Master's Thesis

Improvement of a Multi-Body Collision Computation Framework and Its Application to Robot (Self-)Collision Avoidance

Author
Alexander Martín Turrillas

German Aerospace Center (DLR)
Institute of Robotics and Mechatronics
Oberpfaffenhofen, June 1, 2015

Supervisor DLR
Mikel Sagardia

Supervisor Tecnun
Emilio Sánchez

Abstract

One of the fundamental demands on robotic systems is a safe interaction with their environment. In order to fulfill that condition, both collisions with obstacles and own structure have to be avoided. This problem has been addressed before at the German Aerospace Center (DLR) through the use of different algorithms. In this work, a novel solution that differentiates itself from previous implementations due to its geometry-independent, flexible thread structure and computationally robust nature is presented.

In a first step, in order to achieve self-collision avoidance, collision detection must be handled. In this line, the Robotics and Mechatronics Center of the DLR developed its own version of the Voxmap-Pointshell (VPS) Algorithm. This penalty based collision computation algorithm uses two types of haptic data structures for each pair of potentially colliding objects in order to detect contact points and compute forces of interfering virtual objects; voxelmaps and pointshells.

Prior to the work presented, a framework for multi-body collision detection already existed. However, it was not designed nor optimized to handle mechanisms. This thesis presents a framework that handles collision detection, force computation and physics processing of multi-body virtual realities in real-time integrating the DLR VPS Algorithm implementation.

Due to the high number of available robots and mechanisms, a method that is both robust and generic enough to withstand the forthcoming developments would be desirable. In this work, an input configuration file detailing the mechanism's structure is used, based on the Denavit-Hartenberg convention, so that any type of robotic system or virtual object can use this method without any loss of validity.

Experiments to prove the validity of this work have been performed both on DLR's HUG simulator and on DLR's HUG haptic device, composed of two DLR-KUKA light weight robots (LWRs).

Declaration

This thesis is an account of research undertaken between November 2014 and April 2015 at the Institute of Robotics and Mechatronics of the German Aerospace Center (DLR), Oberpfaffenhofen, Germany.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Alexander Martín Turrillas
Oberpfaffenhofen, April 2015

Master's Thesis Description

Deutsches Zentrum für Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft

Improvement of a Multi-Object Collision Computation Framework and Its Application to Humanoid and Industrial Robots for Collision Avoidance

Predicting and preventing collisions with the environment and with itself is essential for humanoid and industrial robots in order to guarantee safe manipulations. In this line, the Robotics and Mechatronics Center of the German Aerospace Center (DLR) developed its own version of the Voxmap-Pointshell (VPS) Algorithm [1, 2] in order to detect collisions and compute forces of interfering virtual objects. This penalty based collision computation algorithm uses two types of haptic data structures in order to achieve update rates of 1 kHz: voxelmaps and pointshells.

As shown in Fig. 1, voxelmaps are 3D grids in which each voxel stores a discrete distance value $v \in Z$ to the surface. On the other hand, pointshells are sets of points uniformly distributed on the surface of the object; each point has additionally an inwards pointing normal vector. Both structures can be built using hierarchies in order to speed up the collision detection process. During collision detection, distances and penetrations can be computed between these data structures, as explained in Fig. 2.

In order to perform collision detection for humanoids or similar complex mechanisms, engines that support the simultaneous computation of multiple bodies are necessary [3], which could increase the complexity of the problem quadratically. The DLR is currently working on a framework that handles collision detection, force computation and physics processing of muti-body virtual realities in real-time.

The objective of this work consists in supporting the research on muti-body collision detection applied to collision avoidance for humanoid and industrial robots. During the thesis, the humanoid robot SpaceJustin (see Fig. 3) will be used as a test platform [4]. Concrete goals will be established according to the student's profile and needs.

