

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

Maximilian Kaluschke<sup>1</sup>, René Weller<sup>1</sup> and Gabriel Zachmann<sup>1</sup>

**Abstract**—We present a novel method to define the penetration volume between a surface point cloud and arbitrary 3D CAD objects. Moreover, we 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, our algorithm is perfectly suited for streaming point clouds that can be gathered online via depth sensors like the Kinect. We 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.

## I. INTRODUCTION

Haptic rendering for CAD objects, like triangle meshes, is a well studied field. However, today, virtual environments that we interact with, often do not only consist of pre-modelled CAD objects, but often contain highly dynamic parts that are generated online via novel input sensor like the full body tracking of the Kinect or hand tracking with the Leap Motion. The output of these devices is typically not a CAD mesh but an unstructured point cloud. Performing haptic rendering directly on this kind of data is a relatively new field of research.

Obviously, we could simply reconstruct 3D meshes from the point cloud data and then apply the classical haptic rendering techniques like [1]. Unfortunately, mesh reconstruction is time consuming [2] and moreover, these methods often require additional time consuming pre-processing that can be hardly performed in real time, not to speak about frequencies required for haptic rendering, i.e. 1000 Hz. Consequently, it is necessary to perform the haptic rendering directly on the point cloud data.

Haptic rendering usually consist of two parts: first, we have to *find* collision between colliding objects, and second, we have to *resolve* them by applying appropriate forces and torques. Due to the high frequencies in haptics, penalty-based approaches are typically preferred. Obviously, resolving collisions requires additional information about the amount of interpenetration. Basically, there exist three kinds of contact information: 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” [3] 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 [4], [5]. However, they support only collision detection between pairs of watertight 3D CAD objects that must have a certain volume and do not support point clouds until now.

In this paper, we present a novel method for collision detection and resolution between unstructured point cloud data and arbitrary 3D CAD objects. The only pre-condition is that the CAD models have to be watertight and that the points in the point cloud represent a surface. In detail we 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 our volumetric penetration measure that computes forces as well as torques for full 6-DOF haptic rendering.

The main idea is to represent the volume of the CAD object by a polydisperse sphere packing (similar to [5]) and distinguish between parts of those spheres that are inside and outside of the point cloud, based on surface normal information. To do that we propose a novel *neighborhood graph* to identify spheres that are completely located behind the point cloud. The application of traditional data structures for CAD vs CAD collision detection, 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 point cloud tests.

Our algorithms is easy to implement and handles multiple contacts automatically. The results show that our algorithm can perform collision queries at haptic rates for reasonable point cloud sizes that are gathered live via a Kinect.

\*This work was not supported by any organization

<sup>1</sup>Maximilian Kaluschke, René Weller and Gabriel Zachmann are with Faculty of Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany {mxkl, weller, zach}@cs.uni-bremen.de

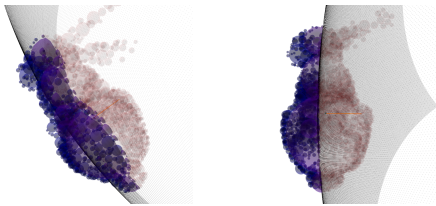


Fig. 1: The penetration volume: Blue parts represent volume that is considered penetrated. Red spheres are outside.

## II. RELATED WORK

Generally, collision detection is used in haptic rendering but also in related fields like physically-based simulations, robotics, or VR and today, there exists a large amount of publications on this topic. Most work on collision detection concentrates on CAD vs CAD collision detection. In this case various bounding volume hierarchies have shown to give efficient results. However, these methods usually support only simple boolean queries.

Especially in haptics more information about the contacts is required to apply appropriate forces and torques. Usually, a minimum vector that defines the minimum translation to separate the objects is used. There also exist generalized formulations that consider general minimum transformations instead of simple translation vectors [6], [7]. However, to our knowledge, none of these approaches can be easily extended to point cloud vs CAD objects.

Another method to define the amount of extend for a pair of intersecting objects is the penetration volume. There exist only very few methods that approximate the penetration volume. Hasegawa et al. [8] explicitly compute the penetration volume for convey polytopes. Faure et al. [4] used layered depth images on the GPU. Their approach supports deformable objects. The approach by Weller et al. [5] is restricted to rigid objects. They use a sphere packing to represent the objects' volume and build a bounding volume hierarchy on top of these inner spheres to accelerate collision queries.

