbitrary object with a sphere packing of some level of precision. First, we connect the spheres that have a touching contact. Touching contact is however hard to clearly define. Typically, one would solve this with a simple distance test against a suitably small $\varepsilon$. When it is unclear in what scale the inner sphere tree exists, the choice of this $\varepsilon$ is not as trivial as choosing a static value. I defined touching contact as any distance that is smaller than the diameter of the smallest sphere. This does not guarantee a touching contact in the physical sense. It does however guarantee that we do not have an edge that penetrates another sphere, which would be equivalent to skipping physical volume by taking this edge in traversal. Unfortunately, the inner sphere trees created by the Protosphere algorithm are not always connected. Hence, we then continuously insert the shortest bridge of all possible bridges until the graph is 1-connected.

You can see such a leaf-graph for a sphere packing with low resolution of the Stanford bunny in Figure 3.8. A large sphere will typically have many edges because it is near to many other spheres. For us, it is unclear if this is a desirable property or not. It might be worth experimenting with setting a maximum sphere radius when creating the sphere packing.

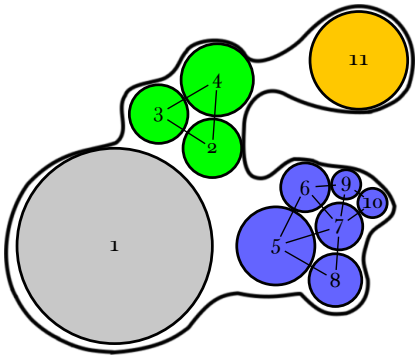
**(a)** Brute force detection of inside spheres (blue filled spheres). An unconnected part of the spiral is found to be inside.

**(b)** Leaf-graph detection of inside spheres. Any part of the spiral that is not connected to the collided part is not found to be inside.

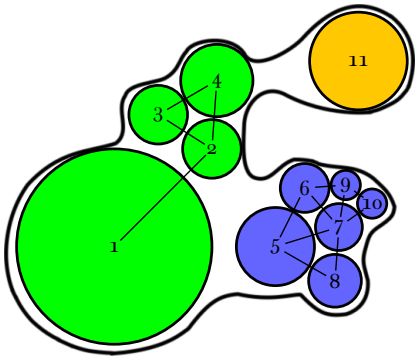
**Figure 3.6**: Difficult arrangement of the virtual tool inside the point cloud, as part of it is not contacting any points. The point cloud is a real recording of a mannequin by a Microsoft Kinect.



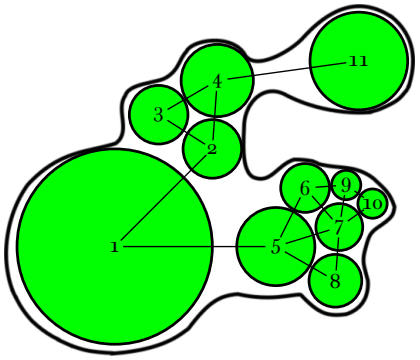

**(a)** The scenario: An arbitrary object that has a sphere packing. Unfilled space and gaps between spheres are exagerrated for simplicity and to make the problem more obvious.




**(b)** All spheres that are touching, meaning their distance is extremely small, we connect with an edge.

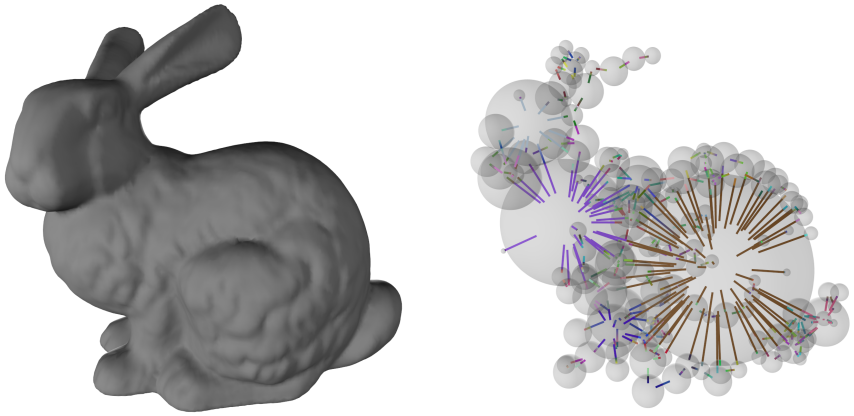



**(c)** Next we repeatedly insert the shortest possible bridge until the graph is connected. Here, sphere 1 and 2 had the shortest distance between them while still being from different connected components.




**(d)** The finished leaf-graph with all connected components being merged to one. The bridges between spheres 1 and 5 and 4 and 11 were inserted.

**Figure 3.7**: Leaf-graph creation procedure for a simple scenario.



**(a)** The original mesh.

**(b)** Leaf-graph. Edges with the same color are incident.

**Figure 3.8**: Low-resolution sphere packing of the Stanford bunny.

#### 3.3.1.1 *Auxiliary Physics Precomputations*

In order to calculate plausible torques we need to precompute physics properties of the given inner sphere tree. Firstly, we need the IST's center of mass in order to process individual forces at each collision point. If we have an inner sphere tree $\mathcal{I}$, then the center of mass is given by

$$C_{\mathcal{I}} = \frac{\sum\limits_{\text{Leaf spheres } s \in \mathcal{I}} c_s (4/3\pi r_s^3)}{\sum\limits_{\text{Leaf sphere } s' \in \mathcal{I}} 4/3\pi r_{s'}^3} \tag{3.4}$$

where $c_s$ and $r_s$ are the center and radius of $s$, and $r_{s'}$ is the radius of $s'$.

Additionally, we need to compute an inertia tensor that represents the inner sphere tree's distribution of volume accurately. First, we calculate the inertia tensor for each sphere $s$ with mass $m_s$ and radius $r_s$ locally:

$$I_s = \begin{bmatrix} \frac{2}{5} m_s r_s^2 & 0 & 0 \\ 0 & \frac{2}{5} m_s r_s^2 & 0 \\ 0 & 0 & \frac{2}{5} m_s r_s^2 \end{bmatrix} \tag{3.5}$$

Then, we use the parallel axis theorem to transform the local inertia tensors to be relative to the IST's center of mass:

$$I_{\mathcal{I}} = \sum_{\text{Leaf spheres } s \in \mathcal{I}} I_s + m_s \begin{bmatrix} R_y^2 + R_z^2 & -R_x R_y & -R_x R_z \\ -R_x R_y & R_x^2 + R_z^2 & -R_y R_z \\ -R_x R_z & -R_y R_z & R_x^2 + R_y^2 \end{bmatrix} \tag{3.6}$$

where $R = C_{\mathcal{I}} - c_s$, the difference of the sphere's center to the IST's center of mass.

### 3.3.2 *Traversal*

After we have found the boundary spheres, in order to find and process all the inside spheres, I developed a two pass algorithm. First, for each the boundary spheres we perform a graph traversal on the sphere graph in order to mark all the inside spheres and collect reference boundary spheres which will be used in the second pass. To perform the actual traversal we will be using a recursive depth-first search. We stop the search if we find either a boundary sphere or a sphere that we already marked as inside (see Algorithm 3.5). We traverse only those edges that are pointing away from the collision normal of the respective boundary sphere. This ensures that we traverse only inside the part of the virtual tool that is inside the point cloud. The traversal can be easily parallelized by traversing all boundary spheres in parallel (see Algorithm 3.6).

Moreover, I tried further optimizing my implementation using CUDA's dynamic parallelism for the first iteration. This helps in cases where the penetration depth is deep, otherwise the overhead of having additional kernel launches negates the benefits.

**Algorithm 3.5**: traverseGraph($s_b$ Boundary sphere, $s$ Sphere)

```
1: if s is not marked and s is not a boundary sphere then
2:     mark s as inside sphere
3:     store reference of s_b in s
4:     forall edges (s, s_i) do
5:         traverseGraph(s_b, s_i)
```

**Algorithm 3.6**: kernel_graphPass1($\mathcal{I}$ Inner sphere tree)

```
1: forall Boundary sphere s ∈ I do in parallel
2:     forall edges (s, s_i) do
3:         c ← center of s
4:         p_s ← averaged collision point of s
5:         n⃗_s ← averaged collision normal of s
6:         if c behind the plane defined by p_s and n⃗_s then
7:             traverseGraph(s_b, s_i)
```

In a second pass, we iterate over all spheres and calculate the penetration volume and haptic feedback according to the connections found by the traversal in the first pass (see Algorithm 3.7 for details on the second pass).

Besides the fact that this approach honors the physical connectivity of the volume, it also performances better in cases of shallow penetration depth (see Figure 3.9).

**Algorithm 3.7**: kernel_graphPass2($\mathcal{I}$ Inner sphere tree)

```
1:  F_total ← (0,0,0)^T
2:  τ_total ← (0,0,0)^T
3:  C_I ← center of mass of I
4:  forall Leaf sphere s ∈ I with reference to s_b do in parallel
5:      p Point
6:      if s is a boundary sphere then
7:          p ← averaged collision point of s
8:          n_s ← averaged collision normal of s
9:          V ← intersectedVolume(s, p, n_s)
10:         F ← n_s V
11:     else
12:         if s has any reference to boundary sphere s_b then
13:             V ← 4/3πr^3                    // Full sphere volume†
14:             p ← center of s
15:     τ ← F × (p − C_I)
16:     F_total ←* F_total + F
17:     τ_total ←* τ_total + τ
```

*Atomic add, since this is a concurrency hazard because $F_{\text{total}}$ and $\tau_{\text{total}}$ are global variables.

†Another possibility would be to select the best fitting boundary sphere in pass one and calculate the penetration volume per inside sphere based on the best boundary sphere's collision information here.

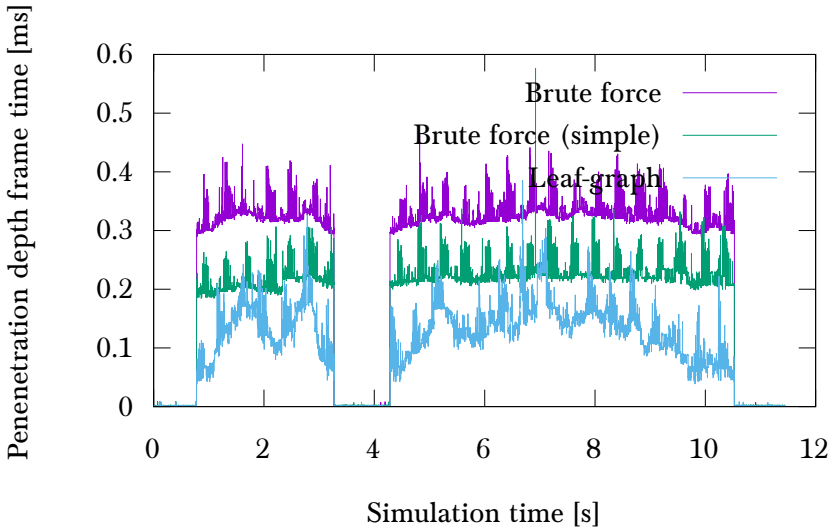



**Figure 3.9**: Shallow penetration test scenario. The virtual tool is scraped against a real recorded point cloud. The leaf-graph shows greater variance (related to penetration depth), but better average computation times compared to both brute force approaches.

Even in non-shallow cases where the penetration depth is up to 25 % and the IST has a small accuracy of around 0.5 k spheres, I usually get better performance than with the simplified brute force approach (see Figure 3.10a), whereas the weighted averaged brute force is even slower until about 32 % penetration depth. This behaviour is amplified with increasing inner sphere tree accuracies of 5 k or 10 k (see Figures 3.10b and 3.10c), leading to the leaf-graph algorithm performing best of all until about 40 %.

#### 3.3.2.1 *Challenges*

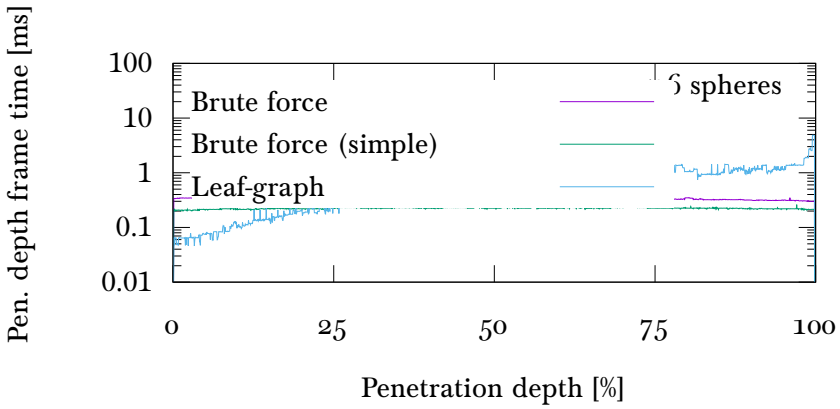
There are however several issues with the graph approach. For one, it is less suited for the GPU's SIMD architecture, since the traversal introduces a lot of thread divergence. Additionally, the traversal can only start from a boundary sphere, so the thread count is also bound by the amount of colliding spheres. Both of these facts contribute to this approach's worse performance in certain scenarios (for example general very deep penetration).

Another example would be a collision in very few or even just one sphere, this results in very few threads having to traverse the whole graph. I even experienced crashes because the memory that was reserved for the stack ran out on the GPU. This comes from the fact that recursion for the massively-parallel architecture is not ideal, unless the function arguments that need to be stored on the stack are very few or the recursion depth can be estimated before-hand. In our case, the recursive function takes only a single address as an argument, however the recursion depth, in the worst case can be as many as the IST has leaves. I only experienced crashes when working with ISTs that are filled with about 10 k or more spheres, and those can be fixed by increasing the memory that is reserved for the stack in the initialization of the CUDA context. However, this shows that in general, the approach is not as suited for a GPU implementation as the brute force approaches. I want to explore more options in the future regarding alternative traversal algorithms, that better take advantage of the GPU's architecture. For example stack-less tree traversal is a method used for GPU-based implementations of ray tracing, which might be applicable to this problem as well.

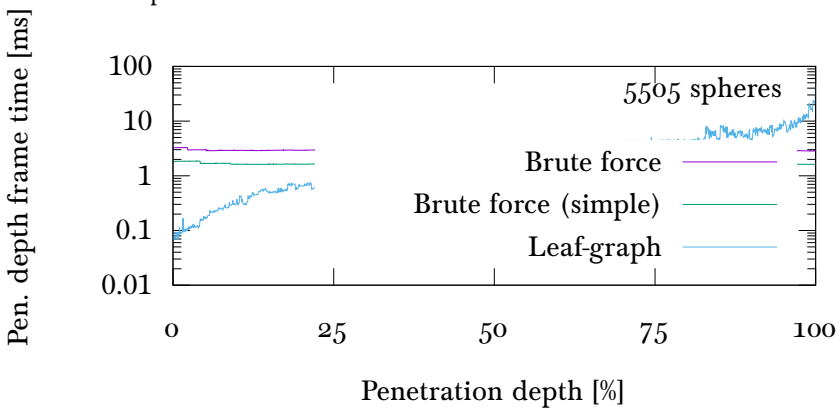
## 3.4 SURFACE ESTIMATION

As the Kinect does not provide any surface information, we have to determine normals for the points generated by the depth sensor. I use the simple observation, that we have knowledge about the viewing direction of the sensor and about the neighborhood of points in the depth image, as it is an ordered point cloud.

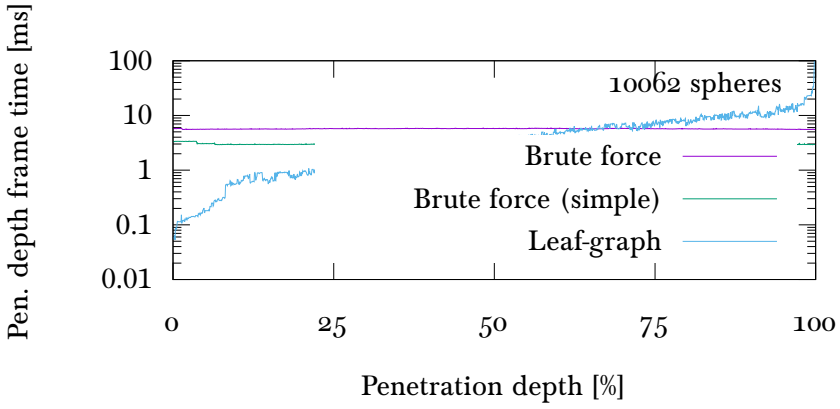
Basically, I use a method that relies on fitting a plane through neighboring points using principal component analysis. More precisely, to com-




**(a)** IST resolution of 536 spheres: As long as less than 25 % of the object is penetrated, the graph approach consistently takes less time to calculate the penetration volume than the simplified brute force.




**(b)** IST resolution of 5505 spheres: The leaf-graph algorithm is the most efficient at selecting the inside spheres until the object is about 40 % of the object is penetarted.




**(c)** IST resolution of 10062 spheres: The leaf-graph algorithm is the most efficient at selecting the inside spheres until the object is about 40 % of the object is penetarted.

**Figure 3.10**: Deep penetration test scenario. The virtual tool completely penetrates a synthetic point cloud in a linear motion path. The leaf-graph method shows a direct correlation between penetration depth and computation times. With increasing IST resolutions the graph algorithm behaves increasingly well in non-shallow penetration depths.

pute the normal $n_p$ for point $p$, we consider its neighborhood of points $Q_p$ by defining the matrix:

$$M_p = \sum_{q \in Q_p} \left(q - c_{Q_p}\right) \left(q - c_{Q_p}\right)^T \tag{3.7}$$

where $c_{Q_p} = \frac{\sum_{q' \in Q_p} q'}{|Q_p|}$, this is simply the mean of the neighborhood. Then, we compute the eigenvector $n_p$ of $M_p$ that corresponds to the smallest eigenvalue of $M_p$. Actually, the plane can have two different normals, so we take the one that points towards the origin because a camera can only see surfaces that point towards it.

$$n_p := \begin{cases} -n_p & \text{, if } n_p \cdot p > 0 \\ n_p & \text{, if } n_p \cdot p \leq 0 \end{cases} \tag{3.8}$$

This algorithm can be easily parallelized by simply starting a thread for each point.

### 3.4.1 *PCA Modifications*

To improve our estimated surfaces, I looked at the state-of-the-art methods that are available to achieve better results. I implemented several modifications of the PCA method which were presented in [23].

#### 3.4.1.1 *Anchoring*

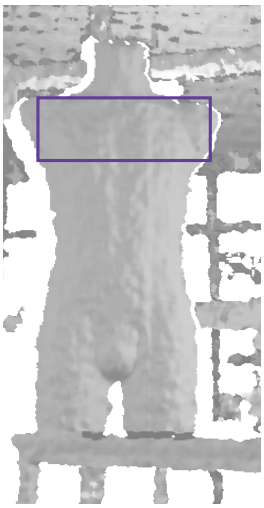
This includes anchoring of the plane origin at the reference point instead of the neighborhood mean. This means our covariance matrix is simplified to

$$M_p^A = \sum_{q \in Q_p} (q - p)(q - p)^T \tag{3.9}$$

The only benefit of this modification is the reduced computational cost to calculate neighborhood means. However, this does also introduce quite some visibly erroneous surface normals in all cases (see Figure 3.11). As this introduces significant errors, I do not enable this modification by default. However, it can still be useful to have a flag for this because of the performance boost, if the more expensive algorithm is not fast enough. Especially when working with streaming point clouds, performance can be an issue. Our implementation is fast enough without this, which is why I do not enable this modification by default.