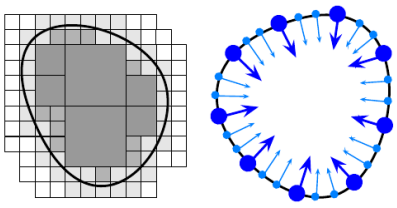


Figure 1. *Left: 2D representation of a voxelmap; deeper voxels are darker. Right: 2D representation of a pointshell; darker and bigger points represent higher positions in the hierarchy.*

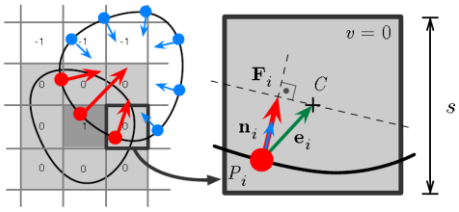


Figure 2. *On the left, a simplified voxelmap (bottom) is colliding with red pointshell points (top), yielding red bold collision forces. On the right, the computation of a single collision force related to a colliding point is graphically shown. Single collision forces are computed scaling the normal vector ($\mathbf{n}_i$) of the colliding point (P_i) with the sum of the local ($\mathbf{n}_i\mathbf{e}_i$) and global (vs) penetration of the point in the object. Torques ($\mathbf{T}_i$) are the cross product between forces ($\mathbf{F}_i$) and point coordinates (P_i).*

Figure 3. *Humanoid robot SpaceJusin. The collision computation and avoidance system should be tested in this robotic system composed of 47 degrees of freedom.*

Qualifications

- Good knowledge of C/C++.
- Knowledge of computational geometry.
- Knowledge of physical phenomena between rigid solids.
- Knowledge of inter-process communications.
- Good English level; German is not necessary, but preferable.
- Optional: Linux, LaTeX, Matlab / Simulink.

Fields of Study

Computer Science, Mechanical Engineering, or similar.

Literature

[1] McNeely, W. A.; Puterbaugh, K. D.; Troy, J.J.: "Voxel-Based 6-DoF Haptic Rendering Improvements". In: Haptics-e. Vol. 3, No.7, 2006.

[2] Sagardia, M.; Stouraitis, T,: Lopes e Silva, J.: "A New Fast and Robust Collision Detection and Force Computation Algorithm Applied to the Physics Engine Bullet: Method, Integration, and Evaluation," in EuroVR, 2014, (to be published).

[3] 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.

[4] Täubig, H.; Bäuml, B.; Frese, U.: "Real-time Swept Volume and Distance Computation for Self Collision Detection," in IEEE/RSJ International Conference on Intelligent Robots and Systems, 2011.

Contact

Mikel Sagardia
Mikel.Sagardia@dlr.de
+49 8153 28 1039

German Aerospace Center (DLR)
Institute of Robotics and Mechatronics
D-82230 Oberpfaffenhofen, Germany

Acknowledgements

First and foremost I offer my sincerest gratitude to my supervisor, Mikel Sagardia, who has provided me the chance to be a part of the prestigious German Aerospace Center (DLR) for almost a year and supported me throughout my thesis with his patience and knowledge, whilst allowing me the room to work in my own way. I attribute the level of my Master's degree to his encouragement, effort and engagement through the learning process. Without him, this thesis, would not have been completed or written. One simply could not wish for a better or friendlier supervisor.

I am using this opportunity to express my gratitude to everyone who supported me throughout the course of this project. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the production of this work. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the project.

I place on record, my sincere thank you to Emilio Sánchez for advising me throughout my thesis, and most importantly, for awakening in me the curiosity to explore the world of robotics and providing me with the skills to successfully tackle this new challenge.

Beyond physics, programming and robotics, it is lucky for me to have met such valuable friends who inspirit my effort to overcome all the difficulties that arise from work and everyday life. I have to acknowledge all my colleagues at DLR (Maialen, Marti, Bastian, Henning, and Javi among many others) for their assistance in so many aspects that I cannot list due to the lack of space.

Special thanks to my co-worker Nora Etxezarreta, who has colored my stay in Munich ever since the first day. Always intellectually stimulating, she has the ability to turn a meal or a mundane train trip into an enlivening discussion scenario helping me notice life from a different perspective. Sometimes, one is carried away by the routine and fails to realize all what is being missed by walking blinded in life . Nora,

however, has constantly shared with me new plans, projects and ideas that have made my stay in Munich everything but dull or boring.

