Conference and Exhibition of the European Association of Virtual and Augmented Reality (2014)
G. Zachmann, J. Perret, and A. Amditis (Editors)

A New Fast and Robust Collision Detection and Force Computation Algorithm Applied to the Physics Engine Bullet: Method, Integration, and Evaluation

Mikel Sagardia [†], Theodoros Stouraitis[‡], and João Lopes e Silva[§]

German Aerospace Center (DLR),
Institute of Robotics and Mechatronics, Germany

Abstract

We present a collision detection and force computation algorithm based on the Voxelmap-Pointshell Algorithm which was integrated and evaluated in the physics engine Bullet. Our algorithm uses signed distance fields and point-sphere trees and it is able to compute collision forces between arbitrary complex shapes at simulation frequencies smaller than 1 msec. Utilizing sphere hierarchies, we are able to rapidly detect likely colliding areas, while the point trees can be used for processing colliding regions in a level-of-detail manner. The integration into the physics engine Bullet was performed inheriting interface classes provided in that framework. We compared our algorithm with Bullet's native GJK, GJK with convex decomposition, and GImpact, varying the resolution and the scenarios. Our experiments show that our integrated algorithm performs with similar computation times as the standard collision detection algorithms in Bullet if low resolutions are chosen. With high resolutions, our algorithm outperforms Bullet's native implementations and objects behave realistically.

Categories and Subject Descriptors (according to ACM CCS): Computing Methodologies [I.3.5]: Computational Geometry and Object Modeling—Geometric algorithms, Object hierarchies; Computing Methodologies [I.3.7]: Three-Dimensional Graphics and Realism—Animation, Virtual reality.

1. Introduction

Methods that perform collision detection and force computation are essential in virtual reality applications, namely for interactive gaming, virtual prototyping, or assembly simulations with force feedback [SWH*12]. In the latter, many available solutions try to find a trade-off between the high computational speed (1 kHz) required by haptics for stable and realistic interaction [BS02] and the obtained accuracy by using simplified geometries. This might result in unrealistic simulations. Similarly, collision detection is still a bottleneck for many realtime motion simulators; although the scenarios are rendered at lower frequencies (60 Hz), reduced representation of real objects are often used, such as primitive shapes (e.g., spheres, boxes, etc.) or convex hulls.

The physics library Bullet [Cou14] provides with several collision detection implementations able to handle simple geometries and a powerful rigid body dynamics framework. Yet, the performance of the library decreases when realistic complex objects are used in the virtual scene. Therefore, the implementation of a fast collision detection algorithm would be useful, for instance, to enable haptic interactions and to simulate realistic multibody environments.

In this work, we present our improvement of the well known Voxelmap-Poinsthell (VPS) haptic rendering algorithm [MPT99] and its integration into the physics engine Bullet as a fast collision detection module. Our implementation uses point-sphere hierarchies and distance fields for each colliding pair. Thanks to the sphere trees it is possible to recognize very fast possibly colliding areas. Moreover, the point tree enables level-of-detail traverse of the surfaces, being possible to provide with a fair contact manifold on a established time budget, even for extremely non-convex geometries.

† e-mail: mikel.sagardia@dlr.de
‡ e-mail: theodoros.stouraitis@dlr.de
§ e-mail:jpedro.e.silva@gmail.com

This paper is organized as follows: In the next sections we discuss related work and specify our contributions. In Section 2, we describe the principles of the VPS algorithm for both its classical penalty-based approach and hierarchical collision detection. In Section3, the integration in Bullet is explained. The evaluation and the results are then presented in Section 4. Finally, in the last section, we present our conclusions and possible future improvements.

1.1. Related Work

One well-known approach for collision detection is the Gilbert-Johnson-Keerthi (GJK) Algorithm [GJK88], which computes distances between convex shapes using their Minkowski sums. This algorithm was also extended to compute penetration values [vdB01]; both approaches are implemented in Bullet. Mamou and Ghorbel [MG09] find convex patches in an initial non-convex object by hierarchically clustering and decimating the mesh. The result is a convex decomposition to which the GJK algorithm can be applied. This approach is also implemented in Bullet and its performance serves as a benchmark in our experiments (see Section 4). It is possible to track the closest points by observing the Voronoi regions of the mesh features, as done by Lin and Canny [LC91]. In this approach, the high spatio-temporal coherence of the collision detection problem has a decisive relevance, since the tracking can be accelerated by assuming that closest points in the next instant will be close to the previous ones.

Many other collision detection approaches based on polyhedral models use bounding volume hierarchies in order to accelerate the queries, for instance, axis aligned (AABB) or object oriented bounding boxes (OBB) [GLM96]. Redon et al. [RKC02] also make use of OBB hierarchies upon polygon soups and apply a continuous collision detection method based on interval arithmetics. This approach is able to compute the first contact instant. Continuous collision detection is also implemented upon Minkowski sums in Bullet [Cou05].

The quadratic nature of multibody environments has also been approached with sweep and prune methods [CLMP95]. These algorithms sort lower and upper limits of objects' bounding boxes and discard collision pairs according to the separating axis theorem, simplifying the quadratic problem. Bullet has its own sweep and prune method in the broad-phase collision detection (see Section 3).

Bullet provides interfaces to integrate third-party collision detection modules. However, to the best of our knowledge, our approach is the first to be published and available, apart from GImpact [Leo07], which handles concave meshes and is also tested in our experiments (see Section 4).

An extensive state-of-the-art compilation of haptic rendering methods is given in [LOLO08]. A multi-resolution representation of the objects that enables time-critical force computation while assuring sensation preserving is built in [OL03] out of polyhedral models. For that, objects are hierarchically segmented in convex patches. One haptic rendering algorithm that dispenses with polygonal models was presented by Weller and Zachmann [WZ09]. This method bounds objects from inside with non-overlapping spheres which are organized into a hierarchy. This approach enables proximity and intersection volume queries, which are reached in haptic rates thanks to the fast sphere overlap checks.

Distance and volume penetration data can be used to compute penalty-based forces. Some other approaches compute impulse-based forces out of contact information [MC94], [CSC05], and it is possible to generate constraint-based forces [ORC07]. The latter work introduces a six-DoF generalization of the Gauss' least constraint principle often referred to as the *God-Object* method.

Our collision detection and force computation approach is based on the Voxmap-Pointshell (VPS) Algorithm [MPT99], [MPT06], which is originally a penalty-based haptic rendering algorithm. The great advantage of this algorithm is that it is able to cope with arbitrarily complex geometries. A more comprehensive description of the algorithm will be provided in Section 2.1.

The VPS Algorithm was improved by Barbič and James [BJ08] to support deformable objects. In that work, hierarchical data structures and signed distance fields are used. Similarly, we developed a haptic rendering algorithm for rigid bodies based on the VPS Algorithm which also uses hierarchies and distance fields. However, our data structures are optimized for fast and accurate collision and proximity queries rather than for deformation simulations.

Recently, constraint-based force computation has been implemented to the VPS algorithm [RC13]. This work processes streaming point-clouds, being possible to perform haptic rendering with devices similar to the Kinect.