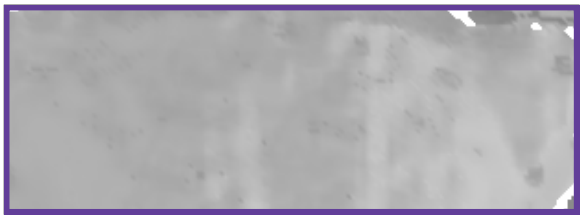
#### 3.4.1.2 *Weighting*

Since we are working with ordered point clouds, two points can be extremely far away, while still being considered neighbors by their 2D grid



**(a)** The complete scene that was recorded, a mannequin standing on a table.

**(b)** No anchoring, plane is expected to pass through the neighborhood mean. Few visible errors.



**(c)** Anchoring enabled, plane is expected to pass through reference point. The lighting shows multiple visibly erroneous surface normals on the mannequin's chest.

**Figure 3.11**: Visual comparison of surface normals by using them for lighting a white textured mesh that was constructed from the ordered point cloud.

location. My initial solution to this problem was having a simple threshold for how far neighbors can be away from each other for them to influence the plane fitting of each other. The problem with this approach however is that a point that is just barely inside the threshold will be weighted exactly the same as a point that is right next to the reference point, which intuitively seems to be an unfair fitting method. This modification solves this problem by using exponential weighting of neighboring points by their euclidean distance to the reference point. The covariance matrix would be modified to

$$M_p^W = \sum_{q \in Q_p} e^{-\frac{||p-q||_2^2}{2\sigma_w^2}} \left(q - c_{Q_p}\right)\left(q - c_{Q_p}\right)^T \tag{3.10}$$

with $\sigma_w$ being a weighting constant that determines how slowly the distance affects the overall weight.

#### 3.4.1.3 *Normalization*

A possible problem that was mentioned by [23] is that the difference vectors that are used for the covariance matrix will be shorter for neighbors that are closer than of those that are farther away. The corresponding covariance matrix is given by

$$M_p^N = \sum_{q \in Q_p} \frac{\left(q - c_{Q_p}\right)\left(q - c_{Q_p}\right)^T}{||q - c_{Q_p}||_2^2} \tag{3.11}$$

I did not find much difference with or without this modification, which might be related to the fact that we are operating on ordered point clouds where distances not as random as in unordered ones. Since there seems to be little to no benefit to this, I do not enable it by default to do without the additional overhead, however small it may be here.

Any combination of the above modifications is possbile and is straight forward enough to implement.

### 3.4.2 *Depth Noise Reduction*

The depth data received from the Kinect camera can be very noise depending on the usecase (factors like lighting, scenery and materials of objects play a role). I implemented a simple bilateral filter for the depth map that is closely related to [52].

I weight each point by two gaussian functions applied to each of its neighbouring points (in the ordered depth map, before transforming to cartesian world coordinates). Let $p$ be a point in cartesian world coordinates with $i(p)$ being its Kinect image space coordinates and $i(p)_x$ and $i(p)_y$ are its 2D components, $i(p)_d$ is its raw depth value. Let $Q_p$ be the neighborhood of $p$ including itself inside the depth map, then we adjust $p$ by

$$p = \frac{\sum_{q \in Q_p} i(q)_z e^{-\frac{1}{2}(\frac{d(p,q)}{\sigma_d})^2} e^{-\frac{1}{2}(\frac{\delta(p,q)}{\sigma_s})^2}}{|Q_p|} \tag{3.12}$$

where

$$d(p,q) = \sqrt{(i(p)_x - i(q)_x)^2 + (i(p)_y - i(q)_y)^2}$$
$$\delta(p,q) = ||(i(p)_d - i(q)_d)||_2$$

As closeness function I chose the gaussian weighted distance between the neighbouring points in image space. I chose image space over cartesian world space because camera transformation weights points by their depth value, which led to unevenly skewed smoothing results. The similarity function is simply the difference in depth values with a gaussian weight as well.

## 3.5 RECOGNIZING EDGES IN ORDERED POINT CLOUDS

In depth images I often encounter the problem that there usually are parts of the image with huge gaps in-between points. Ideally, one would of course want to have a tight weave of points with next to no gaps in-between neighboring points that. This would provide a reliable description of the environment's surface.

Fortunately, these gaps are not random usually. Instead, they stem from the simple fact that the depth camera can only see the front of any object, so the back and sides of any object will have no surface description. See Figure 3.12 for an example scene with the corresponding bird's

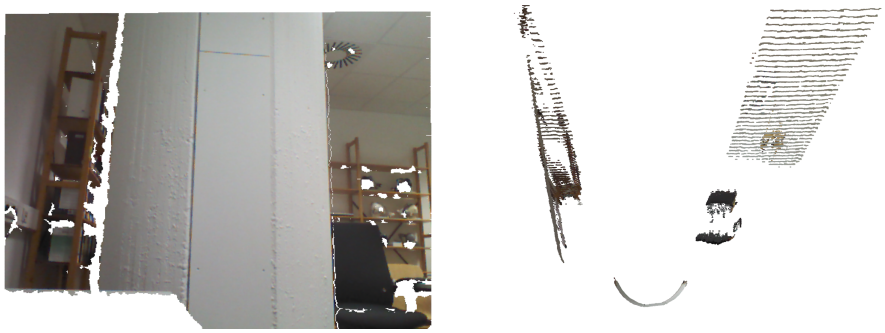
eye view to show the depth changes. When processing a new depth map to convert to a point cloud and it's surface normals, afterwards I additionally run a discrete Laplace operator over the depth map to generate the change in depth per point given as

$$l_{p_z} = \frac{\sum_{q \in Q_p} w_q |i(p)_d - i(q)_d|}{|Q_p|} \tag{3.13}$$

with $w_q$ being the weight of $q$ based on its relative position in image space from the reference point $p$. As I did not have a special use-case in mind and I don't need our value to be normalized to stay in the range of the depth values I chose the following unbiased & simple kernel:

$$D^2_{xy} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.14}$$

This of course means we weight every difference equally, so we can omit $w_q$ in equations 3.13, 3.15 and 3.16 in the implementation. We did still include it here for the sake of completeness.



**(a)** Camera perspective render of the scene. Note the pillar in the foreground occluding a lot of the room.

**(b)** Orthographic bird's eye view of the same pillar scene. Depth value are rendered along the $y$-axis, $x$ remains $x$, the original $y$-values are not visible for clarity. Note the huge gaps between the pillar and the background.

**Figure 3.12**: Pillar scene recorded with a Microsoft Kinect comparing the camera perspective render with a orthographic bird's eye view render to showcase the gaps in the point cloud. This problem occurs in most point clouds which are recorded from one perspective.

However, we don't simply store the found Laplace value $l_{p_z}$, we also want to later know in what 3D direction the depth change occurred. To achieve that, we additionally find $l_{p_x}$ and $l_{p_y}$ as

$$l_{p_x} = \frac{\sum_{q \in Q_p} w_q |i(p)_d - i(q)_d| \left(i(p)_x - i(q)_x\right)}{|Q_p|} \tag{3.15}$$

$$l_{p_y} = \frac{\sum_{q \in Q_p} w_q |i(p)_d - i(q)_d| \left(i(p)_y - i(q)_y\right)}{|Q_p|} \tag{3.16}$$

With this we get a 2D vector that points in the direction with the most depth change, and it's magnitude shows the Laplace value.

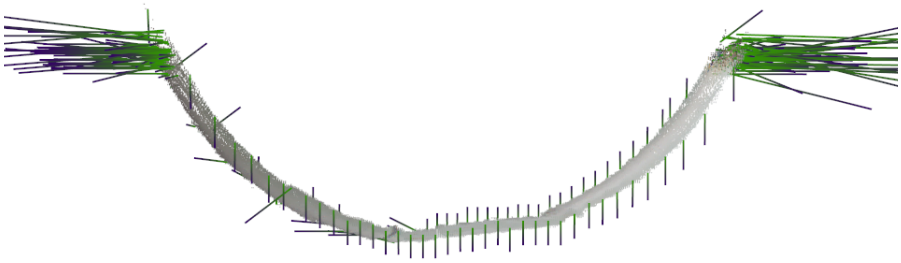


**Figure 3.13**: Cutout of the pillar scene from figure 3.12b. This is a visualization of an intermediate step, where each point of the point cloud has a depth-change vector associated to it. The green-purple colored lines visualize those vectors. Green is the start, purple the end of the line. The vector direction is the direction in which the depth change was recognized, its magnitude indicates the value of depth change that occurred. The left end of the pillar shows great depth change in the left direction, which indicates an edge. Analogously, the right side shows great depth change in the right direction, indicating another edge.

In Figure 3.13, we included a screenshot of a debug view of the scene described in Figure 3.12a with the same rendering settings described in 3.12b. The Laplace results are displayed in the form of vectors that start at the point they are associated with. By this, we implicitly define a plane for Laplace value, since we have a point origin $p$ and we use the previously defined vector components to define the normal $l_p = (l_{p_x}, l_{p_y}, 0)^T$.

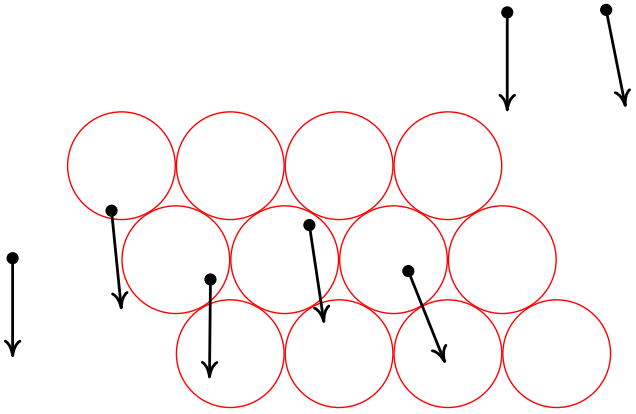
When all of this data is computed we can use it to fit an edge through all the Laplace planes that were accumulated by averaging the Laplace planes to all points $C_p$ and their associated Laplace vectors $C_l$ that collided with the virtual tool this frame by simply averaging them. The detected edge's plane with origin $p_l$ and normal $n_l$ is then given by

$$p_l = \frac{\sum_{p \in C_p} p}{|C_p|} \qquad n_l = \frac{\sum_{l_p \in C_l} l_p}{|C_l|} \tag{3.17}$$

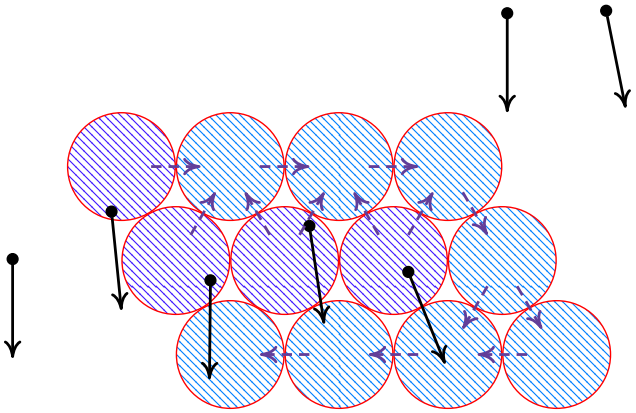
This additional plane can then be used to check against when traversing the leaf-graph. In Figure 3.14a we illustrated an example case where there would be no sensible solution found. Instead, what we illustrated in Figure 3.14b would happen, there would be no sphere considered the outside volume.

The corrected behaviour is shown in Figure 3.14c. This is of course a simplification to illustrate the basic principle in which the edge detection works.

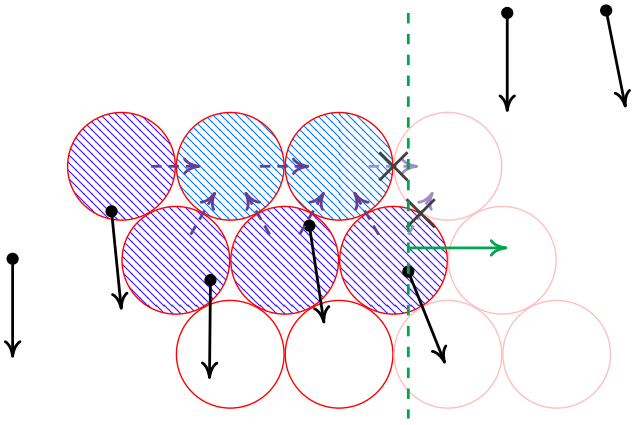
To implement this we simply introduce an additional test when in the recursive traversal. We check if for a sphere $S$ with center $c_S$ and radius $r_S$ whether $n_l \cdot c_S - p_l \cdot \hat{n}_l < 0$ holds. This means the center of the sphere is located behind the plane, in which case traversal can be continued. In



**(a)** The test case setup sketch as seen from the bird's eye perspective. The red spheres represent the virtual tool IST, black dots with arrows are point cloud points and their surface normal. There is a larger gap between the 5th and 6th point.



**(b)** The evaluation of the penetrated IST volume with no edge detection. Shading indicates collided or inside volume. The small edges between spheres indicate a possible edge of the graph traversal. The gap in points on the right side causes the traversal to tunnel to the wrong side of the IST, causing almost all of the volume to be considered collided or inside.



**(c)** The large difference in depth between the 5th and 6th point are recognized and plane is fitted through the 5th point, indicated by the blue line. The traversal algorithm will stop at spheres that are in front of the plane. This fixes the tunneling problem, since the gap is no longer a valid path for the leaf-graph traversal.

**Figure 3.14:** Example case to illustrate the tunneling problem when using the naive leaf-graph traversal algorithm.

the other case we simply stop traversal there. Another possibility would be to check for $n_l \cdot c_S - p_l \cdot \hat{n}_l + r_S < 0$, which would instead also rule out spheres which are just intersecting the plane, but are not necessarily completely behind it.

Whichever solution one chooses, this does introduce discontinuity, since from one frame to the next whole spheres could suddenly be considered inside then outside or the other way around. A solution to this problem is to calculate the volume that was intersected by the edge plane. However, for spheres that actually collided with points are already intersected and cut off by their respective collision plane. To calculate the volume that results from a sphere that is being intersected by two arbitrary planes would probably greatly increase the computational effort of the overall algorithm, which is why we did not look into this issue anymore.

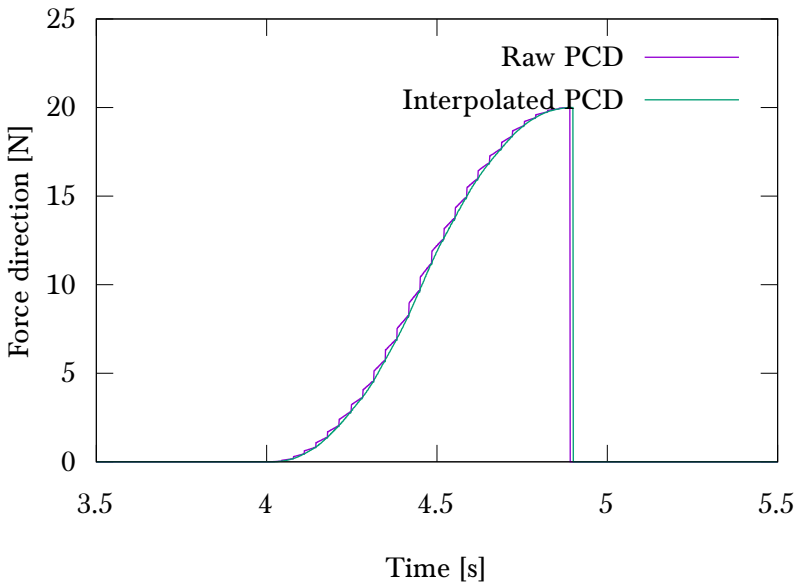
## 3.6 DISCONTINUITY IN POINT CLOUD STREAM

A big challenge we found, was the slow update rate of the range sensor, which emphasizes the discretization of the input data. For the Microsoft Kinect, we are given 30 FPS, so about 33 ms between a frame. Of course a lot can happen in this time, such as moving objects change their position or orientation. This is a major problem when our goal is to achieve continuous forces, this is simply impossible if the input data is already discontinuous. This means it is essential to find a good solution to this problem.

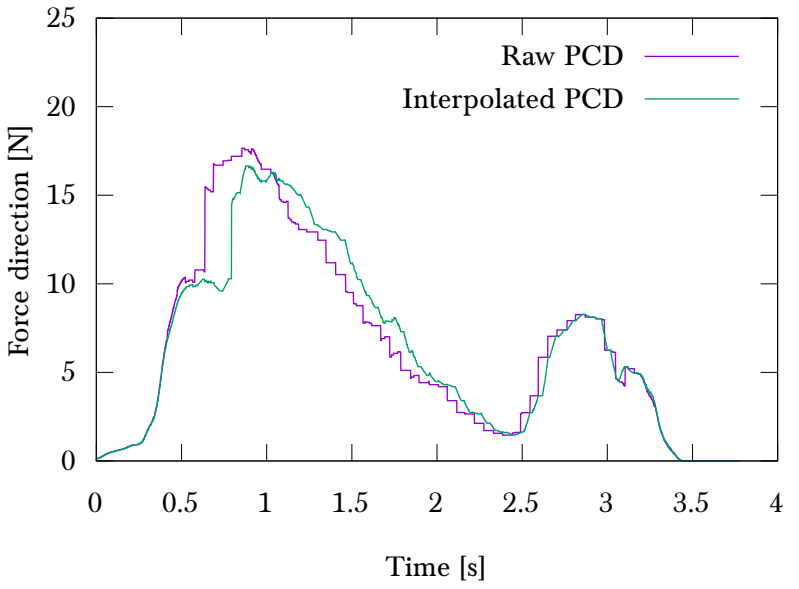
One approach we considered and implemented was having some form of a moving average of the point cloud. We experimented with the simple moving average of 8 frames, as well as with several exponential moving averages with different weighting of older data. This approach simply reduces the change that happens between frames by a certain amount, but it does not solve the core problem that data is only updated every 33th haptic frame, meaning the data is still discontinuous from the view of the haptic rendering. Additionally, the more we reduce the change between frames, the more we have to rely on historic data, which results in a washed out moving picture. So with this approach, it is always a trade-off between having washed out data and having sudden changes in the point cloud. Both states are of course not desirable.

An approach we came up with that we found to have better results is a simple interpolation of the point cloud that is done every time before we calculate new haptic feedback. It sounds like this would be a lot of additional work, however since we have to transform every single point in the point cloud anyway to apply the haptic device's position and orientation anyway, we might as well do a bit of extra work to do a linear interpolation before we transform, the write access is needed in any case.

The interpolation is done by always storing the last two point clouds and estimating the delay until we will get a new frame. If the last frame




**(a)** The setup was a synthetic point cloud wall moving towards the virtual tool, which itself was moving towards the wall. Without interpolation, there are noticeable steps every 33th frame, from the sudden point cloud update.




**(b)** The setup was a real recorded point cloud of a wall that was moving (we moved the camera) back and forth, the virtual tool was only moving slightly. Results are similar to that of figure 3.15a, steps of point cloud updates are signifcantly reduced.

**Figure 3.15**: Force experiment to compare force feedback with PCD interpolation turned on and turned off.

was received at time $t_L$, we currently have time $t_C$ and the estimated frame-delay is $t_D$, then our interpolation factor is given by

$$t = \frac{t_C - t_L}{t_D} \tag{3.18}$$

We simply interpolate every point in world space from the previous frame towards its new value by this factor. However, we don't interpolate values when either of the values are zero, or the distance exceeds a certain threshold. This is done mainly because the Kinect often has missing depth values which will be assigned to zero, even though it does not match the real scenery (see Figure 3.16).

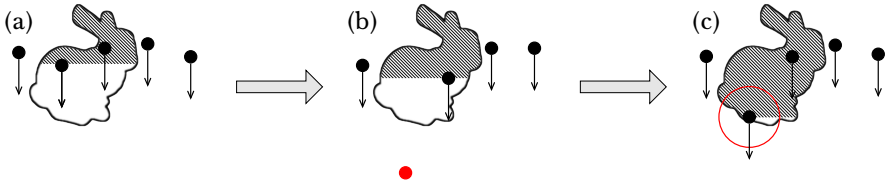



**Figure 3.16**: Example case in which general interpolation would cause an error. (a) and (b) are raw point clouds, in (b) is a red zero depth point (happens very frequently). (c) is a possible interpolation between (a) and (b). If we regard zero depth points we risk getting non-linear interpolation of the resulting volume.