Compared to CAD object representations, the literature on collision detection for point clouds is relatively sparse. Most of the approaches that were presented so far work only for static point clouds. [9] use a CSG representation for the virtual tool and manage to render 10k points. [10] surround each point by an axis aligned box and test against a single point probe. [11] describe a stochastic traversal of a bounding volume hierarchy. In addition to simple boolean collision tests, they support the computation of minimum distances [12]. However, none of these approaches operate on streaming point clouds.

Closely related to point cloud collision detection algorithms is the classic 6-DOF haptic rendering approach – the Voxmap pointshell (VPS) algorithm [13]. The main idea is to divide the virtual environment into a dynamic object, that is allowed to move freely through the virtual space and static objects that are fixed in the world. The static environment is discretized into a set of voxels, whereas the dynamic object

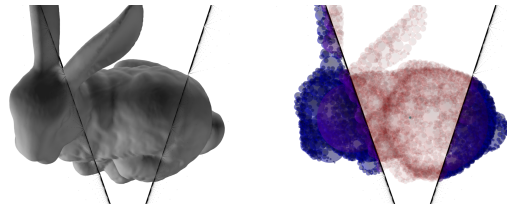


Fig. 2: Penetration volume for multiple contacts. Multiple contacts are automatically handled without need for special handling.

is described by a set of points that represents its surface. During query time, for each of these points it is determined with a simple boolean test, whether it is located in a filled volume element or not. Many extension for the classical VPS algorithms have been proposed [14], [15], [16].

[17] 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 [18] that relies on the classic VPS algorithm. The running time relies 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 [19].

## III. OUR VOLUMETRIC PENETRATION MEASURE

The core of our approach is our volumetric penetration measure. As pre-conditions, it is required that the 3D CAD object is watertight and that each point in the point cloud additionally has a normalized normal pointing into the outside direction<sup>1</sup>

In order to define the penetration volume for a point cloud, we represent the volume of the CAD object by a polydisperse sphere packing. Such sphere packings can be easily pre-computed by the Protosphere algorithm [20]. This algorithm produces space-filling sphere packings for almost any 3D object representation, including polygonal meshes, CSG and NURBS.

A point cloud that intersects such a sphere packing basically divides the spheres into different parts:

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

<sup>1</sup>Please note, in case of sensor generated point clouds, such normals are often not automatically available. In Section V we will describe an algorithm to compute consistent normals for an actual Kinect image.

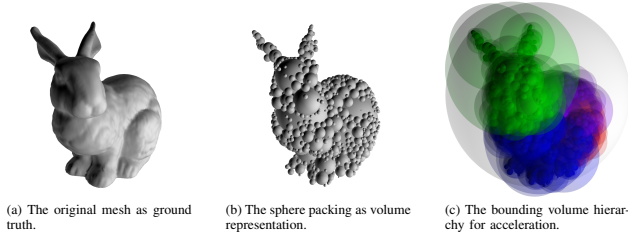


Fig. 3: A polygonal mesh and its inner sphere tree representation.

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  $V = \frac{1}{3} h^2 (3r - h)$  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  $S$  with  $d = |j \cdot n - (c - p) \cdot j|$ , assuming that  $n$  is normalized (see Figure 1).

In case that several different points  $p_i; i = 1; \dots; N$  hit the same boundary sphere, we simply compute the normalized average normal  $n_s$  and the appropriate total volume  $V_s$  per sphere  $S$ :

$$n_s = \frac{\sum_{i=0}^N n_{p_i}}{N} \quad V_s = \frac{\sum_{i=0}^N V_{p_i}}{N}$$

where  $n_i$  are the normals of the points  $p_i$ .

In order to define the inside spheres, we present the notion of our novel *sphere graph*. The idea is to construct a graph data structure based on the spheres in the sphere packing. The nodes of the graph are the spheres. We create an edge between two spheres if they are touching. This creates a connected undirected graph.

Inside spheres are the direct neighbours of the boundary spheres that are located into the opposite directions of the normals, we will call them *entry spheres* and additionally, the neighbours of these spheres. Entry spheres are completely behind the plane spanned by the normal  $n_s$  and the center of the boundary sphere and they are touching  $S$ . Moreover, we recursively count all adjacent spheres of the entry spheres as inside spheres if they are not boundary spheres. Note, the latter inside spheres are not necessarily located behind the planes spanned by the normals. The total penetration volume  $V$  is:

$$V_{total} = \sum_{\text{Boundary spheres } S_i} V_{S_i} + \sum_{\text{Inside spheres } S_j} V_{S_j}$$

In the next section we will present our algorithm to compute this penetration volume efficiently on the GPU.