Important research has been conducted in the past years regarding multibody dynamics simulation [Bar92], [Erl05], [BETC12]. We focus on the contact computation problem –which influences greatly the force and motion generation problem– in this work. Therefore, we have considered physics engines' evaluation works [SR06], [BB07], in order to analyze and select a physics engine. In this line, a recent evaluation [HWS*12] showed after performing exhaustive benchmarking tests amongst five publicly available engines that Bullet behaves indeed robustly and ranks always in the best positions –except when restitution scenarios are tested. Due to these good results, and also due to the facts that it is a popular, actively maintained engine with a clear nice-to-use open source code, we decided to select this engine for our work.

1.2. Contributions

We can summarize the main contributions of our work as follows:

- **Implementation of a fast and accurate collision detection method** based on the haptic rendering algorithm VPS. Our algorithm uses point-sphere hierarchies and signed distance fields for each colliding pair, enabling fast contact, distance, and penetration queries for arbitrarily complex geometries. Although we are able to simultaneously generate penalty forces, we rather create a contact manifold for later processing.
- **Integration** of our algorithm as a plugin into the physics engine **Bullet**.
- **Evaluation** of our algorithm by comparing it with other collision detection algorithms within Bullet, such as GJK, GJK with convex decomposition, and GImpact. The results show that our algorithm is as fast as the compared methods for low resolutions and several orders of magnitude faster –always below 1 ms— for higher resolutions. Furthermore, objects display a realistic motion behavior.
- We enable **haptic and motion rendering** within the physics engine Bullet.

Our implementation has been tested with Bullet 2.82, but is easily portable to other versions and engines by using our API. Additionally, the complementary video shows a summary of the above mentioned evaluation results.

2. Collision Detection and Force Computation Algorithm

We base our collision detection and force computation method on the Voxelmap-Pointshell (VPS) Algorithm [MPT99], [MPT06]. This approach uses two data structures for each pair of colliding objects: (*i*) *voxelmaps* or voxelized representations and (*ii*) *pointshells* or point-sampled structures (see Figure 1).

Voxelmaps are 3D-grids in which each voxel contains a discrete layer value that approximates the distance to the surface [MPT06] (see Figure 3-(a)). In a similar fashion as in [BJ08], we extended voxelmaps to contain floating point distance fields. However, we implemented a mixed data structure that contains both approximative layer values and real distances, which can be locally interpolated in contact areas. This approach allows for fast and accurate penetration and distance computations free of aliasing artifacts while using more modest memory requirements.

Pointshells are point-clouds that sample object surfaces, having each point a normal pointing inwards of the object. Similarly to [BJ08], we implemented a point-sphere tree above the plain *point-soup*. However, we seek a down-top building approach starting with a high point sampling resolution where points are uniformly distributed. Aditionally, we bound point clusters with minimal enclosing spheres [FG04], in contrast to the approach in [BJ08], where sphere centers are located on the object's surface enabling faster deformation computations. This hierarchy allows for fast collision area localization by means of the sphere tree and using high point sampling resolutions in a time-critical manner.

After the offline generation of these haptic data structures, the online collision query algorithm consists in traversing the point-sphere hierarchy detecting the likely colliding regions of the pointshell. The penetration or distance values are computed for the points in this region, yielding penalty collision forces. However, we provide the set of colliding points and penetrations to Bullet in order to let the physics engine simulate the movement and collision forces.

An important advantage of algorithms that use voxelized and point-sampled structures is that their speed depends mainly on the sampling of the object and not on its geometry. This is achieved by pre-computing (offline) as much haptic information as possible of the objects in the scene and storing it in our haptic data structures: voxelmaps and pointshells.

In the following, we explain briefly the generation and properties of our data structures (Section 2.1). Then, the computation of distance, penetration and penalty forces is explained (Section 2.2), and finally, several approaches for hierarchy traversal during collision detection are presented (Sections 2.3 and 2.4).

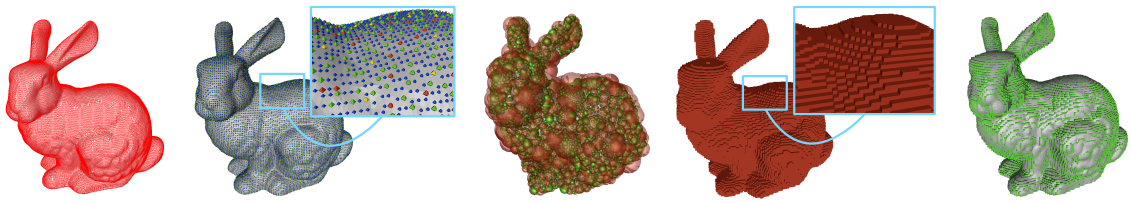
2.1. Data Structures

Given a triangle mesh, we first generate a layered voxelmap and a plain *point-soup* representation of the objects in the order of magnitude of seconds using the methods presented in [SHPH08]. Then, we extend these data structures before the online collision detection takes place.

2.1.1. Signed Distance Fields

Initially, the polygonal model is placed in an empty voxel grid. Our voxelmaps are sized with the bounding box of the object and they conform a regular grid. Each voxel is a cubical cell of edge size or *voxel size s*. Additionally, each voxel contains an integer that represents the distance from the voxel center to the surface of the object. The triangles of the initial mesh are checked for collision against the voxels in their bounding box using the Separating Axis Theorem [AM05]. These overlapping voxels are assigned with a voxel value $v = 0$. Then, using a modified scanline filling algorithm [KS86], voxels in the n^{th} inner layer are assigned $v = n$, whereas voxels in the n^{th} outer layer have $v = -n$. Therefore, our layered voxelmap is a discrete signed distance field.

Once the layered voxelmap is created, we can additionally build more accurate distance fields upon it. The first step



(a) (b) (c) (d) (e)

Figure 1: Different representations of the Stanford Bunny: (a) Triangle mesh with 35606 vertices; (b) Several point tree levels of the bunny coded with colors; (c) Two successive sphere tree levels of the bunny (the red transparent is the upper level); (d) Voxelized representation of the bunny (surface voxels in red); (e) Voxelized representation of the bunny (first inner layer in green).

consists in storing the closest point on the mesh for each surface voxel. Next, we sweep all other voxels and detect their closest surface voxel. This is achieved by moving in the direction of the gradient of the layered voxelmap in the analyzed voxel center until the first surface voxel is found. Thus, for each voxel we have the approximate closest point on the mesh.

We implemented two ways to encode these closest surface points per voxel. In the first option, we build a support voxelmap where the voxels contain double precision floating point distance values. In the second encoding, we update the existing voxelmap with the newly generated values by converting them to integers. For that, the real distances (or penetrations) d are truncated according to a precision value p –the same for the whole voxelmap–. The approximative distance (or penetration) $\tilde{d}$ can be obtained online with the scaling factor λ, as it is shown in (1):

$$\left.\begin{array}{l} d = 1.61803 \\ p = 3 \end{array}\right\} \rightarrow \left\{\begin{array}{l} v = 1618 \\ p = 3 \\ \tilde{d} = v\lambda = v10^{-p} = 1.618. \end{array}\right. \quad (1)$$

In both cases, the distance is signed, as in the layered voxelmap: inner voxels get positive values (penetrations), whereas outer voxels are assigned negative distances.