In Figure 3.15a and 3.15b we demonstrate the benefit of the interpolation. The raw force plots show noticeable steps from the slowly updating Kinect depth data. When we turn on interpolation of the point clouds in both scenarios we get force plots that are noticeably smoother and most of the sharp steps are completely gone. However, the plots also show that the interpolated data is always behind the real point cloud by a maximum of one frame time, since we do need to know the target we are interpolation towards before we can start interpolation. As far as we could find, this is the best solution to this problem, since we get updated data for every time we calculate new forces, making the input data continuous.

# 4 IMPLEMENTATION

## 4.1 ARCHITECTURE

In this section I will give a brief overview of my current implementation of my algorithm in general and the use case in particular. I have implemented my approach in C++ and CUDA. I used OGRE3D as a scene-graph for visualization, CHAI3D to operate a haptics device, namely a Sensable Phantom and finally OpenNI to operate the Kinect. The Phantom supports 6-DOF input but only 3-DOF output. However, the virtual coupling shows a visually correct behaviour also for the torques, even if they are not rendered to the haptic device.

Even if all my algorithms run completely on the GPU, I made heavy use of multi-threading in my implementation. The main reason for this are the different frequencies of the haptic and graphics rendering, but also of the slow Kinect camera. For instance, the Kinect captures a new frame only every 33 ms while the haptic rendering should maintain 1 kHz refresh rate. Moreover, I decided to spent an extra collision detection thread in addition to the haptic rendering thread: even if my collision detection is very fast, in case of deep inter-penetrations it may happen that it exceeds the 1 ms computation interval. Hence, in such cases I interpolate the forces for the haptic rendering in an extra thread. I implemented most of the inter-thread communication using double-buffers, in order to avoid explicit synchronization (see Figure 4.1 for an overview).

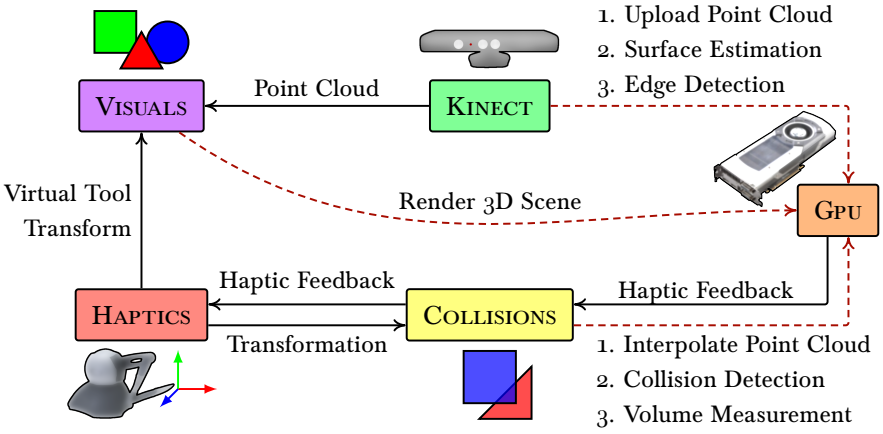



**Figure 4.1**: Thread communication model. Data transfers are visualized as solid black edges and GPU accesses as dashed red edges.

## 4.2 GPU CONCURRENCY

The previously mentioned tasks both benefit from the architecture of the GPU and it's advantages, which is why I implemented both in CUDA. This additionally saves me the data transfer of the resulting surface normals, since the results can simply be written to global GPU memory, which can be used by the other tasks in execution.

However, in principle I want to occupy the GPU as much as possible with the execution of the haptic rendering algorithm, since this will make feedback more responsive. The challenge is that I want to utilize the GPU for the surface estimation while blocking the execution of the rendering algorithm as little as possible.

CUDA has the ability to perform concurrent kernel execution with the right hardware. However, it is not very reliable from my experience, if the two kernels occupy too many of the same resources for example, concurrency will simply not happen, or very infrequently. Of course, in a production setting, one would simply install two GPUs and share the results of surface estimation with the other GPU. I did not have this option, so I decided to implement a workaround that will not solve but reduce the impact that the concurrent kernels have on one another.

In practice, if surface estimation is running and a new haptic rendering frame is lined up to be performed on the GPU, that execution thread will simply wait until all previous kernels have finished in most cases. Since my implementation of surface estimation takes on average about 3 ms, I want to avoid having haptic rendering frame be delayed by 3 ms before it even starts its work, resulting in about 4 ms of potential frame time, which exceeds the 1000 Hz margin by far. It is arguable if this kind of delay every 30th frame would be noticeable, but I still tried to improve on it.

I split up the surface estimation in smaller chunks of work and launch a kernel for each chunk. However, when I do this naively, there is no synchronization between GPU and CPU by default, so I would instantly fill up the stream with the equivalent of the complete task, it would just be split up in multiple chunks. This is not desirable, since all kernel launches for the individual chunks will be issued before a haptic frame has enough time to utilize the GPU in-between.

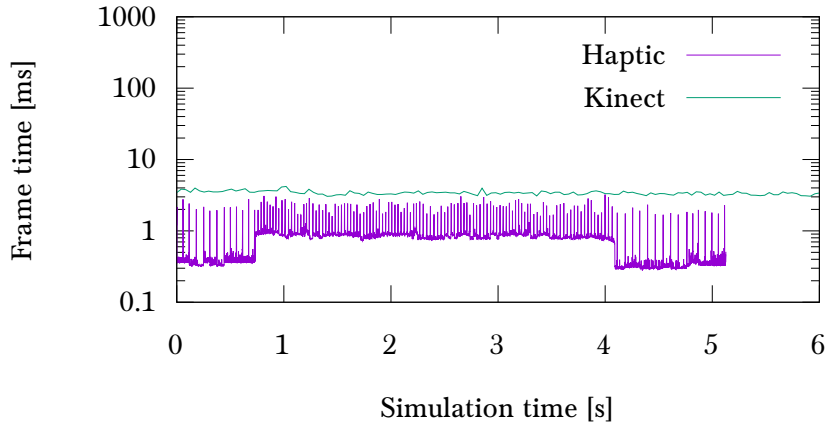
Having explicit stream synchronization and sleeping the CPU thread that will issue the next chunk's kernel launch for a short time gives enough time for other kernels to take priority, because they are issued before the next chunk of surface normals.

I decided to implement a solution that would automatically maximize the sleep times to make the surface estimation as slow as it can be to keep up with the Kinect frequency. I want to of course avoid slowing down the calculations so much that we do not render every available Kinect frame, so I aim for a maximum frame time of 33 ms.

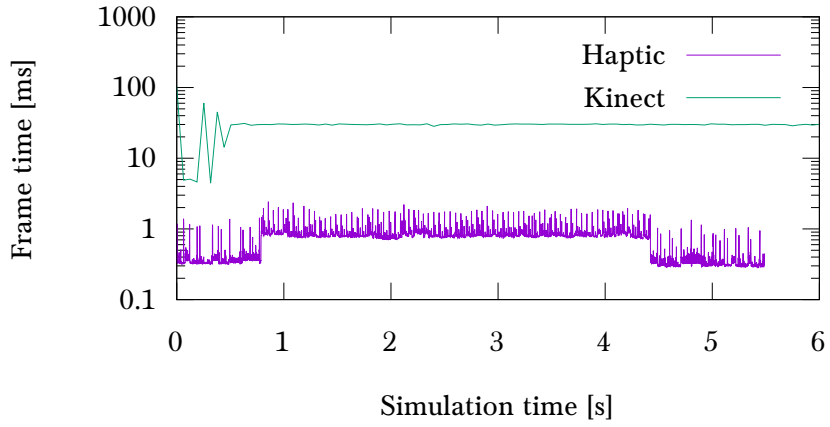
We calculate the total time available to be slept in-between chunks as

$$t_T = \max\{\min\{t_I - t_\lambda, t_I\}, t_M\} \tag{4.19}$$

where $t_I$ is the ideal time a frame would take, chosen as 33 ms in the case of the Kinect. $t_\lambda$ is the estimated time the actual work takes, so that we can fill up the difference with sleep time. We continuously update the estimate $t_\lambda$ in the form of a simple moving average, so that we have a reliable and stable estimate. If the previous frame time is $t_{F'}$ we add $t_{F'} - t_T$ to the simple moving average, since this is the effective work that was done last frame. $t_T$ is then evenly spread as sleep in-between kernel launches for all chunks.




**(a)** Kinect thread does not use chunking, so all work is performed as fast as possible. The frame time is fluctuating with high spikes (Mean=0.74, SD=0.37).



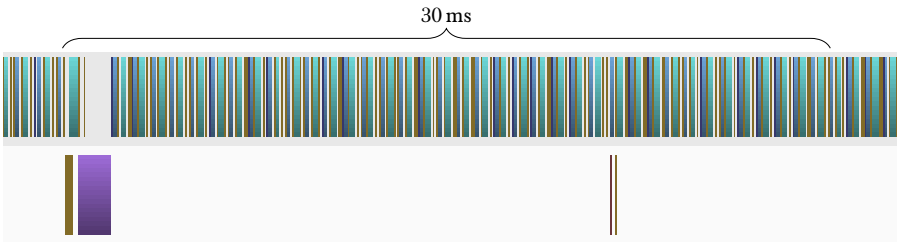

**(b)** Kinect thread uses all of its time by splitting up its work in 16 equal chunks and thereby giving priority to haptic rendering. Frame time spikes are noticeably less pronounced (M=0.69, SD=0.28).

**Figure 4.2**: Comparison of different scheduling approaches.

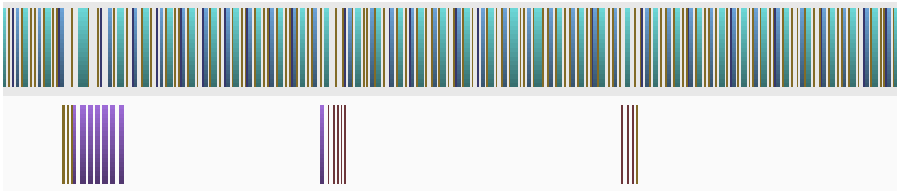
Figure 4.2 shows a comparison of the frame times that are produced by the two different approaches. On one side (see Figure 4.2a), the whole work of the Kinect frame is performed in a single piece, which gives

fast Kinect frame times of around 3 ms. However, the spikes of the haptic frame times are very high here, since in unfortunate scheduling the haptic thread will be blocked for multiple milliseconds. The plot on in Figure 4.2b side shows that the Kinect frame time is always around 30 ms, which is close to the maximum allowed frame time. The fluctuation of Kinect frame times in the beginning comes from the fact that I use a moving average estimate, which only becomes representative after the first few starting frames. The spikes in the haptic frame time are noticeably smaller.

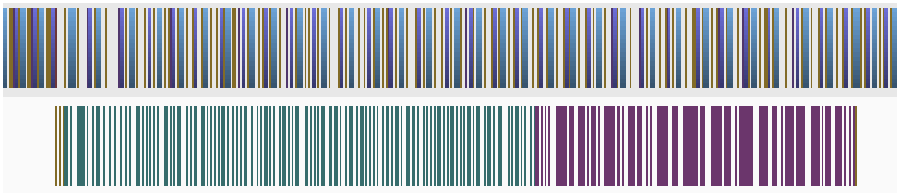
Figure 4.3 shows a debugging time-line of all kernel launches over about one Kinect frame in with different chunking settings. These time-lines also show, the chunking significantly reduces the blocking of the haptic rendering task.




**(a)** Normals are processed in one single pass. The whole Kinect frame is processed in a fraction of the available 33 ms while blocking the haptic rendering for the whole duration.



**(b)** Normals are processed in 8 evenly split chunks of input data. The problem visible in (a) is already significantly better.



**(c)** Normals are processed in 128 evenly split chunks of input data. Haptic rendering is not blocked for any significant amount of time because the Kinect work is evenly spread across most of the available 33 ms

**Figure 4.3:** Exemplary kernel threading overview. NVIDIA Visual Profiler view of kernel launches over time on each stream. Top stream is haptic rendering tasks, bottom stream are point cloud related tasks. Note that kernel activity when it is being recorded shows unusual behaviour because of the additional debugging overhead, which is why actual results are better than can be shown here.

## 4.3 VISUALIZATION

To visualize the results of the algorithm I implemented drawing of dynamic spheres that are colored according to their state. Exemplary scenes can be seen in Figure 4.4.

### 4.3.1 *Dynamic Drawing of Spherical Caps*

While I was developing the debugging capabilities of my software I wanted to draw dynamic spherical caps. I was surprised to find that there seem to be very little resources on solving this problem, so I will present the way I solved this problem.

What we want to visualize here is a portion of a sphere that cut off by an intersecting plane (a collision plane). For that we need to calculate triangles to represent the surface of the spherical cap appropriately.

The first step is to calculate the intersection circle, given the intersecting sphere's center $S$, its radius $r$ and the collision plane's origin $O$ and it's normal $\vec{n}$. If $|\vec{n} \cdot S - \vec{n} \cdot O| > r$ holds we have an intersection with the following center and radius:

$$C = S - \vec{n}(\vec{n} \cdot S - \vec{n} \cdot O) \tag{4.20}$$

$$r' = \sqrt{r^2 - (\vec{n} \cdot S - \vec{n} \cdot O)^2} \tag{4.21}$$

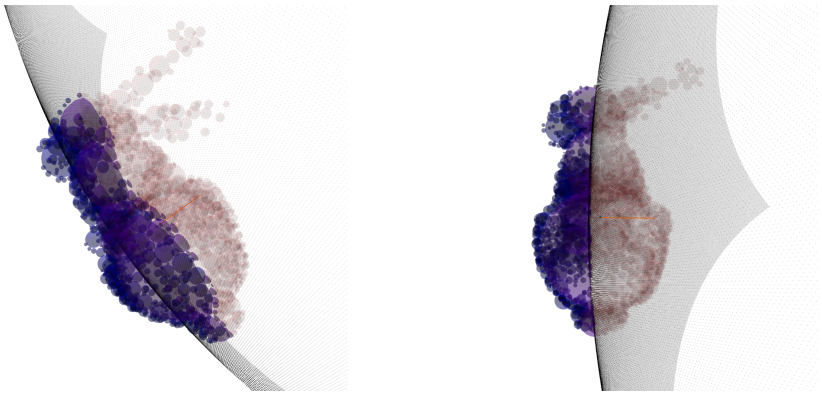
Of course, the normal is equal to the plane's normal $\vec{n}$.

In Figure 4.5, I show an exemplary case for a sphere at $S = (3, 1, 2)$ with radius $r = 3$ and a collision at point $O = (1.14, 1.1, 1.8)$ with normal $n = (-0.52, 0.77, -0.36)$. The corresponding triangle mesh that we get with large interpolation steps can be seen in Figures 4.6 and 4.7.

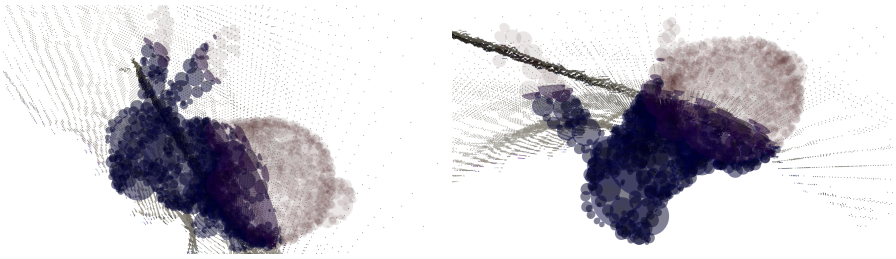
I wanted to have the typical UV-sphere mesh-typology to have evenly spread vertices which result in a smooth surface. Interpolation needs to happen in spherical coordinates, for which we need to have an orthonormal basis that has one axis along the normal of the plane and both other axes reside in the collision plane. We choose $\vec{e_1} = \hat{n}$, $\vec{e_2} = \vec{e_1} \times \hat{v}$, where $\vec{v}$ is any linearly independent vector from $\vec{e_1}$, and finally $\vec{e_3} = \vec{e_1} \times \vec{e_2}$.

With this new basis we can interpolate towards surface points that are at the two poles that are created by the intersection circle in spherical coordinates. For the interpolation step for the segments we have to choose an appropriate $\Delta_\theta$, and for the rings we choose a radian angle step size $\Delta_\phi$.

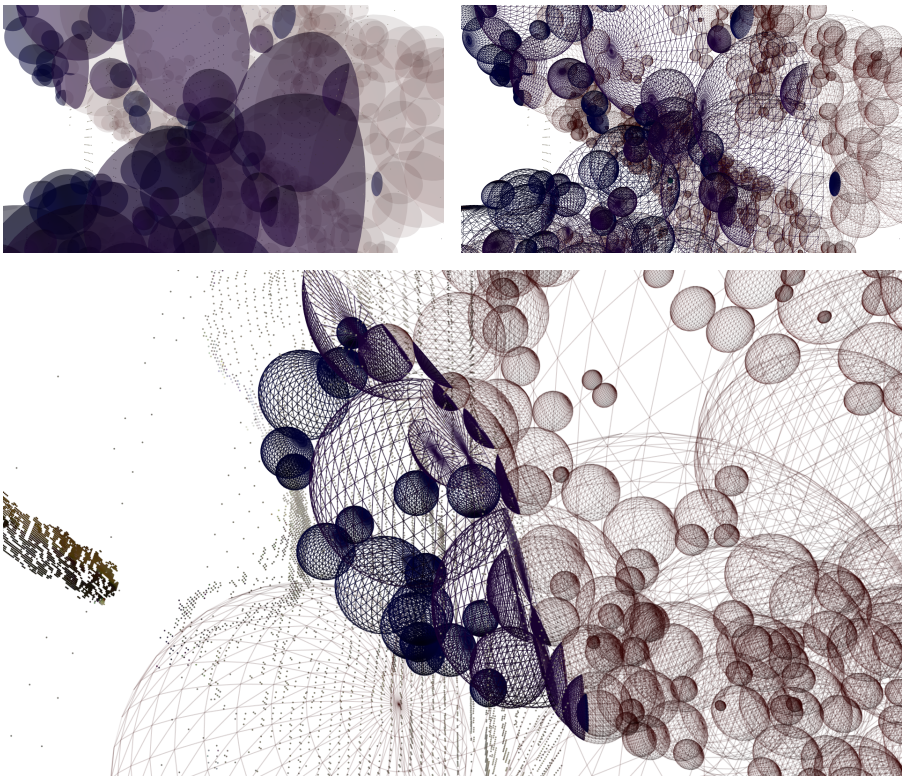
The Algorithm 4.1 shows the process. We first iterate over the segments, where the intersection circle is the first ring that we connect (this will represent the cut surface). We then follow the current segment line and interpolate the $\theta$ angle towards the north pole to get to the next ring of the sphere mesh. We span triangles across the current segment's ring to the next segment's ring until we reach the pole $N$ at $\theta = \pi$. I illustrated the view of one iteration in Figure 4.8.



**(a)** The bunny is penetrating a synthetically generated point cloud.



**(b)** This point cloud was generated from a real recording of a mannequin.



**(c)** Wireframe renderings of a close-up view of the debugging spherical caps generated to visualize the penetrated volume with complete detail.

**Figure 4.4**: The penetration volume of the stanford bunny: Blue spheres represent completely penetrated volume, purple ones are boundary volume, meaning those spheres have collisions. Red spheres are completely on the outside of the environment.