I am deeply grateful to Maria Muñoz, for making of my life an adventure and broaden my horizons, helping me realize that life is only as exciting as we are willing to make it. Without all her encouragement and reinforcement to take on new challenges I would have never obtained the opportunity to complete this project.

Last but not least, I owe more than thanks to my family members, which includes my parents, my sister, my grandmother and my godparents, for their financial support and encouragement throughout my life. Your faith in me was what sustained me thus far.

I want to dedicate this work to the memory of my grandfather Casildo Turrillas. He was my mayor inspiration to become an engineer. His wisdom, generosity and insightful nature made of him an exceptional role model to whom I owe most of what I know today. You will be missed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This introductory chapter starts by presenting the motivation for this research work. Following the motivation section, the main objectives of the thesis are detailed. The next section focuses on specifying the resources being used to carry the research and testing of this project. Finally, an outline of the Master's Thesis' structure is given.

1.1 Motivation

For robots in human and industrial environment, it is crucial to ensure a high level of safety. That involves the avoidance of collisions with both obstacles and the own structure. Over the last decades, this safety aspect has been addressed frequently. A wide array of solutions have been presented to cope with this problem. Many of them focus on presenting collision detection methods [19], [12], [14], while others provide solutions to reactive path-planning of safe trajectories and repulsion from potential obstacles [30], [21], [11]. There are even approaches that implement collision detection and avoidance methods on specific humanoid robots, such as *HRP-2* [31], *ASIMO* [32] or *Justin* [3].

However, to the date of the project, none of them provide enough simplicity to be used on any type of complex and non-convex geometry. Furthermore, many of the methods avaiable today handle collision detection and avoidance unitedly [2], [6], [30]. Greater flexibility is granted when collision detection is separated from the path-planning module, allowing the user to decide how to handle the latter feature. Even if a framework flexible enough to be used with any type of dynamic model is desired, to the date one of the most common applications for collision detection is the field of humanoid robots [31], [32], [3]. For this reason, a module that computes the forward kinematics of the robot given the joint angles/displacements is required.

Another important requirement for this work is the need of a method that can reliably peform at 1 kHz. Some of these methods achieve computational times under 1 ms [19], [6] while others fail to acomplish this rate [2]. In general, the computational time depends on the number and complexity of the objects. However, the VPS algorithm [28] introduced in Section 2.2 enables geometry-independent collision dectection. Due to this fact, a new solution that takes advantage of this algorithm not available until the date is desired.

1.2 Objectives

The main purpose of this work is to develop a framework capable of detecting and dealing with self collisions of multiple objects in real-time. It is also important for it to be generic enough so that any type of mechanism is eligible to make use of it. In order to achieve this goal, several subtasks have been specified:

- Literature research on self-collision detection and avoidance for humanoid and industrial robots.
- Development of the multi-body collision detection framework based on the VPS algorithm [28] and the previously implemented multi-body framework for assembly simulations. The work must satisfy the following requisites:
  - Be valid for any type of geometries.
  - Be valid for any type of mechanisms.
  - Operate robustly in real-time settings, that is, in 1 kHz.
- Development of arbitrary mechanism description file.
- Experiments and results.
- Documentation of complete research.

1.3 Resources

The following computer configurations and software versions have been used to produce this work. Reproducing the work under this same environment should yield similar results. Other configurations and software versions may also be used but the results could vary.

- The computer used to benchmark the Multibody Collision Computation algorithm used Intel Xenon 5060 at 3.2 GHz. The operating system was SUSE Linux Enterprise Desktop 11.

- To create and modify virtual models, Blender has been used in its 2.49 version.
- The VPS framework [28] has been used to compute contacts.
- The trafo3d framework has been used to handle algebraic operations in a simple and robust way.

1.4 Report outline

The report has been divided in the following chapters:

Chapter 2: Related Work The documentation studied is presented and the methods for collision detection are side-by-side compared.