The main difference between all three maps is their size. A layered voxelmap can be easily compressed, whereas the scaled distance field and the double precision distance field require increasingly bigger memory spaces. We choose the representation and the resolution –i.e., voxel size s and precision p, if required– depending on the models and the simulations we want to perform.

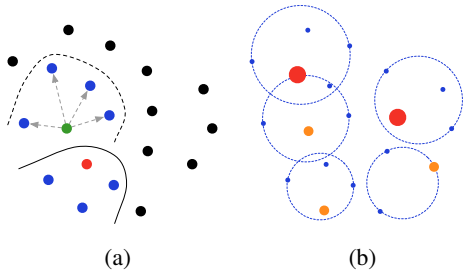
All voxelmaps used for this work were generated within less than 10 seconds and have a size of around 1 MB. Refer to [SHPH08] for more information on generation time.

2.1.2. Point-Sphere Trees

In order to generate the plain pointshell, we require the initial triangle mesh and the layered voxelmap introduced in the previous Section 2.1.1. Each triangle is visited during the process and the surface voxel ($v = 0$) centers in its bounding box are projected onto the triangle. The projected points are considered valid if they are no closer than αs to the points that were already generated for the analyzed triangle and the neighbor triangles, being s the voxel size and α a constant value close to $\sqrt{2}$. Our implementation yields a point cloud with uniformly sampled 3D points. Next, we compute an inwards pointing normal vector for each point. For that, the voxelmap neighborhood of the point is used: the gradient of the distance field yields the normal vector. The final result is a list of 6D unordered points that represent the original mesh.

Once the plain pointshell is generated, we proceed with the construction of the point-sphere tree. First, points are grouped into clusters of K points according to their *similarity*. In our current implementation this similarity criterion is the Euclidean distance, but it is possible to observe the geodesic distance and also the normals of the points. Note that $1 \leq K \leq N_P$ is a parameter selected by the user, being N_P the number of points in the plain pointshell. Figure 2-(a) shows the clustering process with $K = 4$. The first point is randomly chosen and the next most similar K points are located in the neighborhood. A cluster of K elements is formed using the initial point and the $K-1$ most similar –closest, in our implementation– neighbors. The K^{th} most similar neighbor point is used as an initial point for the next cluster. Before jumping to the next cluster, the point in the cluster most similar to the average is selected as the parent point. This process is repeated until all the elements belong to some cluster. When this occurs, the algorithm starts grouping the parent points of the previously defined clusters. The stopping criterion of this recursion is met when we reach the top level of the tree which contains only one cluster.

This sequential clustering approach can return clusters containing points that are much further away from each other than would otherwise be expected. This occurs because the only criterion for starting to build the next cluster is finding the closest point to the current cluster. To eliminate these exceptions, we first tag all the points that lie further than a



(a) (b)

Figure 2: (a) Point clustering: K most similar (closest) points are grouped together, being $K = 4$ in this example; (b) Hierarchized points –coded with color and size– and minimally bounding spheres around point clusters.

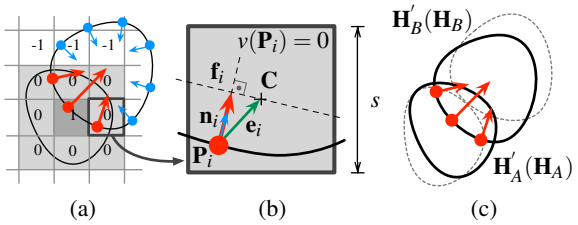
threshold distance to their parent cluster point. When a point is tagged, the center of mass of its cluster is recomputed and we check whether the other points respect the threshold, and they are tagged if it corresponds. After the clustering on a certain level is finished, we sweep the tagged points and compute the distance between them and all parent points present in the level. If the minimum distance is under the threshold, we assign the point to the corresponding cluster. If not, the point conforms a cluster on its own, which contains a unique element. Therefore, although we can expect that the clusters have on average K children, this amount can vary in practice. Note that each level of the point hierarchy represents with points the whole object with a different resolution. A level $l+1$ has K times more points than its predecessor l.

After building the point-tree, we create a sphere-tree upon it. For that, we simply compute bounding spheres for each cluster. In contrast to Barbič and James, our approach generates minimally bounding spheres [FG04] that contain all the cluster points and all the children cluster points until reaching the leaf of the processed cluster. Thus, we generate an optimally wrapped sphere-tree, similarly to [WZ09], since each cluster sphere contains all its children points, but not the children spheres. Figure 2-(b) shows the minimally bounding spheres. This sphere structure enables rapidly locating likely collision areas performing sphere checks.

All pointshells used for this work were generated within less than 20 seconds and have a size of around 2 MB. Refer to [SHPH08] for more information on generation time.

2.2. Proximity Queries and Penalty-Based Force Computation

This section presents the distance and penetration computation of the pointshell points without considering the point traverse and selection problem. In order to have an insight on our hierarchical traverse algorithm see Section 2.3. A straightforward approach would be traversing all the points in the pointshell [MPT99], in case the points are not ordered in a tree. It is also possible to use GPGPU to select the likely colliding points without the need of hierarchies, as done by Rydén and Chizeck [RC13]. The authors place a bounding box around the projection of the virtual tool (voxelmap) onto the depth image (pointshell) in their implementation.



(a) (b) (c)

Figure 3: (a) Voxelized and point-sampled objects in collision. Each voxel has its voxel layer value (v) related to its penetration in the voxelmap, and each point ($\mathbf{P}_i$) its inwards pointing normal vector ($\mathbf{n}_i$). (b) Single point force ($\mathbf{f}_i$) can be computed scaling the normal vector with its penetration with the penalty-based force computation approach. The cross products of forces and points yield torques. In our implementation, we instead provide the contact manifold to Bullet. (c) Representation of the contact manifold ($\{\mathbf{P}_i, \mathbf{n}_i, V(\mathbf{P}_i)\}$) provided to Bullet. The physics engine can then compute the motion that separates the objects from collision ($\mathbf{H}' \leftarrow \mathbf{H}$).