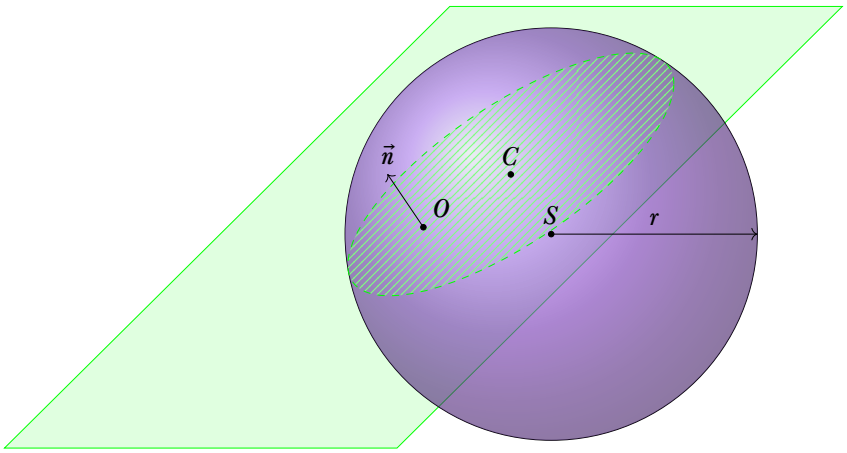



**Figure 4.5**: Example sphere and plane and the resulting intersection circle.

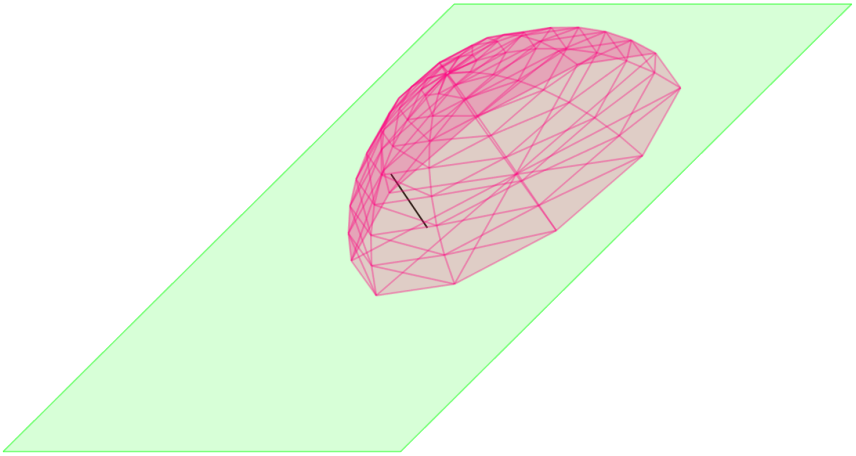


**Figure 4.6**: The resulting spherical cap triangle mesh in front of the plane rendered in wireframes.

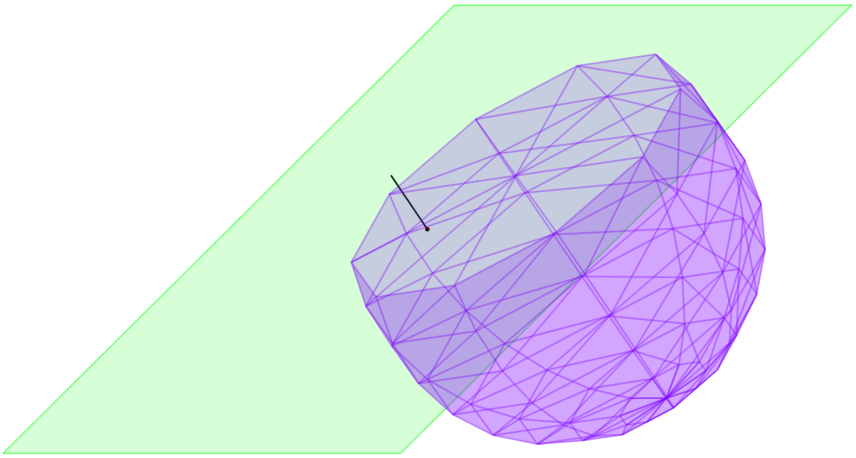


**Figure 4.7**: The resulting spherical cap triangle mesh behind the plane rendered in wireframes.

Normal calculation is not included here, since it is simply the normalized difference of the point itself and the sphere center. Except the vertices of the cut surface, which is flat with all normals being the inverse of the intersection circle's normal.

**Algorithm 4.1**: intersectionMesh (sphere $s$, point $O$, normal $\vec{n}$)

Triangle list $T$
**for** $\phi \in \{0, \Delta_\phi, \ldots, 2\pi - \Delta_\phi\}$ **do**
  $P \leftarrow C + \cos(\phi) r\vec{e}_2 + \sin(\phi) r\vec{e}_3$
  $P' \leftarrow C + \cos(\phi + \Delta_\phi) r\vec{e}_2 + \sin(\phi + \Delta_\phi) r\vec{e}_3$
  $T \xleftarrow{append} (C, P, P')$
  $\theta_P \leftarrow \arccos \frac{(P-S)\cdot\vec{e}_1}{r}$
  $\phi_P \leftarrow \text{arctan2}((P - S) \cdot \vec{e}_3, (P - S) \cdot \vec{e}_2)$
  $\phi_{P'} \leftarrow \text{arctan2}((P' - S) \cdot \vec{e}_3, (P' - S) \cdot \vec{e}_2)$
  $Q \leftarrow P$
  $Q' \leftarrow P'$
  **for** $t \in \{\Delta_\theta, 2\Delta_\theta, \ldots, 1\}^*$ **do**
    $\theta \leftarrow (1 - t)\theta_P + t\pi$
    $v \leftarrow S + \vec{e}_2 r \sin(\theta) \cos(\phi_P) + \vec{e}_3 r \sin(\theta) \sin(\phi_P) + \vec{e}_1 r \cos(\theta)$
    $v' \leftarrow$
     $S + \vec{e}_2 r \sin(\theta) \cos(\phi_{P'}) + \vec{e}_3 r \sin(\theta) \sin(\phi_{P'}) + \vec{e}_1 r \cos(\theta)$
    $T \xleftarrow{append} (v, v', Q)$
    $T \xleftarrow{append} (v', Q', Q)$
    $Q \leftarrow v$
    $Q' \leftarrow v'$
**return** $T$

*In case 1 is not a multiple of $\Delta_\theta$, simply do as many iterations as fit with $t < 1$, then do a final iteration with $t = 1$. This will however make the last triangle ring smaller than the rest.

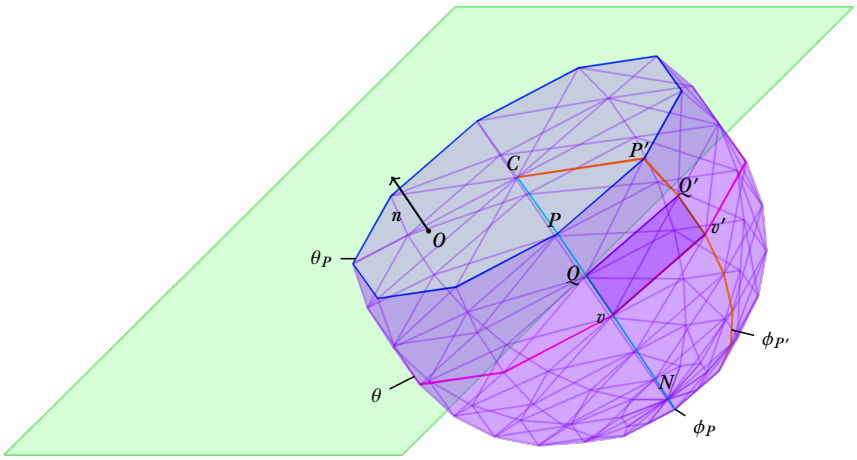



**Figure 4.8**: Second iteration of the inner loop described in Algorithm 4.1.

# 5 RESULTS

I tested my implementation on a computer that is running 64bit Windows 7, has an Intel Core i7–4770K CPU clocked at 3.5 GHz, 16 GB of DDR3 memory and an NVIDIA GeForce GTX 780 graphics card with 4 GB of GDDR5 memory. To evaluate my implementation and in turn my approach, I implemented and ran several experiments. I used synthetic and recorded sensor data in order to have reproducible experiments which I also used to test my implementation while developing it. The recorded sensor data for the haptic are generated while I performed real haptic interaction. For the depth sensor data I recorded depth frames that were generated from a Microsoft Kinect.

The computation times in general will be recorded in single-threaded operation, if not stated otherwise. This is done to eliminate the computational concurrency on the GPU that I described in Section 4.2.

### 5.0.1 *Realistic Conditions*

Firstly, I prepared an experiment with a realistic setup, i.e. all parameters are chosen close to what would happen in a real use-case. In Figure 5.1 I included an overview of the performance of all my presented approaches in this realistic setup. The setup is a pre-recorded haptic interaction with a pre-recorded Kinect frame. I ran the experiment with different IST resolution that represented the virtual tool, ranging from 500 to nearly 17 k.

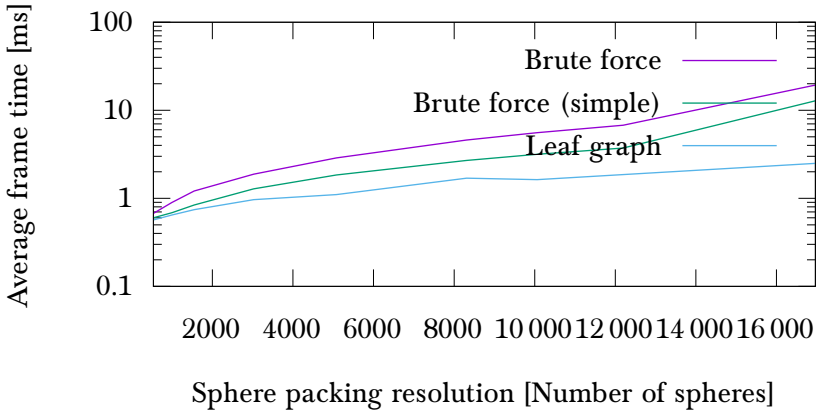


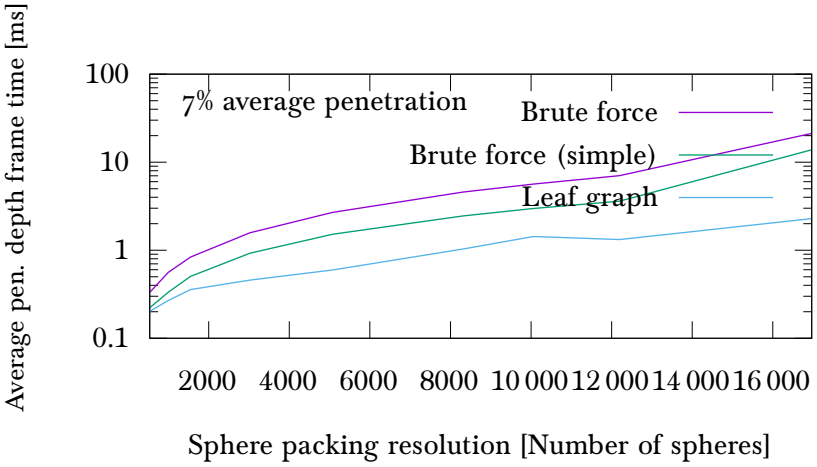

**Figure 5.1**: Realistic performance comparison. I compare my three approaches in a setup with data recorded from real environments and interaction with on average 10 % interpenetration. In all cases, the leaf-graph performs better than both brute force approaches.

I found that in this realistic setup, the graph approach shows the best performance. Moreover, the leaf-graph algorithm manages to stay fast enough for real-time haptic interactions (1 kHz) until a sphere packing size of around 5 k spheres. The brute force approaches are slower for all tested IST resolutions. However, they still manage to process an IST of smaller resolutions in the target 1 ms. The weighted average brute force approach can calculate penetration depth of an IST with up to 1 k spheres in under 1 ms. The simplified brute force even manages to process an IST with more than 2 k in under 1 ms.

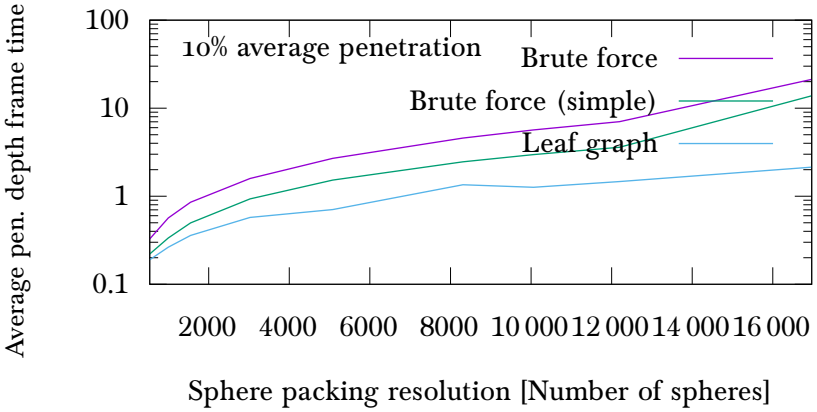
In Section 3.3.2 I already showed that the graph approach might not always be the fastest option and that its performance is directly dependent on the penetration depth of the IST. This stems from the fact that I have to traverse more of the graph the deeper the boundary spheres are located inside the object. In Figure 5.2 and Figure 5.3 I compare the time the volume computation takes for different haptic interactions that have significantly different average penetration depths.

As expected, for haptic interactions with an average penetration depth of 15 % and lower, the leaf-graph performs the best. If we look at the performance for haptic interaction with 16 % interpenetration (see Figure 5.2c), the simplified brute force approach actually calculates the penetration volume of an IST with 700 spheres or less faster. This behaviour gets more emphasized the larger the average penetration depth of the haptic interaction is. In those cases the performance difference is directly linked to the resolution of the IST. From these experiments, we can conclude that in general, the leaf-graph approach performs better if the resolution of the inner sphere tree is large. Additionally, as I already suggested, the performance of the leaf-graph is better for shallow penetrating interactions. To make this clearer, I included a plot in Figure 5.4. These are the data of two individual interaction benchmarks with the penetration depths and computation time of the graph traversal plotted over the simulation time. The direct correlation between the penetration depth and the volume computation times is clearly shown here.

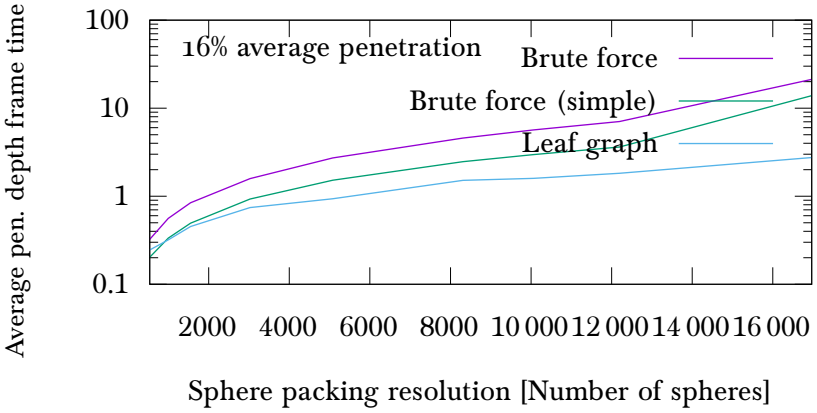
To further validate my observations I broke-down some of the previously mentioned experiments and recorded the time of each individual sub-task. The results are shown in Figure 5.5. The most time intensive sub-tasks are the collision detection and the volume computation via graph traversal. The computation effort of the collision detection shows very little dependence on the actual use-case, neither penetration depth nor IST resolution have a great impact. On the contrary, the graph traversal takes increasingly longer to come up with a result when the IST resolution is increased. Subsequently, the leaf-graph traversal shows the most potential to further optimize the overall implementation by. The penetration depth, as previously mentioned, is another parameter that makes the traversal perform worse. This might not be easy to overcome, since by design there is less work to be done when less of the graph needs to be traversed.




**(a)** The graph approach computes its results faster in shallow penetrations.

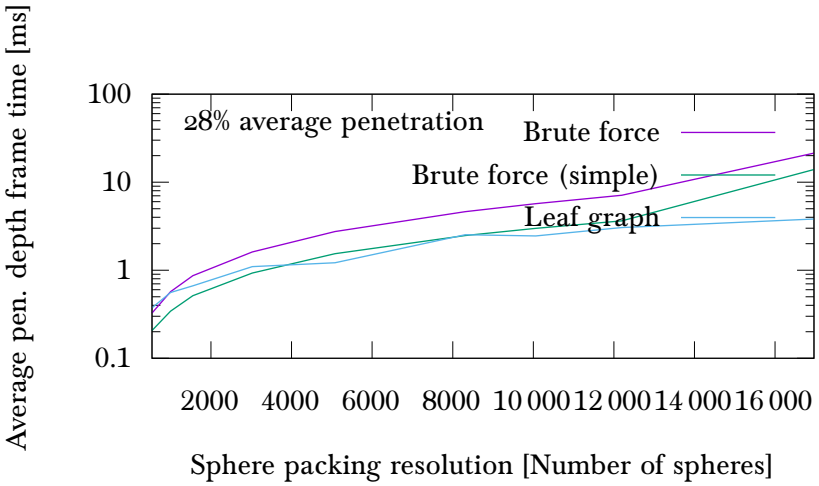



**(b)** The graph approach computes its results faster in inter-penetrations of 10%.

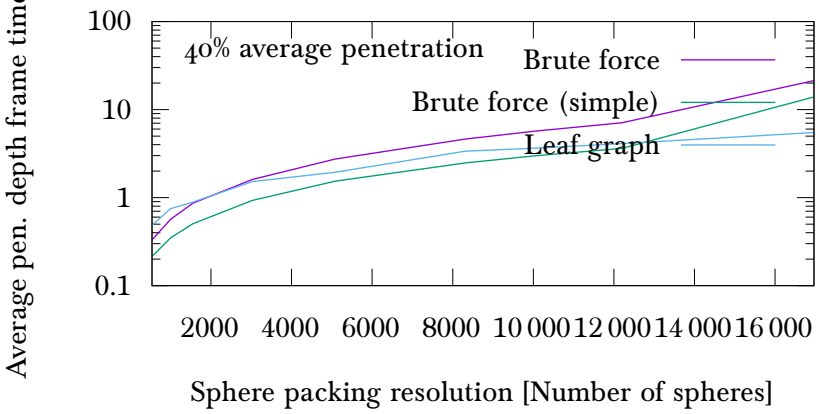



**(c)** The graph approach computes its results faster in medium penetrations of 10–16%. This is true in almost all IST resolutions, only if they are less than 1 k is the simplified brute force approach.

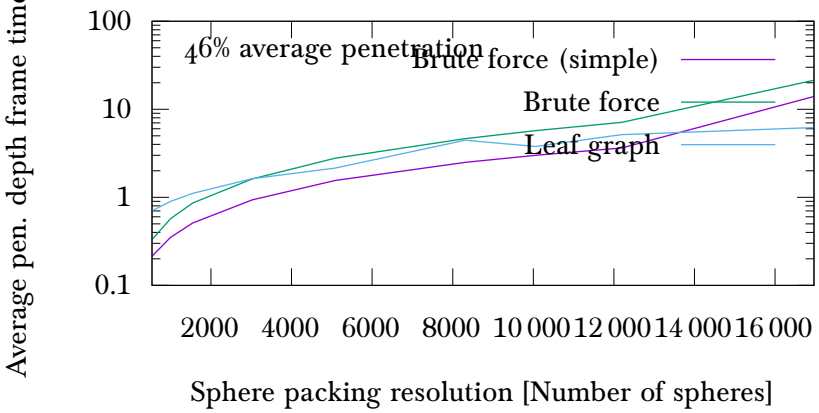
**Figure 5.2**: Realistic computation time comparison in shallow-medium penetrations. I compare the three algorithms in a realistic setup with a recorded PCD with recorded haptic interactions The haptic interactions have the respectively noted averaged penetration depths in the range of 7–16% of the virtual tool's volume. Brute force approaches show virtually no different computation times, no matter the penetration depth. Leaf-graph traversal shows heavily differing computation effort depending on the penetration depth.