#### IV. OUR COLLISION DETECTION ALGORITHM

The definition from the previous section draw two challenges: First, finding the boundary spheres and second, computing the inside spheres. In this section we will present algorithms to solve these challenges. We will start with the identification of the boundary spheres.

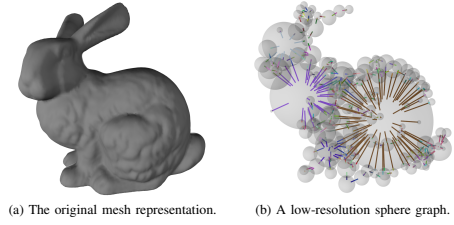


Fig. 4: An object and the corresponding connected graph.

#### A. Boundary Spheres

This task is closely related to traditional collision detection methods. Hence, we use a very similar approach. In a pre-processing step we compute a sphere packing according to [20]. Additionally, we create a wrapped *Inner Sphere Tree* hierarchy based on this sphere packing to accelerate collision queries, similar to [5] (see Figure 3).

We use a typical recursive traversal scheme to find the boundary spheres (see Algorithm 1). This can be easily performed for all spheres in parallel, similar to distance computations described in [21]. Obviously, in addition to simply marking the spheres we can directly sum up the volume and additional data required for collision response calculations.

---

**Algorithm 1:** traverseIST(sphere  $s$ , point  $p$ )

---

```

if  $S$  is leaf then
    mark  $S$  as boundary sphere
forall children  $S_j$  of  $S$  do
    if  $p$  inside  $S_j$  then
        traverseIST( $S_j$ ,  $p$ )

```

---

#### B. Inside Spheres

In order to identify the inside spheres, we need a sphere graph as described in Section III. This can be easily computed in a pre-processing step after the computation of the initial sphere packing. However, the spheres created by Protosphere are not always connected. Hence, we consider two spheres as connected if their distance is smaller than a chosen  $\epsilon$  (in our experiments we used  $\epsilon = \frac{1}{10} r$  where  $r$  is the radius of the smallest sphere). In order to guarantee the connectivity of the graph, we connect unconnected components with the shortest edge using the Euclidean distance. Figure 4 shows an example of the edges of such a sphere graph.

After we have found the boundary spheres, we perform for each of them a graph traversal on the sphere graph in order to

---

**Algorithm 2:** traverseGraph(sphere  $s$ )

---

```

if  $S$  is not marked and not a boundary sphere then
    mark  $S$  as inside sphere
forall edges  $(S; S_j)$  do
    traverseGraph( $S_j$ )

```

---

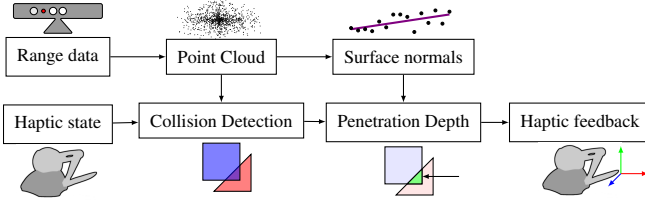


Fig. 5: Our haptic rendering pipeline.

mark the inside spheres and sum up the volume using atomic operations. To do that we use 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 2). Obviously, we traverse those edges pointing away from the normal of the respective boundary sphere. The traversal can be easily parallelized by traversing all boundary spheres in parallel. Moreover, we added CUDA’s dynamic parallelism in our implementation to further optimize the parallelization.

### C. Forces and Torques

For an appropriate collision response we do not only require the penetration volume, but also forces and torques. Actually, we can directly use the penetration volume to define the magnitude of the force per sphere for both, boundary and inside spheres. Obviously, the direction and the magnitude of the forces changes continuously as long as the points and the object move continuously. For the inside spheres we set the magnitude again with respect to the penetration volume, i.e. to the volume of the complete sphere. The normals are interpolated accordingly during the traversal.