During online collision detection, likely colliding points (see Section 2.3) are checked for their voxel value v in the voxelmap. Those points $\mathbf{P}_i$ with $v(\mathbf{P}_i) \geq 0$ are in solid voxels; their normal vectors $\mathbf{n}_i(\mathbf{P}_i)$ are summed, after being weighted by their penetration in the voxelmap ($V(\mathbf{P}_i) \geq 0$), yielding the total collision force $\mathbf{f}_{\text{tot}}$ (see Figure 3). Torques $\mathbf{t}_i$ generated by colliding points are the cross product between point coordinates $\mathbf{P}_i$ and forces $\mathbf{f}_i$, all magnitudes expressed in the pointshell frame, with its origin in the center of mass. These torques $\mathbf{t}_i$ are then summed to compute the total torque $\mathbf{t}_{\text{tot}}$. This process is summarized in (2) and (3):

$$\mathbf{f}_i = V(\mathbf{P}_i)\mathbf{n}_i \rightarrow \mathbf{f}_{\text{tot}} = \sum_{\forall i | V(\mathbf{P}_i) \geq 0} \mathbf{f}_i, \tag{2}$$

$$\mathbf{t}_i = \mathbf{P}_i \times \mathbf{f}_i \rightarrow \mathbf{t}_{\text{tot}} = \sum_{\forall i | V(\mathbf{P}_i) \geq 0} \mathbf{t}_i. \tag{3}$$

The voxelmap distance or penetration function $V(\mathbf{P})$ has two components: global and local penetrations, as shown in (4):

$$V(\mathbf{P}_i) = \underbrace{\mathbf{n}_i \mathbf{e}_i}_{\text{local}} + \underbrace{v(\mathbf{P}_i)\sigma}_{\text{global}}. \tag{4}$$

The global penetration ($v(\mathbf{P})\sigma$) is the value of the voxel

in which the point lies multiplied either by the voxel size ($\sigma = s$) or by the scaling factor ($\sigma = \lambda$), both introduced in Section 2.1.1. In case $\sigma = \lambda$, the global penetration is the approximate distance/penetration $\tilde{d}$. On the other hand, in case $\sigma = s$, the global penetration indicates a coarser approximative depth of the point in the voxelmap. The local penetration ($\mathbf{n}_i\mathbf{e}_i$) is the projection of the vector between the pointshell point and the voxel center ($\mathbf{e}_i = \mathbf{C} - \mathbf{P}_i$) on the normal vector of the point; hence, it represents the depth of the point within the voxel. If the chosen resolution is high enough ($s \to 0$), the influence of the local penetration decreases.

Note that when $V(\mathbf{P}) \leq 0$, we are measuring distance, and $\max(V(\mathbf{P} = \mathbf{Q}) \leq 0)$ is the separation distance between the objects, being $\mathbf{Q}$ the pointshell point which is closest to the counterpart voxelmap object.

We can additionally refine the obtained penetration/distance value by linearly interpolating the floating point values [SH13]. For that, a support floating point distance field which stores the real distance from the voxel center ($V_R(\mathbf{C})$) and the distance gradient in that point (∇V_R) is required (see Section 2.1). The interpolated value is then:

$$V_R(\mathbf{P}) = \frac{1}{s}[\nabla V_R \cdot (\mathbf{P} - \mathbf{C})] + V_R(\mathbf{C}). \tag{5}$$

The number of pointshell points that has to be checked influences the collision computation time, whereas the voxelmap resolution (i.e., the voxel size) affects only the quality of the force magnitudes [WSM*10]. Ideally higher pointshell and voxelmap resolutions should be used only in likely colliding areas.

Forces and torques generated according to (2) and (3) need to be scaled due to the fact that their stiffness strongly depends on the number of points that collide, and therefore, on the pointshell resolution that is used; ideally a similar collision force should be applied on a peg-like object that collides with its end or its body on a plane.

2.3. Hierarchical Point Traverse

The real-time collision detection algorithm traverses the sphere-point tree and computes the contact manifold (M), the penalty forces ($\mathbf{f}$, $\mathbf{t}$), and the distance or penetration value (δ).

Algorithm 1 displays the hierarchical traverse; it is important at this point, though, to define more formally a cluster c according to Section 2.1.2:

$$c = \left(L, (R, \mathbf{X}), \{\mathbf{P}_k, \mathbf{n}_k\}, \{c_k^c\}, c^p\right), \tag{6}$$

where

- L is the level where the cluster is, being the highest level $L = 1$ and the level of the leaves $L = N_L$,
- $(R, \mathbf{X})$ are the radius R and the center $\mathbf{X}$ of the minimally bounding sphere that contains all children points,
- $\{\mathbf{P}_k, \mathbf{n}_k\}_{k=1}^K$ are the K cluster points, being $\mathbf{P}_1$ the cluster parent point,
- $\{c_k^c\}_{k=1}^K$ are the addresses of the K children clusters,
- and c^p is the address of the parent cluster.

Algorithm 1 is called once every cycle. As shown in there, at the beginning of each cycle, the uppermost cluster with the sphere that encloses all points is pushed to a FIFO-queue. Then, the clusters of the queue are iteratively popped in breadth-first manner. In case the popped cluster sphere is colliding, the parent point of the cluster is checked for collision and the children clusters are pushed to the queue. In the case we are processing a leaf cluster, we have to check all the points in the cluster, not only the parent point (lines 23-33 in Algorithm 1). Whenever we detect that a point is colliding, we update the forces as explained in Section 2.2 (lines 19-20 and 30-31 in Algorithm 1). Additionally, we add the colliding point information to the contact manifold (lines 18 and 29 in Algorithm 1). Note that although we can compute online the penalty forces, we provide Bullet only with geometrical contact information –i.e., the contact manifold M– and let this engine compute the corresponding forces and motion. It is important to remark that although we are able to provide more accurate contact information, Bullet accepts contact manifolds of size 4 by default (see Section 3.2).

Figure 4 displays the hierarchical traverse with an example. In that figure we can see primitives (spheres and points) colored in blue when they are checked for collision and in red when the collision check is positive. The three last levels are displayed and only one cluster is considered in that moment in the queue –the uppest one in the figure. Spheres and points are checked in a breadth-first manner starting from this uppest cluster and the algorithm converges to the down right part of the point cloud –where the collision actually occurs– refining the point resolution at each level.

2.4. Segmented Hierarchical Traverse

The segmented or clustered hierarchical traverse is directly related to the traverse explained in the previous Section 2.3. The motivation of the modification done here comes from the convex decomposition approach [MG09] implemented in Bullet. This approach segments the object in m convex hulls that preserve the shape of the object; then, the segmented parts can be checked for collision with the GJK algorithm [GJK88]. Each convex segment can deliver a contact manifold with at most one contact point.

We implemented a similar approach in order to compare our algorithm with the method based on convex decomposition within Bullet, given that it yields very good results. The user can offline select the value of m –we usually take

Algorithm 1: $(M, \mathbf{f}, \mathbf{t}, \delta)$ = `collisionQuery` (V, P)