Chapter 3: Generic Mechanism Model The process of building a configuration file for the mechanism to compute the fordward kinematics and joint forces and torques is described.

Chapter 4: Collision Computation between Mechanisms The framework that enables to compute contacts between any number of objects is detailed.

Chapter 5: Experiments and Results The results yielded by all the trials that have been performed is presented.

Chapter 6: Conclusions and Future Work The final output of this work is summarized and conclusions for the overall result are given. The future goals for the framework to accomplish are suggested.

Chapter 2

Related Work

In order for humanoid robots to become practical they must be able to operate safely and reliably. Self-collisions occur when one or more of the links of a robot collide. These collisions can result in damage to the robot itself, or through a loss of balance or control cause human injury or damage to its surrounding environment. Thus, detecting and avoiding self-collisions is fundamental to the development of robots which can be safely operated in human environments.

To address this problem several different approaches have been presented. To keep the report's length to a reasonable amount, only the most significant approaches related to this thesis are reviewed.

Some of the methods rely on strictly convex geometries to guarantee gradient continuity during the process of distance computation, while others do not give that much emphasis to geometry and focus on the use of repulsion potential fields applicable to both torque and position controlled manipulators. However, most of the approaches do make use of simplified convex geometries. This thesis presents an approach that tackles arbitrary complex geometries.

2.1 Self-Collision Detection for Humanoids

This thesis is influenced by four previous works that are reviewed in the following sections: 2.1.1, 2.1.2, 2.1.3, 2.1.4.

2.1.1 Sphere based Geometry Models for Self-Collision Avoidance

Sphere based geometric models can be used for the human and robot due to the efficiency of the distance computation. For this work the [2] sphere based collision avoidance has been studied, which builds upon [18], [22], [21]. In [2] the method is tested for a human and a *Puma 762* robot arm interacting in a robotic workcell.

The first step, was to model both the human and the robot arm using minimally bounding spheres. Figure 2.1 shows the result of modelling the two bodies using spheres; the human was built using 46 spheres while the *Puma 762* robot arm used 41. More detailed models may have been produced increasing the number of spheres for each object.

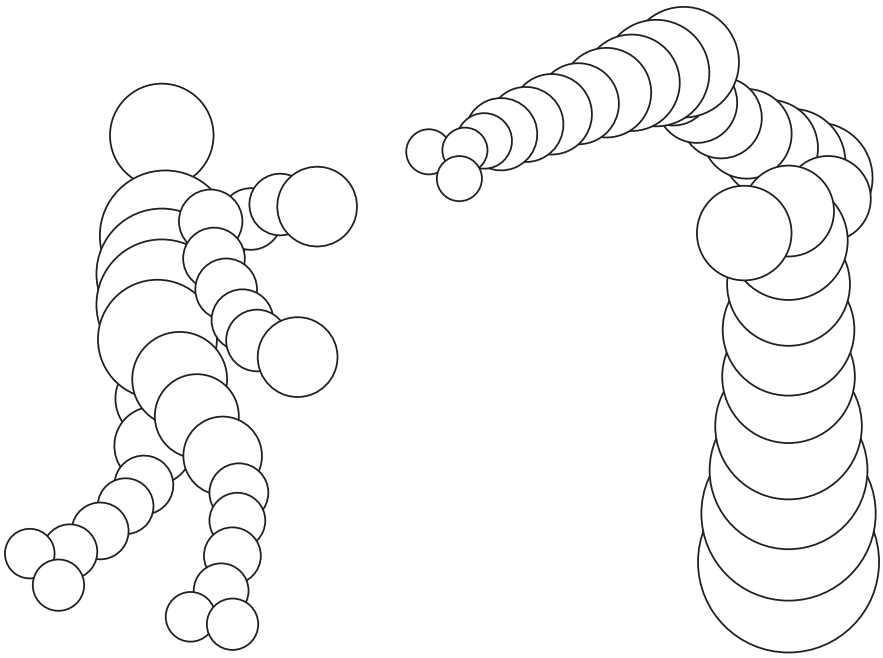


Figure 2.1: Virtual representation of sphere robot and human geometries.