**(a)** For IST resolutions of 4 k to 12 k, the graph traversal is as fast as the simplified brute force approach or even faster. For sphere packings upwards of 12 k, I again get significantly faster results.

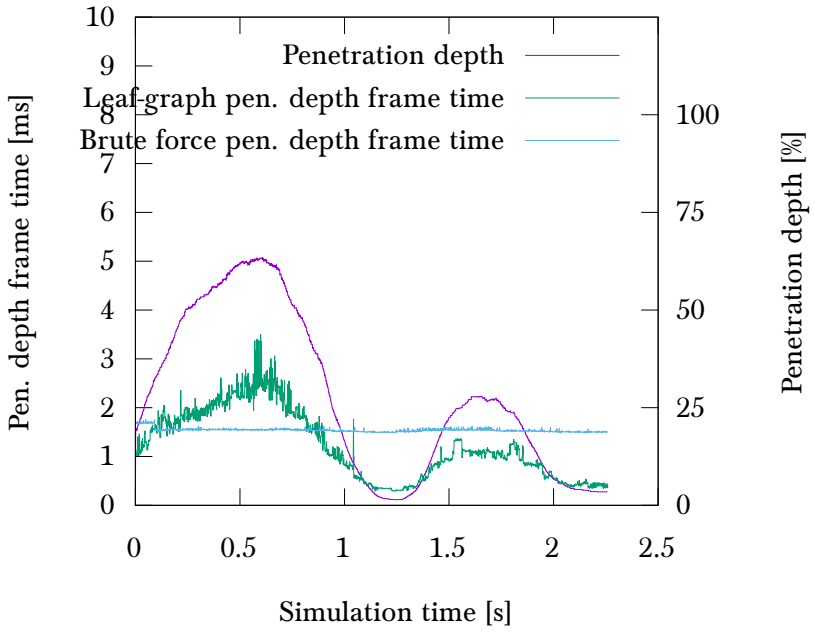



**(b)** For extreme penetration depth of 40%, the graph approach more quickly calculates the penetration depth if the IST has a resolution of about 13 k or more.

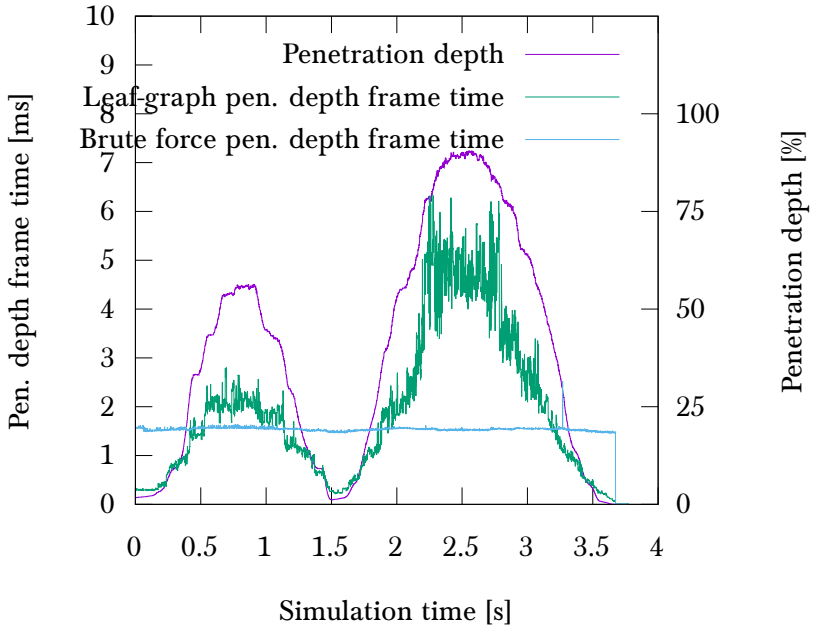



**(c)** Comparable behaviour as for 40%, explained in Figure 5.3b

**Figure 5.3:** Computation time comparison in deep penetrations. I compare the three algorithms in a setup with a recorded PCD with recorded haptic interactions. The interactions have the respectively noted averaged penetration depths in the range of 28–46% of the virtual tool's volume. Brute force behaves equally to all cases showed in Figure 5.2. The graph traversal takes longer than the brute force approaches in case of low IST resolutions and deep penetrations.

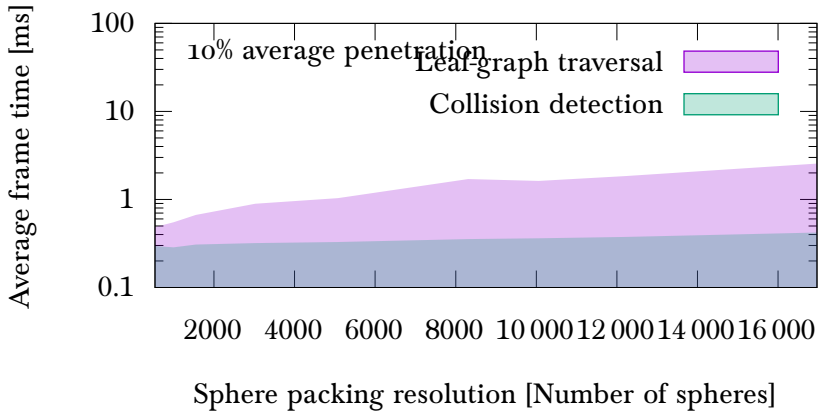



**(a)** This is one of the runs presented in Figure 5.3a that resulted in an average penetration depth of 28 %. The inner sphere tree was generated from a sphere packing of about 5 k spheres.

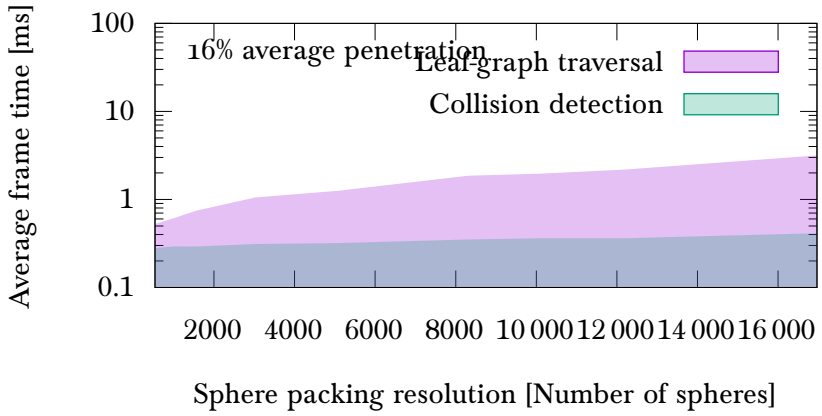



**(b)** This interaction was used to generate data for Figure 5.3b. It has very deep penetrations. Overall, this resulted in 40 % average inter-penetration. The inner sphere tree is made up of about 5 k spheres.

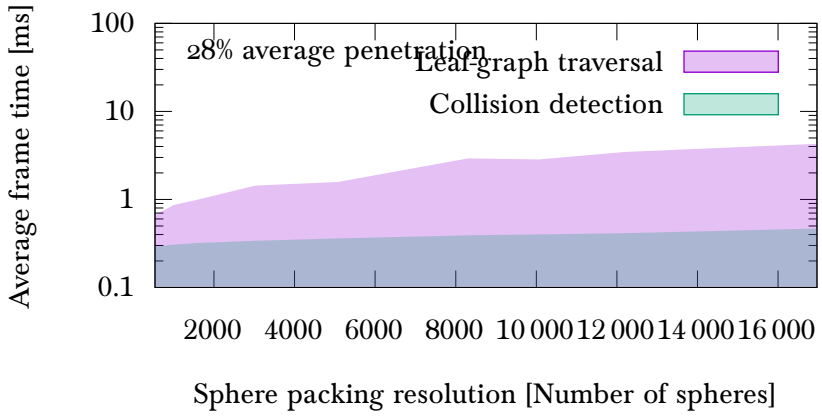
**Figure 5.4**: Computational times for specific interactions over their simulation time. These are individual interactions that show their penetration depth and the corresponding computation time the leaf-graph took to calculate the penetration volume. The correlation between penetration depth and computation effort is very obvious.




**(a)** Test scenario with average penetration depth of 10 %.




**(b)** Test scenario with average penetration depth of 16 %.




**(c)** Test scenario with average penetration depth of 28 %.

**Figure 5.5:** Performance break-down per subtask. Break-down of the performance trends of the two main subtasks of the leaf-graph approach based haptic rendering. The collision detection slope is very flat with increasing IST sizes and penetration depths compared to the graph traversal.

Overall, the best choice of my presented approaches based on performance is largely decided by the average penetration depth. The anticipated depth of the inter-penetrations largely depends on the stiffness of the haptic simulation that you choose for your use-case. If you expect an average penetration depth of more than 30%, the simplified brute force will likely perform better on average. I found that a stiffness that reaches the maximum feedback at 1/8th of the total volume to be reasonable for my use-case. This means I will have a maximum penetration depth of 12.5%, so the average will stay far below that, making the graph approach the best option by far.

I find it noteworthy that the performance curve of the graph algorithm shows more fluctuation than both other approaches. For example, Figure 5.3a and Figure 5.3c show the algorithm taking longer for the 8 k IST than for the 10 k. This suggests that some sphere packings are more suited to be used for the graph-based approach than others. However, I did not look further into this matter, since it would go beyond the scope of this work.

Another characteristic that I noticed in the shown plots is the increase in computational time after 12 k for both brute force approaches. For sphere packings of sizes up to 12 k, the trend of the plots is always very identical, only after 12 k comes a noticeable increase in the plot's slope. I suspect that this is caused by the increasing demand for parallel work reaching the limits of the hardware. It could be interesting to run the same benchmarks on better hardware to see if the plot's slope will rise after a bigger IST resolution.

### 5.0.2 *Unknown Conditions*

The benchmarks in Section 5.0.1 are only a useful resource if you can estimate the average penetration depth of your individual application. If you can not make any guess on what kind of haptic interaction you are anticipating, I included a synthetic benchmark. In Figure 5.6, you can see the results of a synthetic benchmark.

I set it up so that there is an even distribution of possible penetration depth from 0–100 %, so the average is 50 %. The haptic interactions was a synthetically generated perfect linear movement towards a synthetically generated perfect wall of points. I categorize this interaction as highly unrealistic, since usually objects do not pass through each other in rigid physics. And in general, haptic rendering is a physical simulation of the real world. This is basically the worst case scenario for the leaf-graph, because of the deep penetrations.

As such, it is not a surprise to see, that it only performs better with very high resolution inner sphere trees. In any other case, the simplified brute force is the best option in terms of performance, maintaining an average of around 1 ms up to a sphere packing size of around 2 k spheres. This also means its performance should be suitable for real-time hap-

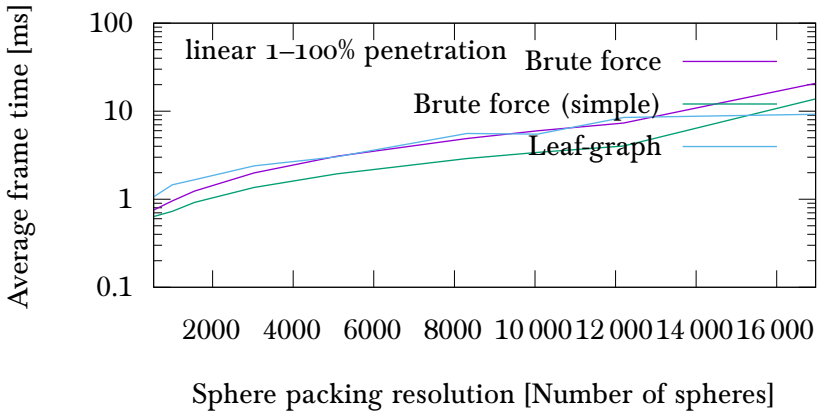



**Figure 5.6**: Full penetration in a synthetic setup. I show the computation behaviour averaged over all possible penetration depths. If the range of penetration depth can not be estimated beforehand, this data can be used evaluate the on average best performing choice based on the resolution of the virtual tool's IST.

tic interaction, regardless of the exact use-case and its specific average penetration depths.

## 5.1 QUALITY

### 5.1.1 *Realistic*

I evaluate the quality of my volume measure in a realistic setting by taking the logged data of the benchmarks presented in Section 5.0.1. Each benchmark produces a penetration volume with a direction, which can then be scaled up with the desired stiffness to cover the whole range of the used haptic device. Subsequently, when I inspect the generated haptic forces, I am also checking the penetration measure's quality. In the following experiments, I chose a haptic device that has a maximum linear force feedback of 10 N, so a haptic feedback of 5 N would correspond to a 50 % penetration volume.

I will inspect the generated penetration volume's progress over the simulation time looking for as little as possible undesirable discontinuities in the measured volume.

In Figure 5.7, I show the results of the low penetrating benchmark. Overall, the forces show a considerable amount of continuity. There is however a small amount of discontinuity only in the direction of the $x$-axis. All of my possible algorithm setups handle this situation well, but in the weighted average brute force approach produces the smoothest result by a small margin. For comparison, see the graph traversal forces in Figure 5.8.

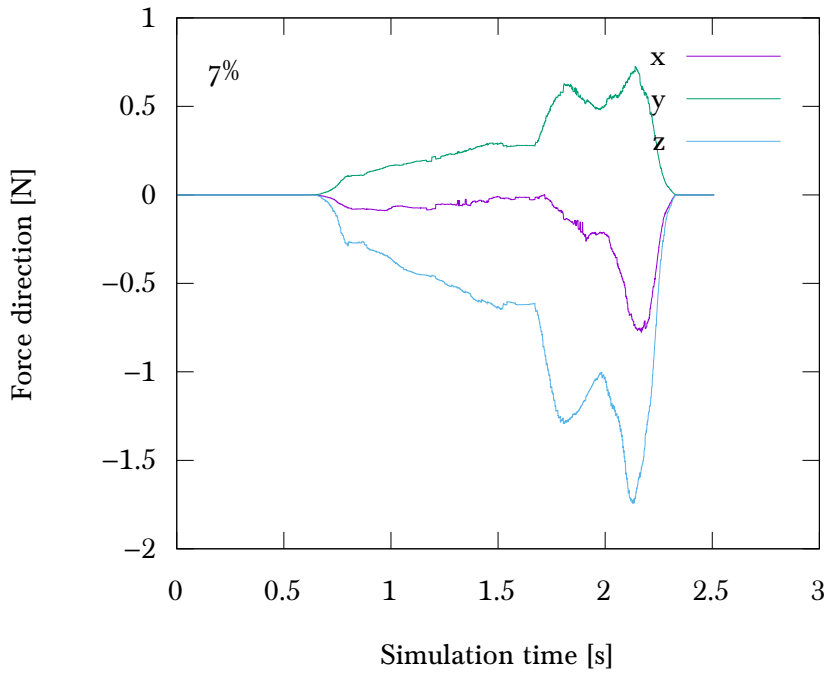



**Figure 5.7**: Force plot for interaction with average penetration depth of 7 %. Sphere packing has a size of 5 k spheres. Brute force with proximity-based priority used for volume computation.

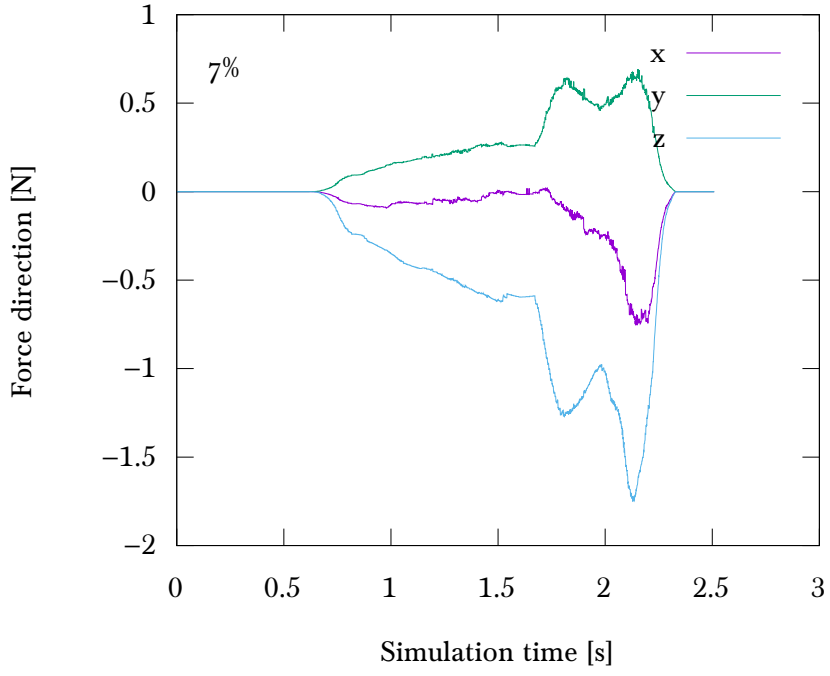



**Figure 5.8**: Force plot for interaction with average penetration depth of 7 %. Sphere packing has size of 5 k spheres. Leaf-graph used for volume computation.

In Figure 5.9, I ran a medium deep penetration benchmark. This time I show the results that the faster leaf-graph produces, there is again a noticeable increase in discontinuity in the produced forces.

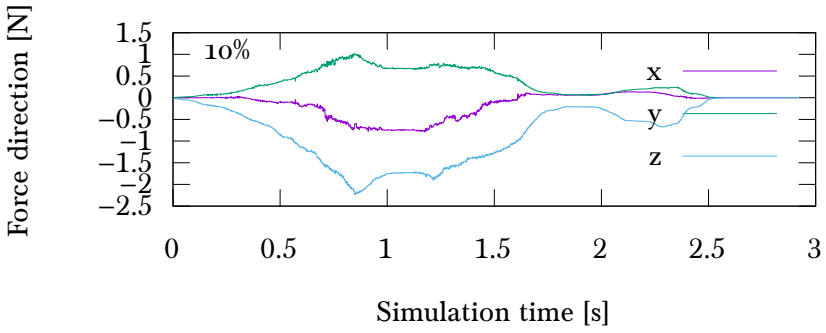



**Figure 5.9**: Force plot for interaction with average penetration depth of 10 %. Sphere packing has size of 3 k spheres. Leaf-graph used for volume computation.

In general, I notice this discrepancy in most of my experiments, it is however more pronounced the deeper the penetration is. Even the simplified brute force will produce forces with less discontinuities in some of those cases. If you for example consider one of my deep penetration experiments of 40 % average inter-penetration. The forces produced by the graph algorithm are shown in Figure 5.10.

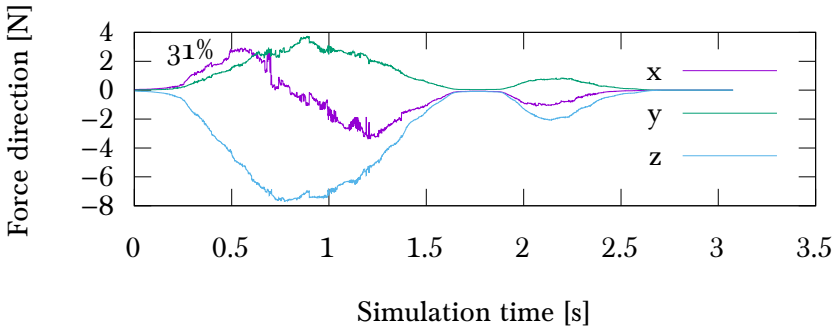



**Figure 5.10**: Force plot for interaction with deep penetration depth of 31 % on average. Sphere packing has size of 1 k spheres. Leaf-graph used for volume computation.