Data: signed distance field $V \in \mathbb{R}^{1\times N_V}$, and point-sphere hierarchy $P = \{c\}_1^{N_C}$ transformed into distanced field coordinates.
Result: Contact manifold $M = \{(\mathbf{P}_j, \mathbf{n}_j, V(\mathbf{P}_j))\}_{j=1}^{N_M} \in \mathbb{R}^{7\times N_M}$, penalty forces $\mathbf{f} \in \mathbb{R}^3$ and torques $\mathbf{t} \in \mathbb{R}^3$, and distance/penetration value $\delta \in \mathbb{R}^1$.

```
1  //Penalty force, torque, and distance/penetration
2  f ← 0, t ← 0, δ ← 0
3  //Contact manifold
4  M ← ∅
5  //FIFO queue that contains clusters to visit
6  Q ← ∅
7  //Root cluster that bounds all pointshell points
8  c_1 ← P.getRootCluster()
9  //Initialize queue
10 Q.push(c_1)
11 while Q ≠ ∅ do
12     c ← Q.pop()
13     if V(c.X) ≤ c.R then
14         //Sphere is colliding
15         //Check parent point for collision
16         if unchecked(c.P_1) AND V(c.P_1) ≥ 0 then
17             //Cluster parent point colliding
18             M.add(c.P_1, c.n_1, V(c.P_1))
19             f ← f + V(c.P_1) · c.n_1
20             t ← t + c.P_1 × f
21             if V(c.P_1) > δ then
22                 δ ← V(c.P_1)
23         //Check if c is a leaf cluster
24         if c.L == N_L then
25             //Cluster c is leaf: check all other points
26             for k = 2 to K do
27                 if V(c.P_k) ≥ 0 then
28                     //Leaf point colliding
29                     M.add(c.P_k, c.n_k, V(c.P_k))
30                     f ← f + V(c.P_k) · c.n_k
31                     t ← t + c.P_k × f
32                     if V(c.P_k) > δ then
33                         δ ← V(c.P_k)
34         //Push children
35         Q.push(c.{&c_k^c}_{k=1}^K)
36     else
37         //Sphere is colliding
38         //Update distance
39         if V(c.X) − c.R > δ then
40             δ ← V(c.X) − c.R
41 return (M, f, t, δ)
```

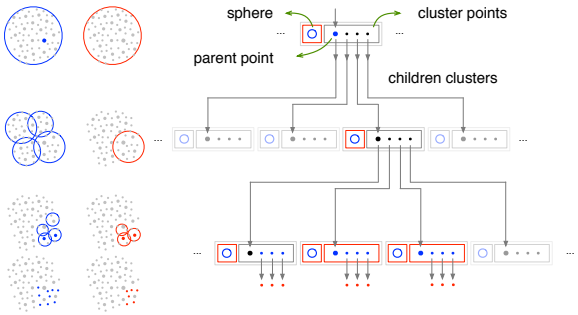


Figure 4: Breadth-first traverse of the point-sphere tree. This figure illustrates the Algorithm 1. Blue spheres and points represent checked primitives, while red primitives are the colliding ones. The algorithm converges to the collision area in the downer right part.

$m \sim 15$–, i.e., the approximate number of segments of the object, which corresponds to the number of contact manifolds to be generated and passed to Bullet. The algorithm then looks at the level that contains a similar number of clusters and sets it to be the reference segmentation level with m' clusters Note that in the segmented collision detection the size of the manifold is $N_M = m'$. Next, Algorithm 1 is modified as follows:

- Line 2: we have m' different penalty values $\mathbf{f}$, $\mathbf{t}$, and δ.
- Line 4: we have an array of m' different contact manifolds $\{M\}_1^{m'}$.
- Line 8: m' clusters corresponding to the segmentation level are popped.
- Lines 11-40: the queue is initialized m' times, once for each of the popped m' clusters and the whole while loop is executed for each of them.
- Lines 18 and 29: the contact point is added to the manifold only if $V(c.\mathbf{P}) \leq \delta$; since the new manifold supports only one point, the existing point is replaced every time a point is found in the segment that maximizes the penetration.

3. Integration into the Bullet Physics Engine

This section explains the integration of our algorithm into the physics engine Bullet. First, a brief overview of the engine's architecture is given in order to explain the basic workflow to which our algorithm had to be integrated. In the second subsection, our new interface structures are described.

3.1. Data Structures and Workflow in Bullet

The layer that controls the dynamics is above the collision detection layer in Bullet. It is possible to perform only contact computation, but we will consider the general case with rigid body dynamics.

The highest control interface that creates the virtual world is `btDiscreteDynamicsWorld`. We can `setGravity()` to it, call `addRigidBody()`, and simulate the next instant in the world with `stepSimulation()`. After each step, it is possible to ask the `btTransform` of each rigid body.

Each `btRigidBody` consists of a `btMotionState` and a `btCollisionShape`; this last class contains the geometry that is used for the collision detection.

The collision detection is divided in two phases in Bullet: (*i*) the broadphase, in which pairs of objects are quickly rejected based on the overlap between their axis aligned bounding boxes and (*ii*) the narrowphase, in which the selected collision detection algorithm is called. The interfaces `btBroadphaseInterface` and `btCollisionDispatcher` are used respectively for these two steps. The latter contains the selected `btCollisionAlgorithm`, which generates a `btPersistentManifold`, that is, the list of object points that conform the contact manifold and which is passed to the motion solver.

Therefore, the contact manifold is the ultimate result of the collision detection process, and the rigid body dynamics simulation can work decoupled from the type of algorithm used, provided the list of contact points. Hence, the goal of our integration is to generate such a manifold as fast as possible, and with the best quality as possible.

Refer to [Cou14] for more detailed information on the Bullet pipeline.

3.2. Integration Interfaces

We modified the following structures by adding references to our algorithm: `btBulletCollisionCommon.h`, `btBroadphaseProxy`, and `btDefaultCollisionConfiguration`. The appropriate algorithm is assigned to each shape type in the last interface.

As shown in Figure 5, we additionally inherited the abstract collision shape interface and extended it to represent the data structures introduced in Section 2. Since our algorithm requires two different data structures for each colliding pair, we defined a mixed structure containing both structures for each object. The structure which is used is selected automatically online: the object with less points will be the `Pointshell`. The inertia matrix and some other features required by Bullet, such as bounding volumes, are automatically created with our interfaces.

In the broadphase, bounding spheres of the highest pointshell hierarchy level are checked against the distance fields. Colliding pairs are handled by the dispatcher, which calls our collision detection algorithm explained in Section 2. The result of the query is a list of colliding points

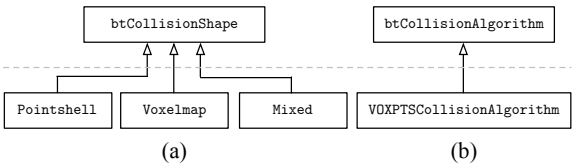


Figure 5: (a) Integration of the three new collision shapes into the pool of collision shapes provided by Bullet: `Pointshell`, `Voxelmap`, and `Mixed`, which contains the previous two. (b) Integration of the `VOXPTSCollisionAlgorithm` by inheriting from Bullet's superclass `btCollisionAlgorithm`.

with their respective normal vectors and penetration values ($\{(\mathbf{P},\mathbf{n}(\mathbf{P}),V(\mathbf{P}))_i\}$, see Figure 3). If cluster based collision detection is performed (see Section 2.4), this list is already segmented in m' clusters and for each one a `btPersistentManifold` is created with the point in the cluster that has the deepest penetration value ($\max(V(\mathbf{P}))$). In case the regular hierarchical traverse is carried out (see Section 2.3), a unique `btPersistentManifold` is filled with the contact points, starting with the deepest point, and adding points so that the manifold area maximizes. Note that the size of the manifold is limited to four points in Bullet, although this constant value can be modified before compilation.