The motion of both human and robot have to be tracked at all times. Whenever the robot links or the human moves, the centers of the spheres are updated with new values. Having updated the transformation matrices of the sphere centers, the main algorithm performs human-robot collision checks.To produce more accurate and consequently safer results, predictions of the robot and human motions are utilized to mitigate the detrimental effect of a non-instantaneous robot operating in a dynamic environment. Lucian Balan and Gary M. Bone[2] present a prediction method based on weighted average velocities applied at the sphere level of the human geometric

model. Averaging is used to reduce the effect of random velocity variations. By choosing to apply the prediction method at the sphere level, each sphere position is predicted independently. Therefore, limbs moving in different directions can be tracked with no effort. Lucian Balan and Gary M. Bone [2] report that it even works when there are several humans moving in the same workcell.

Once the collision detection has been performed the most suitable path to avoid potential contacts has to be selected. An optimization method employing penalty factors is used to determine the path that best balances between motion towards the goal and maximizing the distances between the spheres of the human and the robot. The relative position between two distinct human-robot sphere describe a search direction. For each search direction a cost function is computed; the closer the pair of spheres, the larger the penalty factor assigned to that search direction. The direction of motion with the smallest cost function is selected as the best choice for the robot motion command for the next time-step of the simulation. If all the search directions are predicted to produce collisions at some point along the time-step window, the robot is commanded to stop until the human drifts from the path of the robot. This collision avoidance approach is heuristic; it aims for efficiency rather than optimality of the solution.

The main advantages of this method are simplicity and efficiency. Computing the minimal distance between two spheres only requires a few basic computational operations, whereas for polyhedral models the computation is far more complicated. Due to the its efficiency and deterministic nature, it is a method suited to be used in real-time. However, [2] achieved a 40 Hz sampling rate on a 1.8 GHz Pentium IV PC for a total of 87 spheres, below the standards of some other collision detection methods presented in this chapter.

2.1.2 Reactive Self-Collision Avoidance

A wide field of research focuses on reactive repulsion potential field-based designs, introduced by Khatib [17]. This method has been implemented on the humanoid *ASIMO* [32]. Virtual repulsive forces are generated between potentially colliding links and transformed via an admittance into corresponding joint motions. Alexander Dietrich, Thomas Wimböck, Holger Täubigy, Alin Albu-Schäffer, and Gerd Hirzinger [6] present extensions to reactive self-collision avoidance for torque and position controlled humanoids continuing the work started in [29]. One of the main contributions presented is an efficient damping design which incorporates the configuration dependence of collision-endangered situations. They also present a strategy for indispensable emergency stops of the entire system. Finally, an admittance-based interface to

position controlled subsystems, which can be embedded in the derived force/torque based design approach, is provided.

In a first step, the distances between the virtually represented links have to be computed. For this purpose, a standard distance computation technique for convex hulls [12] is employed. Each rigid link is modelled by a fixed volume V. In every control cycle, all the volumes are transformed int the world frame by applying the corresponding transformations. Then, for every pair of links, the distance and corresponding proximal points are determined, being possible to exclude some pairs of points from being processed. Afterwards, the pairs of smallest distance are kept for further processing.

In the work presented by [6] the humanoid *Justin* [3] was modelled using only 78 points and 28 radii for *Justin*'s 28 links, achieving a tight representation of the humanoid robot. For real-time self-collision avoidance applications, *Justin*'s collision model is calculated once per control cycle applying 302 pairs of links.The computation time varies from 0.3 to 0.4 ms on an Intel Core2Duo Processor T7400 (2.16 GHz).

The basic idea behind the control algorithm is to apply repulsion potential fields to approaching links in order to avoid self-collisions while taking energy out of the system by dissipating kinetic energy via an efficient damping design. Furthermore, another additional safety feature is implemented: even in case collisions are improbable, if the repulsion fields are active, an emergency stop which leads to a mechanical braking of each motor was implemented.