The have a great amount of discontinuity, especially compared to the clear forces produced by the simplified brute force algorithm (see Figure 5.11).

Of course, the weighted average brute force produces an even more desired result (see Figure 5.12). Considering the great speed-up of the simplification of the brute force, this is however not a big difference.

In Figures 5.13, 5.14 and 5.15, I show various other benchmarks of experiments I conducted. I used various inner sphere tree resolutions to show that there is no major difference between low and high resolution ISTs.

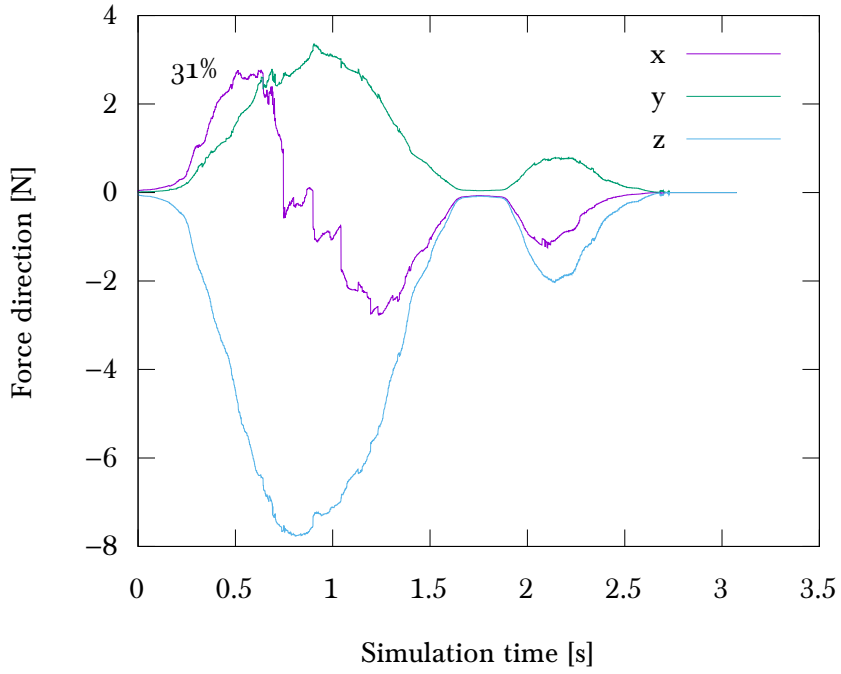



**Figure 5.11**: Force plot for interaction with deep penetration depth of 31 % on average. Sphere packing has size of 1 k spheres. Simplified brute force with proximity-based priority used for volume computation.

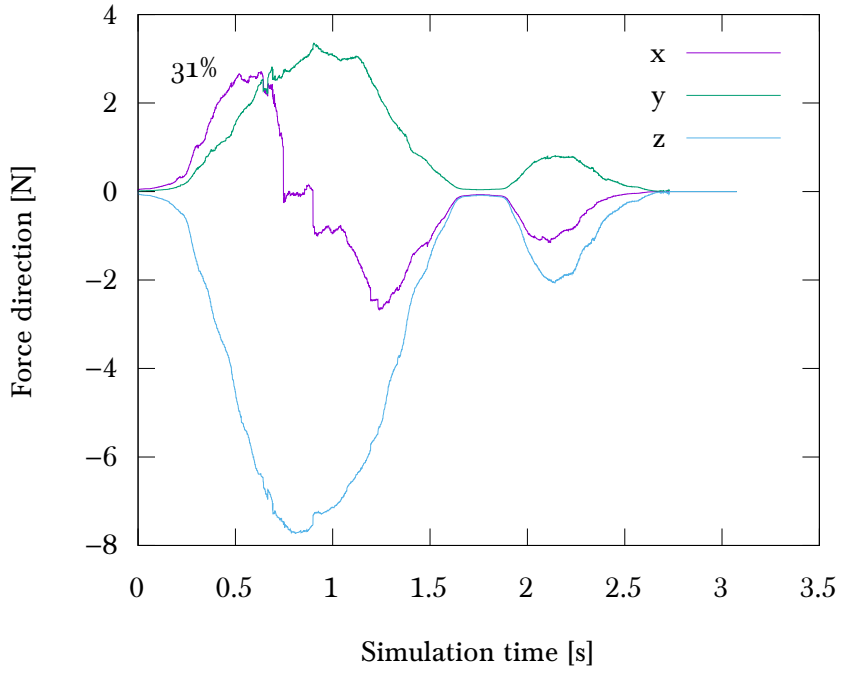



**Figure 5.12**: Force plot for interaction with deep penetration depth of 31 % on average. Sphere packing has size of 1 k spheres. Weighted average brute force with proximity-based priority used for volume computation.

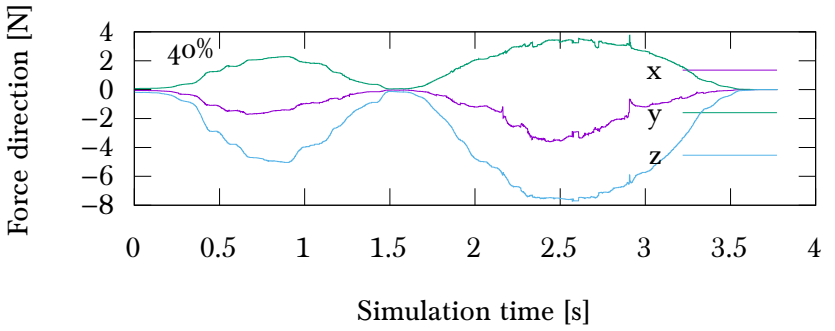



**Figure 5.13**: Force plot for interaction with very deep penetration depth of 40 % on average. Sphere packing has size of 8 k spheres. Weighted average brute force with proximity-based priority used for volume computation.

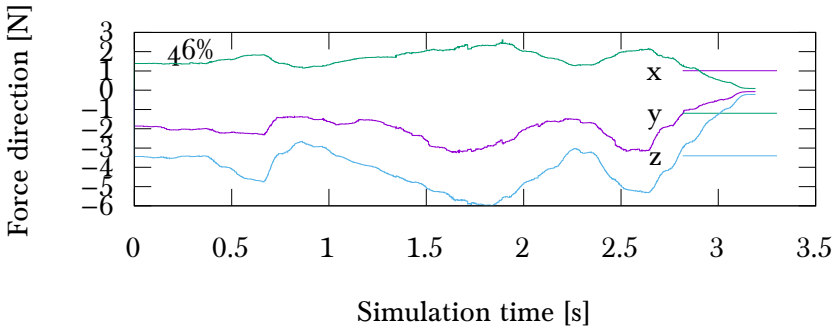



**Figure 5.14**: Force plot for interaction with very deep penetration depth of 46 % on average. Sphere packing has size of 12 k spheres. Weighted average brute force with proximity-based priority used for volume computation.

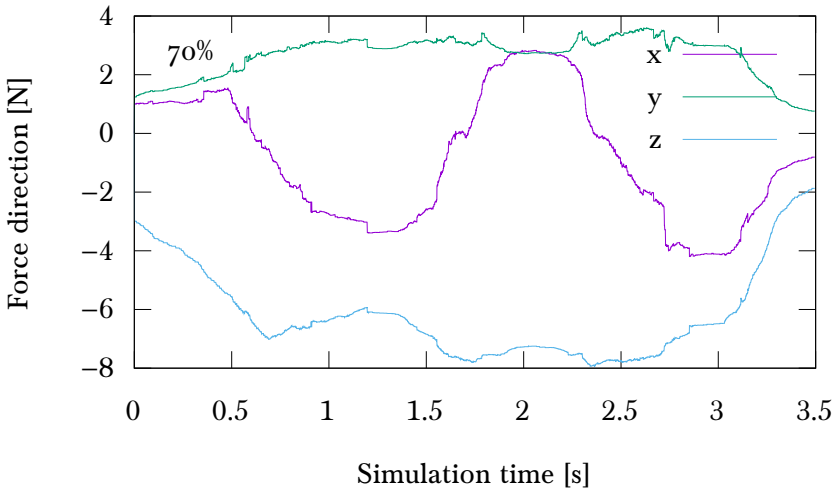



**Figure 5.15**: Force plot for interaction with extremely deep penetration depth of 70 % on average. Sphere packing has size of 1.5 k spheres. Weighted average brute force with proximity-based priority used for volume computation.

In fact, I want to further show how small the difference between the produced forces of very differently sized sphere packings is. In Figure 5.16, I plotted the calculated penetration depth with an sphere packing of 1 k spheres and the other one with 17 k spheres. All other variables for the benchmarks are exactly the same. The plot shows that the volume that is calculated from the low-resolution IST is slightly different the volume of the high-resolution IST. It is not always a subset of the higher resolution result, which is what I suspected. This likely stems from the fact that the distribution of the spheres is different and as a consequence the volume is differently distributed. However, the difference is so minuscule that it is very questionable why one should consider the additional 16 k spheres. As shown in Figure 5.3a, this would increase the computational effort by nearly two orders of magnitude.

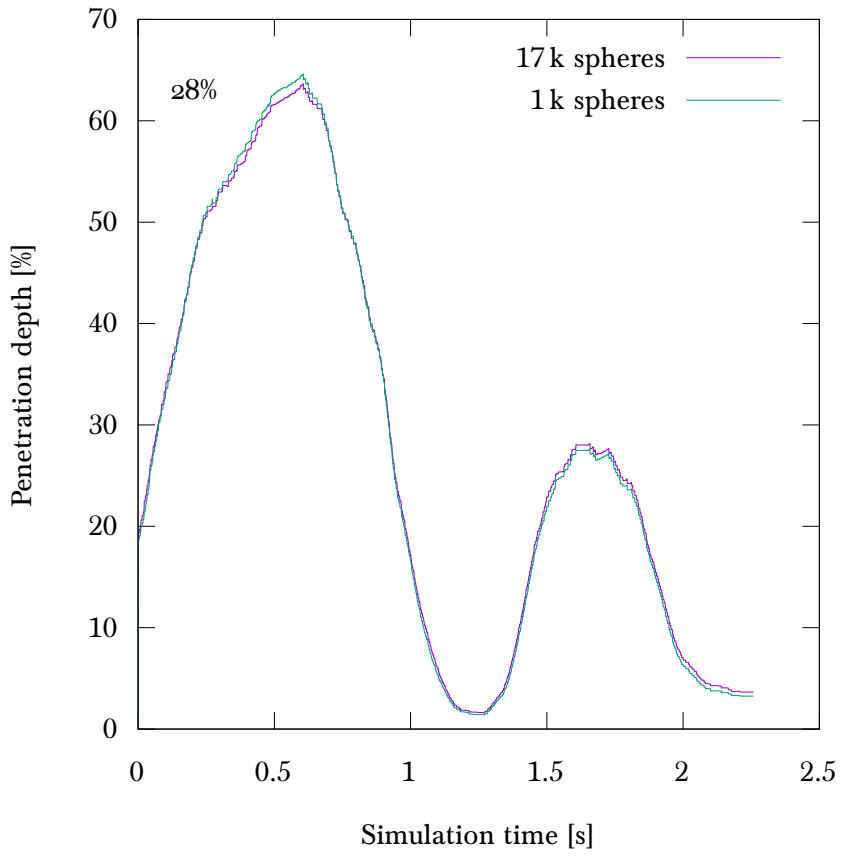



**Figure 5.16**: Force plot for interaction with deep penetration depth of 28 % on average. Weighted average brute force with proximity-based priority used for volume computation.

Overall, the quality of my presented approaches behaves in relation to their respective performance. The most expensive method, the weighted average brute force produces the smoothest forces. The simplification of the brute force offers a significant performance boost, but the forces have noticeably more steps, because collision data of just one boundary

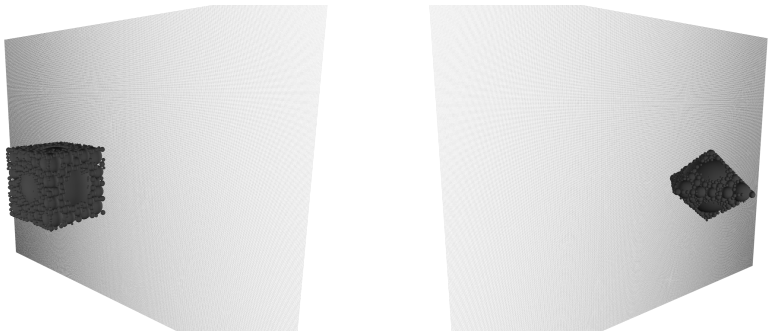
sphere is taking into consideration. The graph-based approach offers a great increase in performance under the right circumstances. Its quality is similar in some cases and worse in cases where the penetration depth is very high. In those cases, the penetration depth and the resulting forces show considerable amounts of discontinuity. This can be explained by the different approach to process the inside spheres. The brute force approaches will intersect the inside spheres with collision data of appropriate boundary spheres, where the graph algorithm simply counts them as fully inside.

### 5.1.2 *Synthetic*

It is difficult to know how the penetration depth is supposed to look when using real sensor data, so it is hard to judge how well the penetration measure is performing, besides tracking the amount of discontinuity. To have a more solid idea of how precise my penetration measure is, I conducted a series of experiments with synthetic sensor data, giving me quantifiable results of the quality. In these scenarios I simulated sensor data of familiar and simple environments and interactions. The sphere packing that represents the virtual tool is chosen as one of three simple objects here (a cube, a tetrahedron and an octahedron), so that I am dealing with familiar distributions of volume. The haptic interaction here is usually just a linear translation along a straight path in order see the progress of the recognized penetration volume against various environments that are approximated by an artificial point cloud. I constructed the point cloud environment from the same resolution of Microsoft's Kinect (about 307 k points) in order to have a realistic density of points.

#### 5.1.2.1 *Wall*

The first synthetic environment is a simple wall that is perpendicular to the $z$-axis (see Figure 5.17). The virtual tool is a cube that is approx-



**(a)** The virtual tool is a cube filled with about 1 k spheres.

**(b)** The virtual tool is a octahedron filled with about 1 k spheres.

**Figure 5.17**: Synthetic wall setup. The point cloud is a perfect wall.

imated by about 1 k spheres (see Figure 5.17a). It is traveling along a

linear path along the $z$-axis, perpendicular to the wall. The resulting penetration depth over the progress of the simulation can be seen in Figure 5.18. The ideal line would be a straight line from $(0, 0)$ to $(100, 100)$.

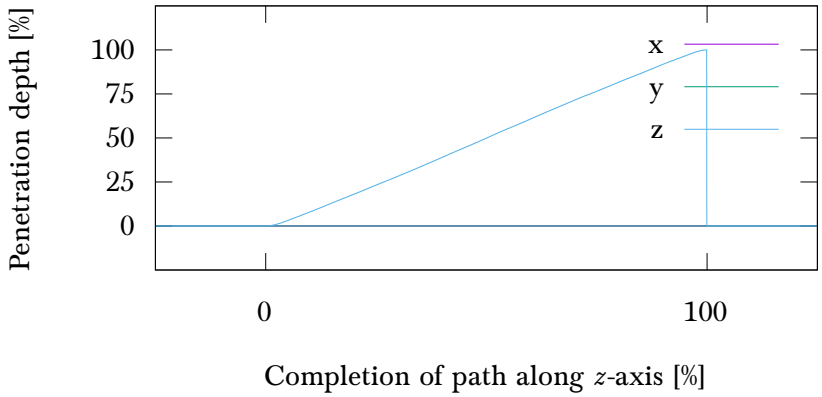



**Figure 5.18**: Penetration depth for a cube entering a wall.

The result is nearly that, the slight rounding off of the edges comes from the fact that the spheres can't represent a perfectly sharp edge of the volume distribution, which a sharp cube would have.

If I conduct a similar experiment, however I let an octahedron pass through the wall instead of a cube (see Figure 5.17b), I get the result shown in Figure 5.19. The curve looks very smooth going from 0% to

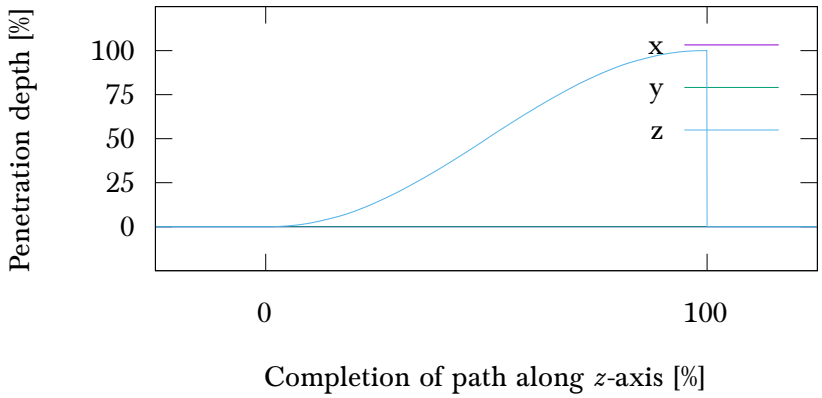
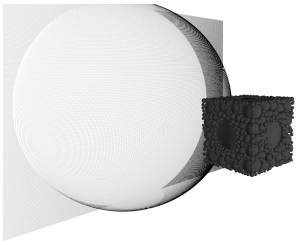



**Figure 5.19**: Penetration depth for an octahedron entering a wall.

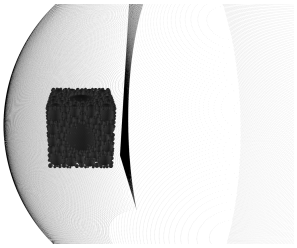
100% as well. The steeper incline in the middle is the result of the volume distribution of the octahedron. It has a crosscut that is largest in the middle and tapers off towards the ends. This is perfectly represented in the curve. My algorithm generates nearly perfect results for a simple wall. A wall is a perfectly uniform object, which is why these tests are easy to handle for my algorithm.

#### 5.1.2.2 *Spheres*

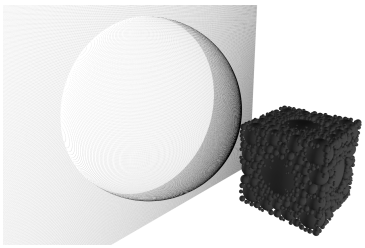
I want to test my penetration measure in more irregular environments. For that I generated more synthetic point cloud environments. In Figure 5.20, you can see the various synthetic setups that I used in the following. I generated point clouds in the form of spheres' outsides and



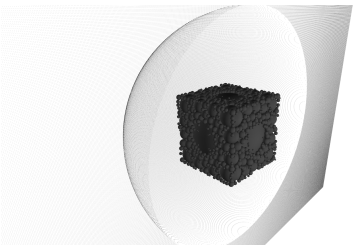
**(a)** Outside of a large sphere with a radius of 480 pixel.



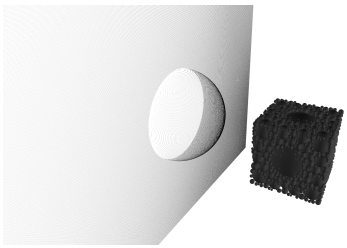
**(b)** Inside of a large sphere with a radius a 480 pixel.



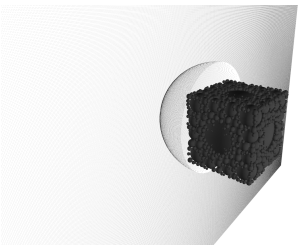
**(c)** Outside of a medium-sized sphere with a radius of 320 pixel.



**(d)** Inside of a medium-sized sphere with a radius of 320 pixel.



**(e)** Outside of a small sphere with a radius of 160 pixel.



**(f)** Inside of a small sphere with a radius of 160 pixel.

**Figure 5.20**: Synthetic point cloud spheres at various radii (given in Kinect image space pixel units).

insides at various radii.