Everything is implemented in C++, and we use CMake for the building process, as it is done by Bullet. Altogether, within the Bullet framework, three headers were modified and four added, three source files were added, and four CMake files were modified. The integration plug-in is installed with a simple script that makes a backup of the original files in the Bullet repository and copies the new files to their corresponding folders, allowing also uninstallation. Bullet must be recompiled after the installation of our plug-in.

4. Experiments and Results

In this section, we firstly show the results of simple experiments to prove the validity of our algorithm, which is compared to the default algorithms in Bullet. Afterwards, we compare our algorithm with the fastest collision detection algorithm in Bullet in more challenging scenarios. All tests were carried out using a PC running SUSE Linux Enterprise Edition 11 with an Intel Xeon CPU at 4x2.80 GHz.

4.1. Bouncing Ball

In this scenario we analyze the height of a sphere dropped onto a plane as well as its maximum penetration. The sphere with a radius of 0.5 m and a mass of 1 kg was dropped from a height of 1 m. The gravity was considered to be 10 m/s^2 and the frequency of the simulation was of 200 Hz. Given that

the experiment uses very simple objects, our algorithm will generate at most at each time step one colliding point. For this type of collision, Bullet calls a simple algorithm called `btSphereBoxCollisionAlgorithm`.

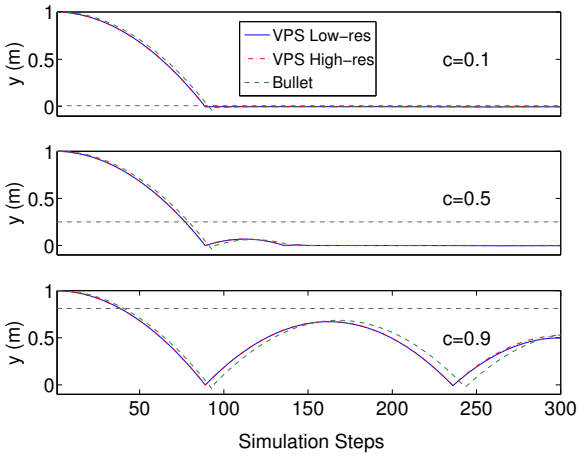


Figure 6: Height of the center of mass of the bouncing ball (radius 0.5 m) using different coefficients of restitution c. The pointshell of the sphere is composed of 275 (low resolution) and 25880 (high resolution) points. The plane's voxelmap has a resolution (voxel edge size) of 5 mm. The black dashed line represents the ideal maximum height after the first collision.

Table 1: Maximum penetration errors (mm) in the bouncing ball experiment using Bullet and our algorithm. Two resolutions are considered for our algorithm: coarse with 275 points and high with 25880 points.

Restitution coefficient [-]	**Penetration error** (mm)		
	Bullet	VPS low	VPS high
0.1	42.5	6.8	3.5
0.5	42.5	9.5	4.0
0.9	42.5	20.1	19.2

The results show that the height profile of the center of mass of the ball for our algorithm roughly matches the one yielded by using Bullet's algorithm. The discrepancies between our approach and Bullet's are due to Bullet having higher penetration errors (see Table 1), which delay the rebound and increase the period of the bouncing. Having pointshell objects with much higher resolutions seems to provide lower penetration errors, but the benefits are not substantial.

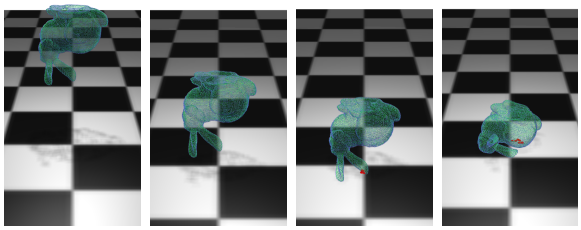
Alongside the bouncing ball experiment, we also tested stacking similar objects. Stacking spheres and disabling freezing of the objects will cause them to collapse, eventually. As we increase the resolution of the pointshell representation of the sphere the stack gets more and more stable. Using a coarser sampling grid to generate the pointshells intrinsically adds some quantization noise to the modeled object; that could explain its apparently more realistic behavior.

4.2. Stanford Bunny Tests

In this section we first compare our algorithm against other available algorithms in Bullet varying the resolution of the Stanford Bunny. Next, we focus on the fastest algorithm in Bullet –based in the convex decomposition.

4.2.1. General Comparison with Bullet Algorithms Varying Resolution

Figure 8 shows computation time (μs) and linear velocity (m/s) diagrams produced by our algorithm, the Bullet's convex decomposition, and the Bullet's GImpact, which is used with arbitrary non-convex triangle meshes. During the experiment a Stanford Bunny (35606 vertices) was dropped onto a horizontal plane as shown in Figure 7.



(a) (b) (c) (d)

Figure 7: Successive frames of a Stanford Bunny dropped onto a plane, shown at the attached video. This experiment corresponds to the Sections 4.2.1 and 4.2.3, and the Figures 8 and 10. In the case of Figure 8, the frames match with the following steps: (a) Step $\sim$ 50, (b) Step $\sim$ 175, (c) Step $\sim$ 200, and (d) Step $\sim$ 300.

During full operation (Figure 8, steps 250 to 600), our algorithm is 137$\times$ faster than GImpact and requires 0.71 ms for each check on average using the fine resolution (34892 points). The bunny is simplified to a convex hull composed of only 42 vertices in the Bullet's GJK implementation and to 8 convex hulls with 100 vertices each in the convex decomposition approach. With these conditions, GJK is 339$\times$ faster than our algorithm with a fine resolution (34892 points), but the convex decomposition is only 1.3$\times$ faster than our algorithm with a coarse resolution (799 points).

4.2.2. Segmented Collision Detection

In this section we validate the segmented collision detection method introduced in Section 2.4. Firstly, we will provide evidence supporting the fact that the speed of the collision detection is barely influenced when this method is applied.

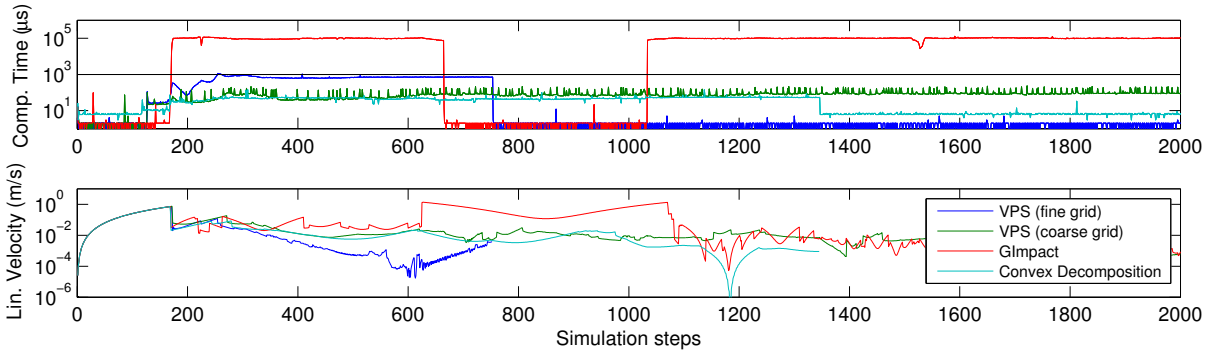


Figure 8: Computation time (μsec) and linear velocity (m/sec) curves in logarithmic scale for the testing scenario in which a Stanford Bunny with 35606 vertices is dropped onto a plane (see Figure 7). The pointshell of the bunny is composed of 799 (coarse) and 35596 (fine) points, and the voxelmap with $306 \times 305 \times 282$ voxels. The decomposed bunny consists of 8 convex hulls with 100 vertices each. Note that Bullet de-activates collision detection under certain kinetic conditions causing sudden steps in the computation time curves.