2.1.3 Swept Volume Distance Computation

The task of collision detection and distance computation prior to the collision avoidance strategies can be performed using very distinct methods that may vary in efficiency and accuracy, being more suitable for one or another application. [33] introduce an algorithm based on computing the swept volumes of all bodies and checking them pairwise for collisions. It operates on joint angle intervals. Therefore, it does not only test a single or N intermediate configurations but assures safety of a whole movement. The key idea of the swept volume computation is representing volumes as convex hulls extended by a buffer radius producing the sphere swept convex hulls. This leads to tight and compact bounding volumes. The operation set available to model the different joints is strictly conservative and allows for a trade-off between accuracy and computation time. During a configurable timespan the algorithm updates a table of pairwise distances, and thus can guarantee real-time.

This method handles large braking distances, safeguards the whole braking movement, and executes in real-time. It first computes the swept volumes of all bodies, i.e., the volume the body touches within its movement. All volumes are represented in terms of sphere swept convex hulls of a finite set of points. So each volume is the Minkowski-Sum of a convex polyhedron given by a set of points, and a ball of radius r. For the case of the humanoid robot *Justin*, 26 bodies containing 80 points and 26 radii were used for its modelization, producing a fairly tight representation. A single body's representation uses only 3.1 points on average, less that 4 for a tetrahedron, the simplest polyhedral volume. Non-convex bodies have to be split into convex subparts which are treated as separate bodies.

Afterwards, all pairs of swept volumes are checked for collision, computing distances between two convex polyhedra given as arrays of points through the GJK-algorithm [12]. The algorithm requires a kinematic model of the robot defining joint-frames and a geometrical model with the robot's rigid bodies represented in one joint-frame. Testing an application with a fast moving humanoid robot along with the already described self-collision detection method, contacts were properly avoided according to [33]. 0.4ms computation time was achieved on an Intel T2500 at 2 GHz.

2.1.4 Protective Hulls for Collision Detection

This method is a variant of the sphere bounding algorithm for geometries explained in Section 2.1.1. Protective hulls, however, allow a tighter representation of the geometry while still permitting the inclusion of a safety margin. [19] describe an efficient geometric approach for detecting link interference, suitable for complex articulated robots such as humanoids, relying on fast, feature-based minimum distance determination methods for convex polyhedra [24]. Threshold values can be set on the allowable minimum distance between links in order to provide a safety margin that accounts for errors in modelling and control.

In the case of serial-chain manipulators, immediately adjacent links cannot collide if proper joint angle limits are defined. Each chain, in turn, must avoid collisions with all the other chains present in the mechanism. Given the number of links N the number of pairs P that should be checked for collision is given by:

$$P = \frac{N^2-3N+2}{2}. \tag{2.1}$$

In order to represent the link geometry for interference detection, approximate convex protective hulls of each link are derived from the original CAD models, which represent inherently closed surface models of solid objects. The hulls completely enclose the underlying geometry, and provide a safety margin around each link. Should a link have severe non-convex geometrical cal features, it can always be subdivided into a rigid collection of convex pieces. [19] have achieved a representation of the *H7* humanoid robot (30 DOF, 137 cm, 55 kg) using 2702 triangles, compared to the initial 314588 triangles from the CAD model.

For the minimum distance determination, the Voronoi-clip (*V-clip*) algorithm was selected [24]. This is a feature-based algorithm which improves upon the Lin-Canny algorithm [20]. For convex polyhedra, *V-clip* does not need to construct hierarchies of bounding volumes, like other methods [26], [13]. The running time does not depend on the distance between objects; only on their geometric complexity and motion relative to the previous query. The closest points between each active pair of links obtained through the algorithm are tracked over the course of an entire trajectory and verified for collision avoidance.

According to the work presented by [19], for a single posture of the robot (7 joint angles for each leg), 19 closest-feature pairs can be updated in less than 0.13 ms on average, including the forward kinematics calculation. All the minimum distances between all possible relevant body pais (435 pairs) can be calculated in approximately 2.5 ms.