As one would suspect, we find out that more irregular point clouds are a more difficult circumstance to compute the penetration depth for my algorithm. Firstly, we look at the case of a cube entering a large sphere. As a sphere gets larger the surface that contacts the virtual tool is more resembled of a wall. Thus, it is not surprising that my algorithm handles a large sphere (as seen in Figure 5.20b) quite well. In Figure 5.21 the resulting penetration depth for a cube with about 1 k spheres can be seen leaving a large sphere from the inside. The result looks very similar to the perfect wall example, except for a very small amount of noise on

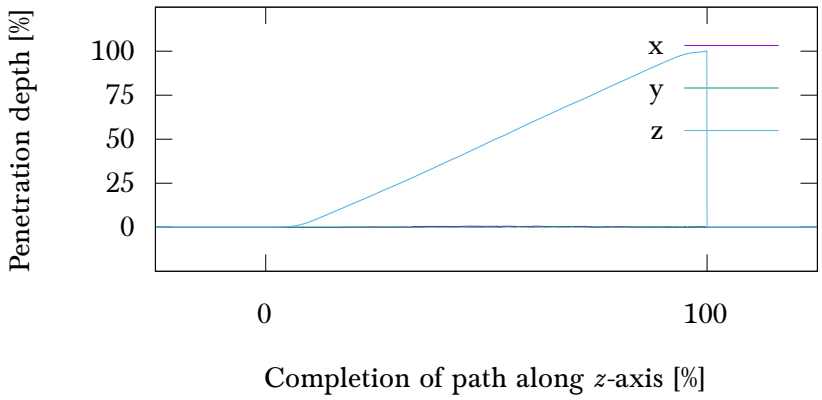



**Figure 5.21**: Penetration depth of a cube leaving a large sphere.

the $x$- and $y$-axes. Also, the penetration depth shows some irregularities towards the end of the simulation, when the cube is leaving the point cloud contact. Overall, this is still a very good result, the penetration depth is correctly measured at almost all ranges of the simulation.

Now, if we reduce the size of the sphere further we create an even more irregular surface. I conducted the same experiment for a medium-sized sphere (as seen in Figure 5.20d). Figure 5.22 shows the results. However,

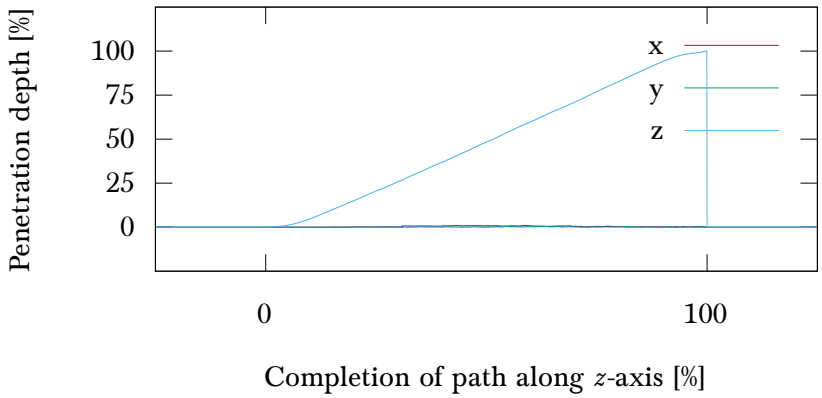



**Figure 5.22**: Penetration depth of a cube leaving a medium-sized sphere.

we find that again, the result is very close to that of the perfect wall with some added noise on the other axes and an irregular progression towards the end.

The above experiments were against the inside of a sphere. When take take the outside of a sphere as the point cloud environment, the results look very different. For example, an equally sized sphere as previously, approached from the outside can be seen in Figure 5.20c. The resulting penetration depth over the course of the simulation are shown in Figure 5.23. The beginning of the plot is very similar to the one where the sphere was approached from the inside, however the later parts have more

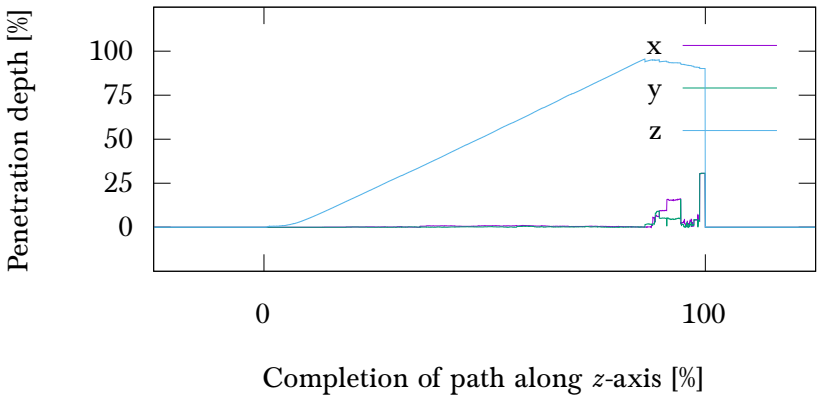



**Figure 5.23**: Penetration depth of a cube entering a medium-size sphere.

exaggerated irregularities. There are significant amounts of penetration depths that are attributed to be in the direction of the $x$- and $y$-axis, while it should just be in the direction of the $z$-axis. In Figure 5.24, you can see the result of smaller sphere (see Figure 5.20e) as the point cloud being penetrated. The plot shows large amount of penetration volume being

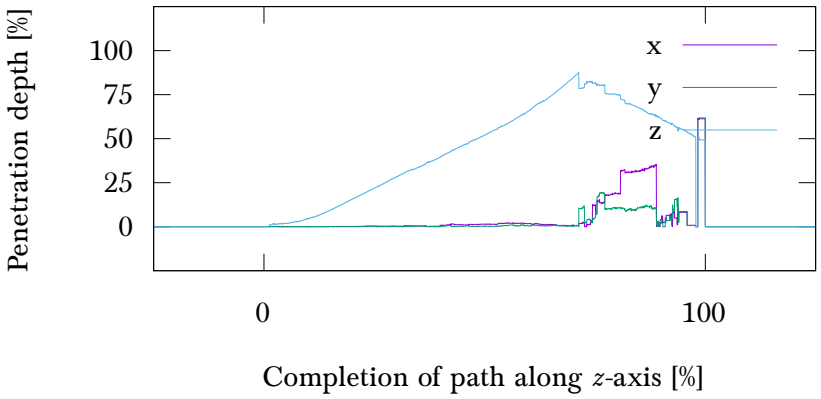



**Figure 5.24**: Penetration depth of a cube entering a medium-size sphere.

falsely ascribed to the $x$- and $y$-axis. In fact, all of the errors look exactly like in the plot to the medium-sized sphere, just scaled up. Besides these errors towards the end, the penetration depth is close to the ideal until about 75 %, which is still a large range. To show that these errors are not caused by the size of the sphere, just enhanced, I also conducted the experiment again where I approached the sphere from the inside (see Figure **??**). There are no big errors visible, the results look very similar to those of the large sphere (as seen in Figure 5.21).

#### 5.1.2.3 *Ridges & Corners*

My previous experiments indicate that my algorithm handles concave environments better than convex ones. In the following part I will look

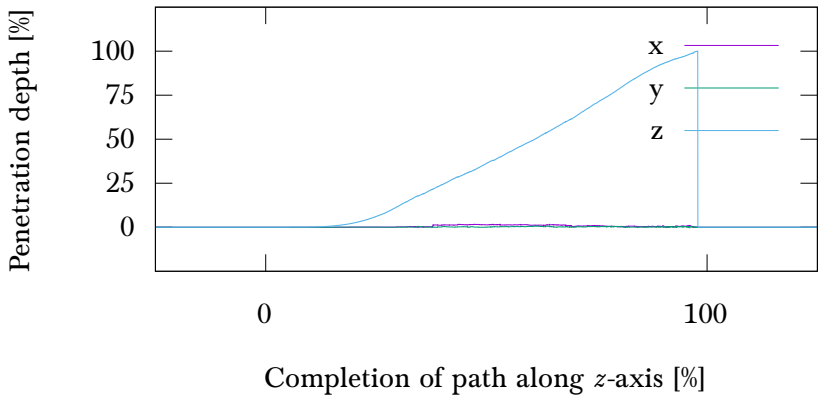
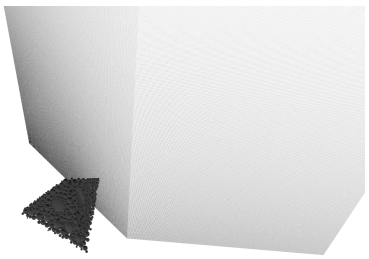
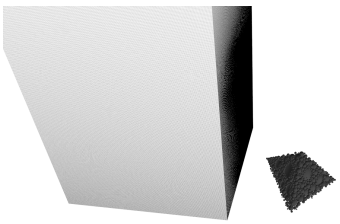



**Figure 5.25**: Penetration depth of a cube entering a medium-size sphere.

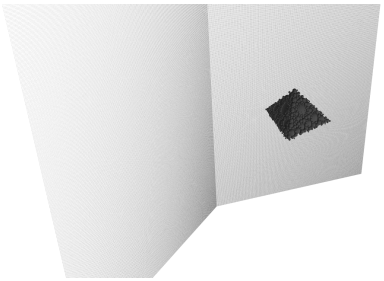
at the behaviour that the penetration measure shows when we penetrate a ridge as the point cloud environment (see Figure 5.26). For the virtual tool I used a tetrahedron. It has a similar volume distribution as the octahedron, so the penetration volume curve will look similarly steeper in the mid section and tapes off towards the ends.



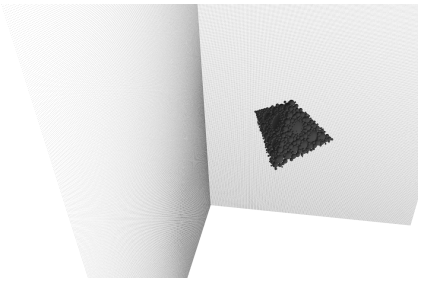
**(a)** Blunt ridge with an opening angle of 135°.



**(b)** Sharp ridge with an opening angle of 90°.



**(c)** Blunt corner with an opening angle of 225°.



**(d)** Sharp corner with an opening angle of 270°.

**Figure 5.26**: Different corners and ridges with various angles as point cloud environments.

The first experiment is against a blunt corner (see Figure 5.26d). The resulting penetration depth progression over the course of the simulation are shown in Figure 5.27. Overall, the curve shows the anticipated penetration volume along the $z$-axis, with a steeper incline in the mid-

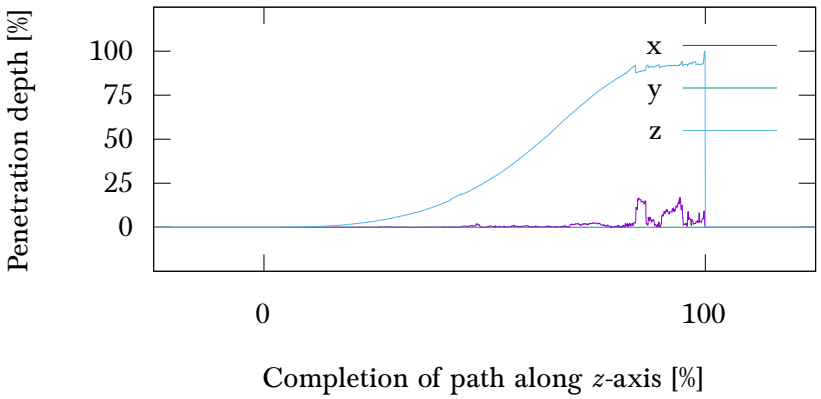



**Figure 5.27**: Penetration depth of tetrahedron penetrating a blunt corner with an opening angle of 225°.

dle. Above 80 % penetration depth there starts to be a significant amount of volume being ascribed to the $x$-axis. The penetration measurements would in application result in a force towards either of the $x$ directions when penetration the corner. Ideally, the resulting feedback would point into just the $z$ direction, the direction of the biggest opening of the corner. However, that is the case for a large portion of the simulation path, in the more realistic range of penetration depth.

In the next benchmark, I used a blunt ridge instead of a corner, so I simply changed the same angled wall sections from concave to convex, expecting a worse result. The resulting penetration depth can be seen in

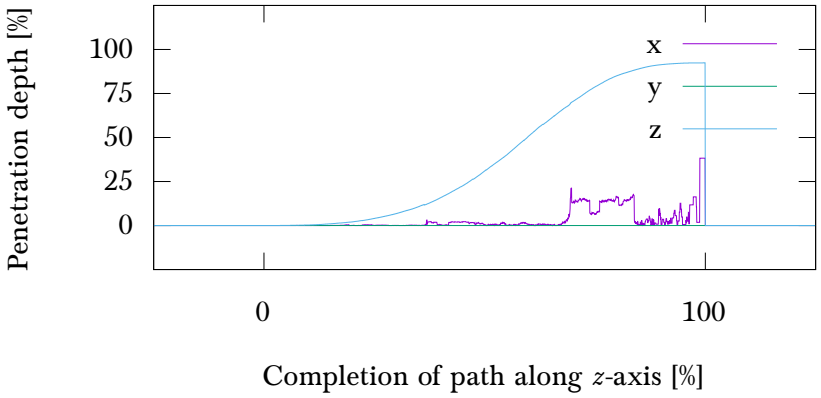



**Figure 5.28**: Penetration depth of tetrahedron penetrating a blunt ridge with an opening angle of 135°.

Figure 5.28. As expected, the penetration measurement degrades earlier than in the concave example. After about 60 % of penetration depth, the measured volume would result in a considerable force in the $x$ direction. This result in this case is not a big fault, as in reality when pushing against a ridge, the pushed object would be propelled towards either of the sides of the ridge.

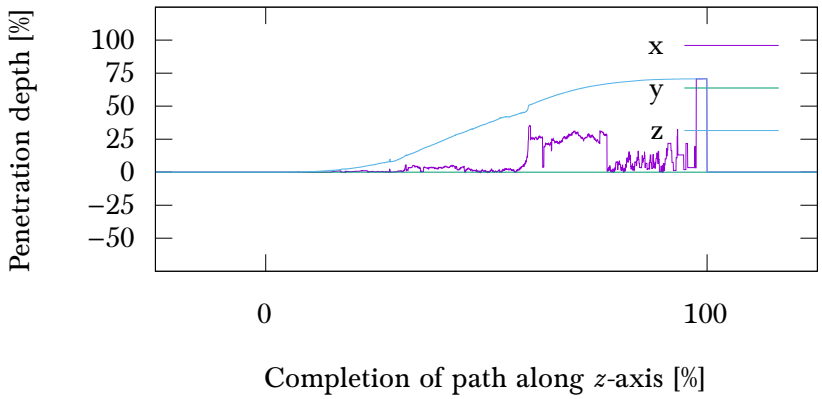



**Figure 5.29**: Penetration depth of tetrahedron penetrating a sharp ridge with an opening angle of 90°.

In Figure 5.29, I repeated the same experiment, just with a sharper ridge. As the virtual tool enters the ridge, the penetration measure again results in a large portion of the volume being ascribed to either of the $x$ sides. When generating forces from this penetration volume, the haptic feedback would propel the virtual tool sooner towards either of the sides of the ridges. Only until about 45 % of penetration depth would the generated force keep a relatively stable force in the $z$-direction alone.

# CONCLUSION & FUTURE WORK

# 6

I presented a novel penetration measure that works for general 3D objects in environments that are approximated by a range sensor. I conceived three different algorithms and describe how to implement them. All of them are suited for GPU parallelized implementations.

I have developed an efficient implementation of a working prototype, which I used to test and evaluate my concept. I compare the different approaches under various circumstances to illustrate their respective advantages and disadvantages. All of the presented approaches can manage to stay within haptic-ready frame times, however for differently accurate approximations of the 3D CAD object. The brute force approaches have the general draw back of not respecting the connectivity of the geometry of the virtual tool in order to propagate collisions through the object's volume. Instead, the brute force algorithms affect spheres globally, regardless of the underlying volume topology. In scenarios where the virtual tool is concave, this might lead to spheres being influenced by boundary spheres that have no valid connection to it. Both presented brute force approaches are however straight forward to implement and exhibit very stable behaviour both in terms of performance and feedback quality under various circumstances. There is an opportunity to further explore the priority of boundary spheres in regard to other spheres beyond the simple metrics of distance or collision count, which were introduced here. The performance of both brute force approaches is unaffected by penetration depth, however the collision detection will be slower because it has to traverse farther into the bounding volume hierarchy.

The performance of the graph-based approach is shown to be very fast and significantly faster for haptic interactions that exhibit a low average penetration depth. Especially for very large sphere packings, the graph-based traversal shows better performance, managing to process the same penetration situation an order of magnitude faster (for sufficiently shallow penetration and highly accurate sphere packings). The graph-based approach manages to avoid having to process a large portion of the inner spheres, since the algorithm uses preprocessed neighbourhood connectivity to search for affected spheres locally. The graph algorithm shows a noticeable amount of noise in the calculated force directions. This stems from the fact the algorithm has concurrent threads potentially traversing the same parts of the graph and marking spheres. This results in the sphere associations being non-deterministic, so the surface direction the inside volume will be contributed to is unstable. I demonstrated that this does not affect the overall volume that was calculated however. I suspect that there is an opportunity to explore how different parameters of the sphere packings affect the performance of the graph-based algorithm, as

I noticed a pattern in the performance benchmarks that favored specific sphere packings over others.

I successfully combined real-time surface estimation with my new penetration measure on a shared GPU to realize 6-DOF haptic rendering of streaming point clouds. The shared GPU caused the haptic rendering tasks to occasionally be blocked by the surface estimation. I solved this problem by partitioning a single Kinect frame in small chunks and processing them slowly over the total amount of frame time that is available to stay in real-time limits. This reduced the blockage of the haptic thread, thus decreasing the average frame time as well as decreasing its variance, leading to a more stable frame time.

My experiments show that my penetration measure can handle concave point cloud environments well. Convex point clouds can lead to irregular penetration volume on axis that should not be affected. This problem does however only arise when the penetration depth is extremely large or the virtual tool is nearly passing through the point cloud. As this is a very unrealistic circumstance I find that my penetration measure is well suited for many types of point cloud environments, as long as the interactions stay within a realistic limit.

I think it is worth to further develop this method. Some ideas that I have were already mentioned in this chapter or earlier, but I want to offer you more of my thoughts. Firstly, the implementation has still room to be optimized. For example, some memory accesses that are not random are not yet implemented to be coalesced accesses. The implementation could also be enhanced in order to support a multi GPU setup, where one GPU handles all the surface estimation and transfers the point cloud and its normals to the other GPU afterwards, additionally the 3D rendering would be done on that GPU. The other GPU would simply do the collision detection and penetration depth computations continuously, without being interrupted for anything.

The core concept could also be improved. For example, holes in the point cloud are an issue for the graph-based approach because it can cause tunneling of the traversal to the wrong side of the surface, leaving all spheres to be considered inside. The edge detection that I presented only works for a single edge, this could be increased to more. Another possibility would be to implement a more general solution that groups similarly located points and normals with high depth change values together and fitting a plane per group. The mentioned problem with the direction of the forces of the graph-based approach is another issue that can be improved.

# APPENDIX

# LIST OF FIGURES

# LIST OF ALGORITHMS

# BIBLIOGRAPHY


[1] Nina Amenta and Marshall Bern. "Surface Reconstruction by Voronoi Filtering." In: *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*. SCG '98. Minneapolis, Minnesota, USA: ACM, 1998, pp. 39–48. ISBN: 0-89791-973-4. DOI: 10.1145/276884.276889. URL: http://doi.acm.org/10.1145/276884.276889.

[2] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. "A New Voronoi-based Surface Reconstruction Algorithm." In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 415–421. ISBN: 0-89791-999-8. DOI: 10.1145/280814.280947. URL: http://doi.acm.org/10.1145/280814.280947.