After that, we will show that with this method we can guarantee a more robust performance where low frequency simulations (<50 Hz) are concerned. The following experiments used a Stanford Bunny of low (799 points) and high resolutions (35596 points) with a mass of 1 kg and an inertia diagonal matrix $I = (4.79, 4.46, 6.38) \times 10^{-3}$ u, which was dropped from a height of 0.5 m onto a plane, as shown in Figure 7.

We measured the time taken to detect a collision for 10 random initial rotation matrices on the bunny and selected the average. This average was computed using the time interval between step 200 and 600 when the algorithms were detecting most collisions and were, thus, performing slower. The number of points chosen for each run are the number of clusters in a certain level L of the point-sphere tree, where $L = 1$ represents the top level containing only one cluster. Note that for $L > 1$, the number of segments are $m' = LK$, being K the number of children clusters; in our experiments, we used $K = 4$. The results in Table 2 show that the computation times are very similar throughout all levels.

Table 2: Computation time (μsec) of our collision detection algorithm when supplying different number of contact points according to the level of the point-sphere tree selected for the segmentation. Low resolution: 799 points; high resolution: 35596 points.

Res.	**Level**				
	$L=1$	$L=2$	$L=3$	$L=4$	$L=5$
Low	102.2	102.1	104.2	102.3	102.8
High	1387.9	1477.5	1407.4	1449.6	1494.3

We performed the same experiment at frequencies lower than 50 Hz using the Stanford Bunny and observed the penetration errors. When performing the test for $L = 1$ (one colliding point) the algorithm was able to maintain the bunny above the plane until we reached $\sim$ 30 Hz. Operating at a lower frequency would lead to the bunny falling through the plane after the initial collision was detected. The same experiment was done using $L = 4$ and $L = 5$ and the bunny was maintained above the plane for frequencies of simulation down to $\sim$ 10 Hz, where the penetration errors were minimized if a higher level L was chosen.

4.2.3. Comparison with the Convex Decomposition

According to our experience, the fastest collision method in Bullet for complex objects involves using the convex decomposition method that segments the object into convex patches [MG09] that can be independently checked for collision using the GJK algorithm [GJK88]. In Figure 9, a comparison between the segmentation done by the convex decomposition method and our clustering is presented. In this section, we will compare the performance of our method against the convex decomposition by analyzing the speed of the algorithms, the physical behavior of the objects and the accuracy of the colliding object used by both algorithms. The experiments were conducted in the same conditions as in the previous section.

Figure 10 shows the computation time (μsec) and the kinetic energy (J) of the bunny at each time step when using our algorithm and Bullet's convex decomposition. During full operation time (between time steps 200 and 600), the convex decomposition method and our algorithm have an average computation time of $\sim 30\,\mu$sec and $\sim 100\,\mu$sec (levels $L = 1, 4$, $K = 4$ number of children). Although convex decomposition is little more than three times faster, our method provides much more realistic physical behavior probably

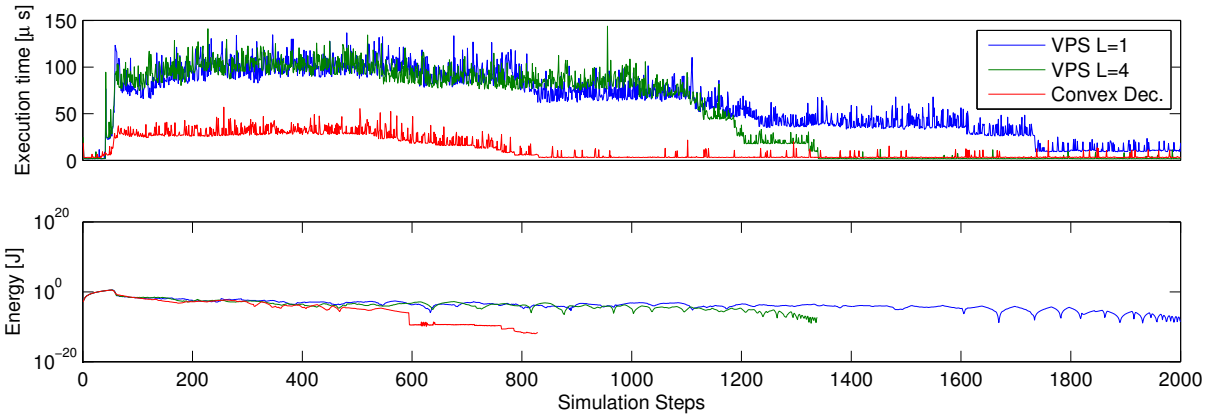
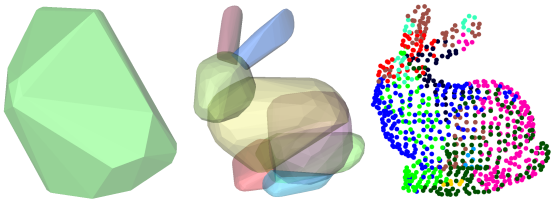


Figure 10: Computation time (μsec) and kinetic energy (J) curves in linear and logarithmic scale, respectively, for the testing scenario in which a Stanford Bunny with 35606 vertices is dropped onto a plane (see Section 4.2.3 and Figure 7). The pointshell of the bunny is composed of 799 points (coarse). The decomposed bunny consists of 8 convex hulls with 100 vertices each. Note that Bullet de-activates collision detection under certain kinetic conditions causing sudden steps in the computation time curves.



(a) (b) (c)

Figure 9: Different representations of the Stanford Bunny used in the tests of Section 4.2.3: (a) Convex hull created with Bullet; (b) Convex decomposition created by Bullet [MG09]; (c) Our segmented point representation encoded by colors.

due to the fact that our sampling of the point cloud of the object does not involve any approximation, but only a change in the resolution. We observe that convex decomposition has a worse visual performance because, for instance, when the object rolls through the plane after the first collision, one can see when the bunny transitions from being supported in one facet of one of its convex segments to another facet. This "discrete" transition does not occur in our algorithm, where the rolling is much smoother.

5. Conclusions and Future Work

We presented the integration and evaluation of a haptic rendering algorithm in the physics engine Bullet. Our algorithm is based on the Voxelmap-Pointshell Algorithm, since it also uses voxelmaps and pointshells. However, the improvements we made on these data structures allow us for faster and more accurate distance, penetration and penalty force computation.