Similarly we compute the torques. For each boundary sphere  $S$  that is intersected by points  $p_i; i = 1; \dots; N$ , with force  $f = v n$  where  $v$  is the interpolated volume and  $n$  the interpolated normal we set the interpolated contact point  $p$  and the torque as

$$p = \frac{\sum_{i=0}^N p_i}{N} = (C \quad p) \quad f$$

where  $C$  is the center of mass of the object. For the inside spheres we simply use their centers as contact point.

The forces and torques are defined per sphere. Hence, to get the total force  $f_{total}$  and torque  $total$  acting on the object we can simply sum them up:

$$f_{total} = \sum_{s_i \in S} f_i \quad total = \sum_{s_i \in S} \tau_i$$

where  $f_i$  and  $\tau_i$  are the individual forces and torques of the spheres  $s_i; i = 1; \dots; N$  in a sphere packing  $S$ . The per sphere definition of forces and torques also explains, why our approach automatically handles multiple contacts (see Figure 2).

### V. POINT CLOUD NORMALS

Usually, depth sensors like the Kinect do not provide normals for the points in the point cloud. In this section we will shortly sketch our idea to generate consistent normals

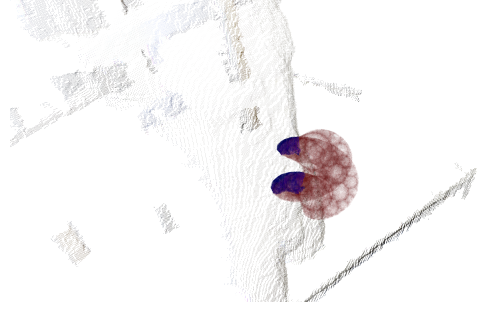


Fig. 6: Our real-world test case: The point cloud is captured from a Kinect. It shows a mannequin and its background. The virtual tool is represented by a twisted torus.

from a depth image. It relies on the simple observation, that we have knowledge about the viewing direction of the sensor and about the neighbourhood of points in the depth image.

Basically, we use a method that relies on fitting a plane through neighboring points using principal component analysis. More precisely, to compute the normal  $n_p$  for point  $p$ , we consider its 7-neighborhood of points  $Q$  by defining the matrix:

$$M_p = \begin{matrix} \times & \begin{matrix} O & P & 1 & O & P & 1 & T \\ \textcircled{p_i} & \frac{p_j}{|Q_j|} & \textcircled{C} & \textcircled{B} & \textcircled{p_i} & \frac{p_j}{|Q_j|} & \textcircled{C} \end{matrix} \end{matrix}$$

and computing that eigenvector  $n_p$  of  $M_p$  that corresponds to the smallest eigenvalue. Actually, the plane can have two different normals, so we take the one that points towards the origin since a camera can only see surfaces that point towards it.

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

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

## VI. RESULTS

### A. Haptic Rendering

We have implemented the whole pipeline of our haptic rendering method for streaming point clouds, i.e. the collision detection, the collision response and the normal calculation for the point cloud (see Figure 5).

The forces and torques are not directly rendered to the device. Instead we use a virtual coupling [22] to improve the stability.

We tested our implementation on a computer running 64bit Windows 10 with an Intel Xeon E5620 CPU, 32GB DDR3 RAM and a NVIDIA GeForce GTX 1080. We performed synthetic as well as real-world experiments to measure the computational time but also the quality of the generated forces and torques. The synthetic tests consist of synthesized point clouds and pre-recorded path. Additionally, we included data generated with our use case scenario described above, i.e. real haptic interaction with real-time generated depth images by a Kinect. However, we captured the depth images and recorded a path in a single-run haptic session

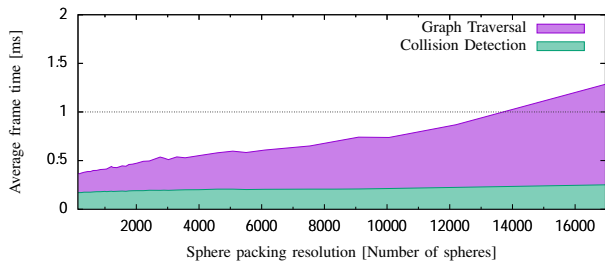


Fig. 7: Performance with respect to the number of spheres. Times are averaged over all the frames. The times are taken from real-world interactions with an average penetration depth of about 5% of the virtual tool’s volume.

and replayed it for the timing. This allows a fair comparison for different sphere resolutions.