[3] H. Badino, D. Huber, Y. Park, and T. Kanade. "Fast and accurate computation of surface normals from range images." In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. May 2011, pp. 3084–3091. DOI: 10.1109/ICRA.2011.5980275.

[4] Gino Van Den Bergen. "Proximity queries and penetration depth computation on 3D game objects." In: *In Game Developers Conference*. 2001.

[5] Gino van den Bergen. "Efficient Collision Detection of Complex Deformable Models Using AABB Trees." In: *J. Graph. Tools* 2.4 (Jan. 1998), pp. 1–13. ISSN: 1086-7651. DOI: 10.1080/10867651.1997.10487480. URL: http://dx.doi.org/10.1080/10867651.1997.10487480.

[6] Michael Bosse and Robert Zlot. "Map Matching and Data Association for Large-Scale Two-dimensional Laser Scan-based SLAM." In: *Int. J. Rob. Res.* 27.6 (June 2008), pp. 667–691. ISSN: 0278-3649. DOI: 10.1177/0278364908091366. URL: http://dx.doi.org/10.1177/0278364908091366.

[7] Alexandre Boulch and Renaud Marlet. "Fast and Robust Normal Estimation for Point Clouds with Sharp Features." In: *Computer Graphics Forum* 31.5 (Aug. 2012), pp. 1765–1774. ISSN: 0167-7055. DOI: 10.1111/j.1467-8659.2012.03181.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2012.03181.x.

[8] S. Chan, F. Conti, N. H. Blevins, and K. Salisbury. "Constraint-based six degree-of-freedom haptic rendering of volume-embedded isosurfaces." In: *World Haptics Conference (WHC), 2011 IEEE*. 2011, pp. 89–94. DOI: 10.1109/WHC.2011.5945467.

[9] Tamal K. Dey, Gang Li, and Jian Sun. "Normal Estimation for Point Clouds: A Comparison Study for a Voronoi Based Method." In: *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics*. SPBG'05. New York, USA: Eurographics Association, 2005, pp. 39–46. ISBN: 3-905673-20-7. DOI: 10.2312/SPBG/SPBG05/039-046. URL: http://dx.doi.org/10.2312/SPBG/SPBG05/039-046.

[10] Tamal K. Dey and Jian Sun. "Normal and Feature Approximations from Noisy Point Clouds." In: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science: 26th International Conference, Kolkata, India, December 13-15, 2006. Proceedings*. Ed. by S. Arun-Kumar and Naveen Garg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 21–32. ISBN: 978-3-540-49995-4. DOI: 10.1007/11944836_5. URL: http://dx.doi.org/10.1007/11944836_5.

[11] N. R. El-Far, N. D. Georganas, and A. El Saddik. "An algorithm for haptically rendering objects described by point clouds." In: *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*. 2008, pp. 001443–001448. DOI: 10.1109/CCECE.2008.4564780.

[12] Naim R. El-Far, Nicolas D. Georganas, and Abdulmotaleb El Saddik. "Collision Detection and Force Response in Highly-Detailed Point-Based Hapto-Visual Virtual Environments." In: *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*. DS-RT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 15–22. ISBN: 0-7695-3011-7. DOI: 10.1109/DS-RT.2007.17. URL: http://dx.doi.org/10.1109/DS-RT.2007.17.

[13] François Faure, Sébastien Barbier, Jérémie Allard, and Florent Falipou. "Image-based Collision Detection and Response between Arbitrary Volumetric Objects." In: *ACM Siggraph/Eurographics Symposium on Computer Animation, SCA 2008, July, 2008*. Dublin, Irlande, July 2008.

[14] Mauro Figueiredo, Joao Oliveira, Bruno Araujo, and Joao Madeiras. "AN EFFICIENT COLLISION DETECTION ALGORITHM FOR POINT CLOUD MODELS." In: *Proceedings of Graphicon*. 2010.

[15] Susan Fisher and Ming Lin. "Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields." In: *Proc. International Conf. on Intelligent Robots and Systems (IROS)*. 2001, pp. 330–336.

[16] S. Gottschalk, M. C. Lin, and D. Manocha. "OBBTree: A Hierarchical Structure for Rapid Interference Detection." In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 171–180. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237244. URL: http://doi.acm.org/10.1145/237170.237244.

[17] A. Gregory, A. Mascarenhas, S. Ehmann, Ming Lin, and D. Manocha. "Six degree-of-freedom haptic display of polygonal models." In: *Visualization 2000. Proceedings.* 2000, pp. 139–146. DOI: 10.1109/VISUAL.2000.885687.

[18] Antonin Guttman. "R-trees: a dynamic index structure for spatial searching." In: *SIGMOD Rec.* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266. URL: http://doi.acm.org/10.1145/971697.602266.

[19] Dirk Holz, Stefan Holzer, Radu Bogdan Rusu, and Behnke. "Real-time Plane Segmentation Using RGB-D Cameras." In: *Robot Soccer World Cup XV*. Ed. by Thomas Röfer, Norbert Michael Mayer, Jesus Savage, and Uluç Saranlı. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 306–317. ISBN: 978-3-642-32059-0. URL: http://dl.acm.org/citation.cfm?id=2554542.2554572.

[20] S. Holzer, R. B. Rusu, M. Dixon, S. Gedikli, and N. Navab. "Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2012, pp. 2684–2689. DOI: 10.1109/IROS.2012.6385999.

[21] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. "Surface Reconstruction from Unorganized Points." In: *SIGGRAPH Comput. Graph.* 26.2 (July 1992), pp. 71–78. ISSN: 0097-8930. DOI: 10.1145/142920.134011. URL: http://doi.acm.org/10.1145/142920.134011.

[22] P. M. Hubbard. "Interactive collision detection." In: *1993 (4th) International Conference on Computer Vision* (1993), pp. 24–31. URL: http://dx.doi.org/10.1109/VRAIS.1993.378267.

[23] K. Jordan and P. Mordohai. "A quantitative evaluation of surface normal estimation in point clouds." In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sept. 2014, pp. 4220–4226. DOI: 10.1109/IROS.2014.6943157.

[24] Max Kaluschke, Uwe Zimmermann, Marinus Danzer, Gabriel Zachmann, and René Weller. "Massively-Parallel Proximity Queries for Point Clouds." In: *Virtual Reality Interactions and Physical Simulations (VRIPhys)*. Bremen, Germany: Eurographics Association, Sept. 2014.

[25] Tero Karras. *Thinking Parallel, Part II: Tree Traversal on the GPU*. Nov. 2012. URL: http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/.

[26] M. Khouil, N. Saber, and M. Mestari. "Collision detection for three dimension objects in a fixed time." In: *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*. 2014, pp. 235–240. DOI: 10.1109/CIST.2014.7016625.

[27] Young J. Kim, Miguel A. Otaduy, Ming C. Lin, and Dinesh Manocha. "Six-degree-of-freedom Haptic Rendering Using Incremental and Localized Computations." In: *Presence: Teleoper. Virtual Environ.* 12.3 (June 2003), pp. 277–295. ISSN: 1054-7460. DOI: 10.1162/105474603765879530. URL: http://dx.doi.org/10.1162/105474603765879530.

[28] Klaas Klasing, Daniel Althoff, Dirk Wollherr, and Martin Buss. "Comparison of Surface Normal Estimation Methods for Range Sensing Applications." In: *Proceedings of the 2009 IEEE International Conference on Robotics and Automation*. ICRA'09. Kobe, Japan: IEEE Press, 2009, pp. 1977–1982. ISBN: 978-1-4244-2788-8. URL: http://dl.acm.org/citation.cfm?id=1703435.1703753.

[29] Jan Klein and Gabriel Zachmann. "Point Cloud Collision Detection." In: *Computer Graphics forum (Proc. EUROGRAPHICS)*. Ed. by M.-P. Cani and M. Slater. Vol. 23. Grenoble, France, 2004, pp. 567–576. URL: http://www.gabrielzachmann.org/.

[30] Jan Klein and Gabriel Zachmann. "Interpolation Search for Point Cloud Intersection." In: *Proc. of WSCG 2005*. University of West Bohemia, Plzen, Czech Republic, 2005, pp. 163–170. ISBN: 80-903100-7-9. URL: http://www.gabrielzachmann.org/.

[31] P. Kumari, K. G. Sreeni, and S. Chaudhuri. "Scalable rendering of variable density point cloud data." In: *World Haptics Conference (WHC), 2013*. 2013, pp. 91–96. DOI: 10.1109/WHC.2013.6548390.

[32] L. Kurnianggoro and K. H. Jo. "Free road space estimation based on surface normal analysis in organized point cloud." In: *2014 IEEE International Conference on Industrial Technology (ICIT)*. Feb. 2014, pp. 609–613. DOI: 10.1109/ICIT.2014.6895000.

[33] Christian Lauterbach, Q. Mo, and Dinesh Manocha. "gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries." In: *Comput. Graph. Forum* 29.2 (2010), pp. 419–428. URL: http://dblp.uni-trier.de/db/journals/cgf/cgf29.html\#LauterbachMM10.

[34] A. Leeper, S. Chan, K. Hsiao, M. Ciocarlie, and K. Salisbury. "Constraint-based haptic rendering of point data for teleoperated robot grasping." In: *2012 IEEE Haptics Symposium (HAPTICS)*. 2012, pp. 377–383. DOI: 10.1109/HAPTIC.2012.6183818.

[35] A. Leeper, S. Chan, and K. Salisbury. "Point clouds can be represented as implicit surfaces for constraint-based haptic rendering." In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. 2012, pp. 5000–5005. DOI: 10.1109/ICRA.2012.6225278.

[36] Y. Li, M. Tang, S. Zhang, and Y. J. Kim. "Six-degree-of-freedom haptic rendering using translational and generalized penetration depth computation." In: *World Haptics Conference (WHC), 2013*. 2013, pp. 289–294. DOI: 10.1109/WHC.2013.6548423.

[37] C. Liu, D. Yuan, and H. Zhao. "3D point cloud denoising and normal estimation for 3D surface reconstruction." In: *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. Dec. 2015, pp. 820–825. DOI: 10.1109/ROBIO.2015.7418871.

[38] M. Liu, F. Pomerleau, F. Colas, and R. Siegwart. "Normal estimation for pointcloud using GPU based sparse tensor voting." In: *2012 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. Dec. 2012, pp. 91–96. DOI: 10.1109/ROBIO.2012.6490949.

[39] David Mainzer and Gabriel Zachmann. "CDFC: Collision Detection Based on Fuzzy Clustering for Deformable Objects on GPU's." In: *WSCG 2013 - POSTER Proceedings*. Vol. 21. 3. Poster. Plzen, Czech Republic, July 2013, pp. 5–8. ISBN: 978-80-86943-76-3.

[40] Z. C. Marton, R. B. Rusu, and M. Beetz. "On fast surface reconstruction methods for large and noisy point clouds." In: *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*. 2009, pp. 3218–3223. DOI: 10.1109/ROBOT.2009.5152628.

[41] Zoltan Csaba Marton, Radu Bogdan Rusu, and Michael Beetz. "On Fast Surface Reconstruction Methods for Large and Noisy Datasets." In: *in Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2009.

[42] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. "Six Degree-of-freedom Haptic Rendering Using Voxel Sampling." In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 401–408. ISBN: 0-201-48560-5. DOI: 10.1145/311535.311600. URL: http://dx.doi.org/10.1145/311535.311600.

[43] Niloy J. Mitra and An Nguyen. "Estimating Surface Normals in Noisy Point Cloud Data." In: *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*. SCG '03. San Diego, California, USA: ACM, 2003, pp. 322–328. ISBN: 1-58113-663-3. DOI: 10.1145/777792.777840. URL: http://doi.acm.org/10.1145/777792.777840.

[44] Jia Pan, Sachin Chitta, and Dinesh Manocha. "Probabilistic Collision Detection between Noisy Point Clouds using Robust Classification." In: *International Symposium on Robotics Research*. Flagstaff, Arizona, 2011. URL: http://www.isrr-2011.org/ISRR-2011//Program_files/Papers/Pan-ISRR-2011.pdf.

[45] Diego C. Ruspini, Krasimir Kolarov, and Oussama Khatib. "The Haptic Display of Complex Graphical Environments." In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 345–352. ISBN: 0-89791-896-7. DOI:

10.1145/258734.258878. URL: http://dx.doi.org/10.1145/258734.258878.

[46] R. B. Rusu, I. A. ¿ucan, B. Gerkey, S. Chitta, M. Beetz, and L. E. Kavraki. "Real-time perception-guided motion planning for a personal robot." In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2009, pp. 4245–4252. DOI: 10.1109/IROS.2009.5354396.

[47] F. Rydén and H. J. Chizeck. "A Proxy Method for Real-Time 3-DOF Haptic Rendering of Streaming Point Cloud Data." In: *IEEE Transactions on Haptics* 6.3 (2013), pp. 257–267. ISSN: 1939-1412. DOI: 10.1109/TOH.2013.20.

[48] F. Rydén and H. J. Chizeck. "A method for constraint-based six degree-of-freedom haptic interaction with streaming point clouds." In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. 2013, pp. 2353–2359. DOI: 10.1109/ICRA.2013.6630896.

[49] F. Rydén, S. Nia Kosari, and H. J. Chizeck. "Proxy method for fast haptic rendering from time varying point clouds." In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2011, pp. 2614–2619. DOI: 10.1109/IROS.2011.6094673.

[50] Hoi Sheung and Charlie C. L. Wang. "Robust Mesh Reconstruction from Unoriented Noisy Points." In: *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*. SPM '09. San Francisco, California: ACM, 2009, pp. 13–24. ISBN: 978-1-60558-711-0. DOI: 10.1145/1629255.1629258. URL: http://doi.acm.org/10.1145/1629255.1629258.

[51] K. G. Sreeni and S. Chaudhuri. "Haptic rendering of dense 3D point cloud data." In: *2012 IEEE Haptics Symposium (HAPTICS)*. 2012, pp. 333–339. DOI: 10.1109/HAPTIC.2012.6183811.

[52] C. Tomasi and R. Manduchi. "Bilateral Filtering for Gray and Color Images." In: *Proceedings of the Sixth International Conference on Computer Vision*. ICCV '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 839–. ISBN: 81-7319-221-9. URL: http://dl.acm.org/citation.cfm?id=938978.939190.

[53] Caihua Wang, H. Tanahashi, H. Hirayu, Y. Niwa, and K. Yamamoto. "Comparison of local plane fitting methods for range data." In: *2001. Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. Vol. 1. 2001, pp. 663–669. DOI: 10.1109/CVPR.2001.990538.

[54] Jan W. Weingarten, Gabriel Gruener, and Alpnach Dorf. "Probabilistic plane fitting in 3d and an application to robotic mapping." In: *IEEE Int. Conf. on Robotics and Automation (ICRA)*. 2004, pp. 927–932.

[55] Rene Weller, Udo Frese, and Gabriel Zachmann. "Parallel Collision Detection in Constant Time." In: *Virtual Reality Interactions and Physical Simulations (VRIPhys)*. Lille, France: Eurographics Association, Nov. 2013.

[56] René Weller, David Mainzer, Mikel Sagardia, Thomas Hulin, Gabriel Zachmann, and Carsten Preusche. "A benchmarking suite for 6-DOF real time collision response algorithms." In: *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology*. VRST '10. Hong Kong: ACM, 2010, pp. 63–70. ISBN: 978-1-4503-0441-2. DOI: http://doi.acm.org/10.1145/1889863.1889874. URL: http://cg.in.tu-clausthal.de/publications.shtml#vrst2010.

[57] Rene Weller and Gabriel Zachmann. "A Unified Approach for Physically-Based Simulations and Haptic Rendering." In: *Sandbox 2009: ACM SIGGRAPH Video Game Proceedings*. New Orleans, LA, USA: ACM Press, Aug. 2009. URL: http://cg.in.tu-clausthal.de/research/ist.

[58] Rene Weller and Gabriel Zachmann. "Inner Sphere Trees for Proximity and Penetration Queries." In: *2009 Robotics: Science and Systems Conference (RSS)*. Seattle, WA, USA, June 2009. URL: http://cg.in.tu-clausthal.de/research/ist.

[59] Rene Weller and Gabriel Zachmann. "Stable 6-DOF Haptic Rendering with Inner Sphere Trees." In: *International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, (IDETC/CIE)*. CIE/VES Best Paper Award. San Diego, CA, USA: ASME, Aug. 2009. URL: http://cg.in.tu-clausthal.de/research/ist.

[60] René Weller and Gabriel Zachmann. "ProtoSphere: A GPU-Assisted Prototype Guided Sphere Packing Algorithm for Arbitrary Objects." In: *ACM SIGGRAPH ASIA 2010 Sketches*. Seoul, Republic of Korea: ACM, Dec. 2010, 8:1–8:2. ISBN: 978-1-4503-0523-5. DOI: http://doi.acm.org/10.1145/1899950.1899958. URL: http://cg.in.tu-clausthal.de/research/protosphere.

[61] Rene Weller and Gabriel Zachmann. "Inner Sphere Trees and Their Application to Collision Detection." In: *Virtual Realities*. Ed. by Sabine Coquillart, Guido Brunnett, and Greg Welch. Springer (Dagstuhl), 2011. Chap. 10, pp. 181–202. ISBN: 978-3-211-99177-0. DOI: 10.1007/978-3-211-99178-7.

[62] Liangjun Zhang, Young J. Kim, and Dinesh Manocha. "A Fast and Practical Algorithm for Generalized Penetration Depth Computation." In: *Robotics: Science and Systems Conference (RSS07)*. 2007.

[63] Liangjun Zhang, Young J. Kim, and Dinesh Manocha. "C-DIST: Efficient Distance Computation for Rigid and Articulated Models in Configuration Space." In: *Proceedings of the 2007 ACM Symposium on*

*Solid and Physical Modeling*. SPM '07. Beijing, China: ACM, 2007, pp. 159–169. ISBN: 978-1-59593-666-0. DOI: 10.1145/1236246.1236270. URL: http://doi.acm.org/10.1145/1236246.1236270.

[64] Liangjun Zhang, Young J. Kim, Gokul Varadhan, and Dinesh Manocha. "Generalized Penetration Depth Computation." In: *Proceedings of the 2006 ACM Symposium on Solid and Physical Modeling*. SPM '06. Cardiff, Wales, United Kingdom: ACM, 2006, pp. 173–184. ISBN: 1-59593-358-1. DOI: 10.1145/1128888.1128914. URL: http://doi.acm.org/10.1145/1128888.1128914.

[65] X. Zhang and Y. J. Kim. "Scalable Collision Detection Using p-Partition Fronts on Many-Core Processors." In: *IEEE Transactions on Visualization and Computer Graphics* 20.3 (2014), pp. 447–456. ISSN: 1077-2626. DOI: 10.1109/TVCG.2013.239.

[66] Wei Zhao and Lei Li. "Improved K-DOPs collision detection algorithms based on genetic algorithms." In: *Electronic and Mechanical Engineering and Information Technology (EMEIT), 2011 International Conference on*. Vol. 1. 2011, pp. 338–341. DOI: 10.1109/EMEIT.2011.6022939.

[67] Wei Zhao, Rui pu Tan, and Wen-Hui Li. "Parallel collision detection algorithm based on mixed BVH and OpenMP." In: *System Simulation and Scientific Computing, 2008. ICSC 2008. Asia Simulation Conference - 7th International Conference on*. 2008, pp. 786–792. DOI: 10.1109/ASC-ICSC.2008.4675468.

[68] C. B. Zilles and J. K. Salisbury. "A constraint-based god-object method for haptic display." In: *Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings. 1995 IEEE/RSJ International Conference on*. Vol. 3. 1995, 146–151 vol.3. DOI: 10.1109/IROS.1995.525876.

[69] I. A. Şucan, M. Kalakrishnan, and S. Chitta. "Combining planning techniques for manipulation using realtime perception." In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. 2010, pp. 2895–2901. DOI: 10.1109/ROBOT.2010.5509702.