The integration into the physics engine Bullet was performed inheriting interface classes provided in that framework and we plan to make public the source code of our plug-in.

Our experimental results show that, while achieving a higher accuracy, this first integration of our presented haptic rendering algorithm presents similar computation times as the tested ones with low resolutions, whereas it outperforms them when the resolution is increased. Therefore, we expect that our contributions could improve multibody contact and movement computations, which are used in a wide range of applications, such as gaming, robotics, or virtual prototyping.

Future work will address, among other topics, modifying the Bullet force constraint solver and extending a time-critical approach of our algorithm to work with the Bullet framework. Furthermore, we plan to automatically generate our data structures within Bullet given a mesh, since currently these have to be created outside the physics engine.

References

[AM05] AKENINE-MÖLLER T.: Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses* (2005), ACM, p. 8. 3

[Bar92] BARAFF D.: *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, 1992. 2

[BB07] BOEING A., BRÄUNL T.: Evaluation of real-time physics simulation systems. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques (ACM)* (2007). 2

[BETC12] BENDER J., ERLEBEN K., TRINKLE J., COUMANS

E.: Interactive simulation of rigid body dynamics in computer graphics. In *EuroGraphics* (2012). 2

[BJ08] BARBIČ J., JAMES D. L.: Six-dof haptic rendering of contact between geometrically complex reduced deformable models. *Haptics, IEEE Transactions on 1*, 1 (2008), pp.39–52. 2, 3

[BS02] BASDOGAN C., SRINIVASAN M. A.: Haptic rendering in virtual environments. *Handbook of virtual environments* (2002), pp.117–134. 1

[CLMP95] COHEN J. D., LIN M. C., MANOCHA D., PONAMGI M.: I-collide: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference* (1995). 2

[Cou05] COUMANS E.: *Continuous Collision Detection and Physics*. Tech. rep., Sony Computer Entertainment, 2005. 2

[Cou14] COUMANS E.: Bullet physics library 2.82, 2014. Web page accessed on August 29th, 2014. URL: http://bulletphysics.org/wordpress/. 1, 8

[CSC05] CONSTANTINESCU D., SALCUDEAN S., CROFT E.: Haptic rendering of rigid contacts using impulsive and penalty forces. *Robotics, IEEE Transactions on 21*, 3 (June 2005), pp.309 – 323. 2

[Erl05] ERLEBEN K.: *Stable, Robust, and Versatile Multibody Dynamics Animation*. PhD thesis, University of Copenhagen, 2005. 2

[FG04] FISCHER K., GÄRTNER B.: The smallest enclosing ball of balls: combinatorial structure and algorithms. *International Journal of Computational Geometry & Applications 14*, 04n05 (2004), pp.341–378. 3, 5

[GJK88] GILBERT E. G., JOHNSON D. W., KEERTHI S. S.: A fast procedure for computing the distance between complex objects in three-dimensional space. *Robotics and Automation, IEEE Journal of 4*, 2 (1988), pp.193–203. 2, 6, 10

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of ACM SIGGRAPH '96* (1996). 2

[HWS*12] HUMMEL J., WOLFF R., STEIN T., GERNDT A., KUHLEN T.: An evaluation of open source physics engines for use in virtual reality assembly simulations. In *International Symposium on Visual Computing* (2012). 2

[KS86] KAUFMAN A., SHIMONY E.: 3d scan-conversion algorithms for voxel-based graphics. In *Proceedings of the 1986 workshop on Interactive 3D graphics* (1986). 3

[LC91] LIN M. C., CANNY J. F.: A fast algorithm for incremental distance calculation. In *In IEEE International Conference on Robotics and Automation* (1991). 2

[Leo07] LEON F.: Gimpact - geometric tools for vr, 2007. Web page accessed on August 29th, 2014. URL: http://gimpact.sourceforge.net/. 2

[LOLO08] LIN M. C., OTADUY M., LIN M. C., OTADUY M.: *Haptic rendering: Foundations, algorithms and applications*. AK Peters, Ltd., 2008. 2

[MC94] MIRTICH B., CANNY J.: Impulse-based dynamic simulation. In *Proceedings of Workshop on Algorithmic Foundations of Robotics* (1994). 2

[MG09] MAMOU K., GHORBEL F.: A simple and efficient approach for 3d mesh approximate convex decomposition. In *IEEE International Conference on Image Processing (ICIP)* (2009), pp. 3501–3504. 2, 6, 10, 11

[MPT99] MCNEELY W. A., PUTERBAUGH K. D., TROY J. J.: Six degree-of-freedom haptic rendering using voxel sampling. In *Proc. of ACM SIGGRAPH* (1999). 1, 2, 3, 5

[MPT06] MCNEELY W. A., PUTERBAUGH K. D., TROY J. J.: Voxel-based 6-dof haptic rendering improvements. *Haptics-e 3*, 7 (2006), pp.1–12. 2, 3

[OL03] OTADUY M. A., LIN M. C.: Sensation preserving simplification for haptic rendering. In *Proceedings of ACM SIGGRAPH 2003 / ACM Transactions on Graphics* (2003). 2

[ORC07] ORTEGA M., REDON S., COQUILLART S.: A six degree-of-freedom god-object method for haptic display of rigid bodies with surface properties. *Visualization and Computer Graphics, IEEE Transactions on 13*, 3 (2007), pp.458–469. 2

[RC13] RYDÉN F., CHIZECK H. J.: A method for constraint-based six degree-of-freedom haptic interaction with streaming point clouds. In *ICRA* (2013). 2, 5

[RKC02] REDON S., KHEDDAR A., COQUILLART S.: Fast continuous collision detection between rigid bodies. In *EUROGRAPHICS 2002* (2002). 2

[SH13] SAGARDIA M., HULIN T.: Fast and accurate distance, penetration, and collision queries using point-sphere trees and distance fields. In *SIGGRAPH* (2013). 6

[SHPH08] SAGARDIA M., HULIN T., PREUSCHE C., HIRZINGER G.: Improvements of the voxmap-pointshell algorithm - fast generation of haptic data-structures. In *53. IWK - TU Ilmenau* (2008). 3, 4, 5

[SR06] SEUGLING A., RÖLIN M.: *Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool*. Master's thesis, Umea University, 2006. 2

[SWH*12] SAGARDIA M., WEBER B., HULIN T., PREUSCHE C., HIRZINGER G.: Evaluation of visual and force feedback in virtual assembly verifications. In *IEEE Virtual Reality* (2012). 1

[vdB01] VAN DEN BERGEN G.: Proximity queries and penetration depth computation on 3d game objects. In *Game Developers Conference 2001* (2001). 2

[WSM*10] WELLER R., SAGARDIA M., MAINZER D., HULIN T., ZACHMANN G., PREUSCHE C.: A benchmarking suite for 6-dof real time collision response algorithms. In *ACM Virtual Reality and Software Technology* (2010). 6

[WZ09] WELLER R., ZACHMANN G.: A unified approach for physically-based simulations and haptic rendering. In *ACM SIGGRAPH Video Game Proceedings* (2009). 2, 5