The synthetic benchmarks contain situations of heavy interpenetrations (for instance objects passing completely through walls) that hardly happen in real-world applications. In these artificial benchmarks we restricted the degrees of freedom of the motion. More precisely, we included only translations into a single direction. This allows a better investigation of the quality of the forces and torques.

### B. Performance

In the real-world test scenario, the point cloud consists of 300k points according to the resolution of a Kinect depth image. We used a complicated convex object for the haptic tool. The pre-recorded path contains medium to heavy interpenetrations of 5% to nearly 10% of the volume of the haptic tool. We tested several sphere resolutions of up to 17k spheres. Figure 9 shows the time per frame for 1k spheres. Our algorithm is able to compute forces and torques in less than 1ms, even in case of deep inter-penetrations of nearly 10%.

Figure 7 shows the performance of our algorithm with respect to the number of spheres. Additionally, the timings are divided into the search for boundary spheres, i.e. the IST traversal and the search for inside spheres, i.e. the graph traversal. Most of the time is spent for the graph traversal. This also means that the influence of the number of points in the point cloud is relatively small.

Actually, larger sphere packings of up to 14k spheres exceed the 1ms interval required for haptics. It turns out, that larger sphere packings do not necessarily produce better forces (see next section). Actually, the distance between two points is a lower bound for a reasonable minimum sphere size. In real-world applications the minimum sphere size should be even larger in order to avoid noises produced by fast movements of the spheres. However, a theoretical basis for finding the best sphere packing is still an open question that we will further investigate in the future.

### C. Quality

Figure 8 shows the force magnitude for different sphere resolutions in the real-world test case described above. The forces are continuous. However, most of the penetration volume is already discovered when we use only 1k spheres.

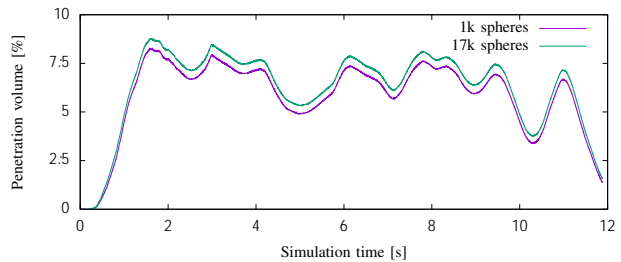


Fig. 8: Penetration volume profile for the real-world benchmark.

Adding over 10k spheres does not change the volume significantly. Actually, we have even observed more noise in the signal of large sphere packings. Consequently, relatively small amounts of spheres are sufficient to produce realistic volumetric forces. This is an advantage of our algorithm, because the number of spheres (and hence, the memory footprint) can be kept small compared to voxel-based approaches or methods that rely on the polygon resolution of the object.

Additionally, we tested artificial test scenarios with synthetic point clouds. We used simple objects that allow an analytic investigation of the expected overlap volumes as ground truth. We added some noise to the point clouds as presented in [23] in order to simulate the noise found in captured depth images. We used three simple objects (a cube, a tetrahedron and an octahedron). The movement was restricted to translation in z-direction. We synthesized two different point clouds (a plain wall and an inverted sphere).

Figure 10 shows the resulting force magnitudes for the sphere scenario. As expected, the cube generates a linearly increasing volume for the plane. Those of the tetrahedron and octahedron are steeper in the middle because of their larger cross section. In case of the inverted sphere point cloud we see a similar behaviour. However, in all cases the results show that our algorithm computes continuous forces for both, direction and magnitude. The torque was, as expected, close to zero in all our test cases.

## VII. CONCLUSION AND FUTURE WORK

We have presented a novel method to define the penetration volume for a point cloud and arbitrary watertight 3D CAD objects. This definition enabled us to develop a new algorithm to compute this penetration measure completely on the GPU. Even more, we used the penetration measure to define continuous forces and torques. In practice, the

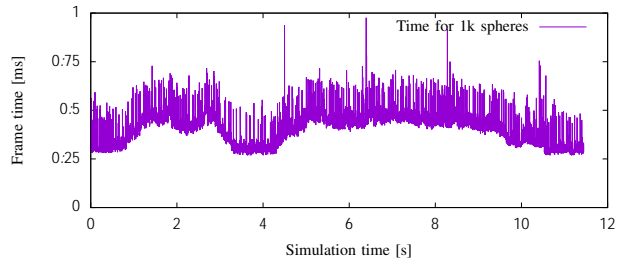


Fig. 9: Time per frame in the real-world test.