2.1.5 Comparison between Methods for Self-Collision Detection for Humanoids

Collisions between bodies can be thought of as a binary result in its simplest form, i.e., whether two or more bodies overlap or not. This implies that checking for collision between objects following continuous motion trajectory necessitates either computing the swept volume of the object motions and checking for interference, or discretizing the trajectory into a finite set of samples which are individually tested for collision. Since swept volume calculations are overall difficult and expensive to compute, discretization is frequently used [2], [6], [19]. Nevertheless, regardless of the discretization resolution selected, it is always possible to construct a case in which a potentially dangerous collision goes undetected due to an insufficient number of samples. Due to this fact, when studying the trajectory sampling alone, methods that employ swept volumes would be more desirable.

Another important feature of the collision detection methods is the geometry

	geometry representation	trajectory sampling	coll. det. algorithm	f. kinematics module	force computation
sphere based CA [2]:					
	spheres	disc.	cost function based	no	no
reactive CA [6]:					
	convex hulls	disc.	GJK [12]	no	reactive pot. field
swept volume CA [33]:					
	convex hulls	cont.	adapted GJK [33]	no	no
protective hulls for CD [19]:					
	convex hulls	disc.	V-clip [24]	no	no

Table 2.1: Comparison between the methods presented in Sections 2.1.1, 2.1.2, 2.1.3, 2.1.4.

representation. Tighter representation (more accurate to the original geometry) produces more accurate results when checking for collisions. Some methods cannot handle any type of geometries and rely on methods that produce approximate convex reproductions of the original object to perform the collision computation [2], [6], [33], [19]. Overall, all these representations are conservative and always completely bound the original body. Moreover, the inclusion of a safety margin further mitigates the need for perfectly accurate modelization of the object for collision avoidance purposes. Nevertheless, a tight representation will always be preferred for more accurate results.

Today collision avoidance is frequently used in different purpose mechanisms [2], [6], [33], [19]. The sensors used to track the motion of these mechanisms sometimes yield the local joint angles and displacements [6] instead of the transformation matrices to the global reference frame [2], therefore computing the forward kinematics is needed. A method that provides and can be used alongside a module to compute the forward kinematics is advantageous in that it simplifies its implementation on different mechanisms.

All the main features of the methods for collision avoidance (CA) studied in this section have been contrasted in Table 2.1.

Observing Table 2.1 it can be inferred that all the methods rely on modelling the geometry with convex shapes. The representation of highly non-convex geometries requires more pre-processing and the computational expense of all the presented methods is dependant on the complexity of the geometry. This work presents a method based on the VPS Algorithm, detailed in Section 2.2, that is not affected by the complexity of the original geometry. Only the resolution employed to mesh the object directly increases the computational requirements.

Another feature that was found overall lacking, is a module able to compute the forward kinematics of the mechanism given the joint angles and displacements. Even if some of the methods affirm to compute the forward kinematics internally [6] they do not provide a module to enable its use on any type of mechanism. In this work we aim to fill this gap by developing a module that can be integrated on any mechanism if needed to compute the forward kinematics.

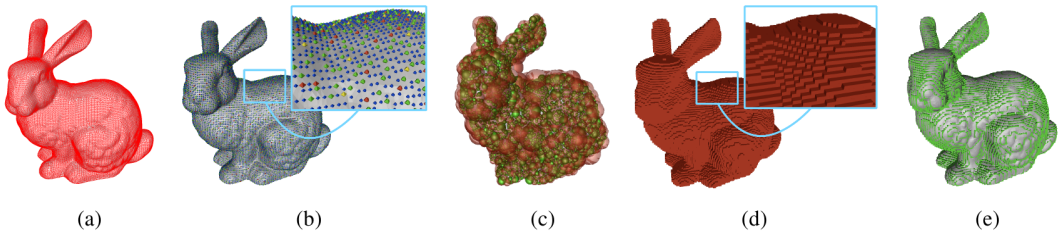
Collision avoidance is handled in different ways throughout the methods presented. Most of these methods do not intend to provide haptic feedback [2, 19] and therefore no forces are computed in order to avoid collisions. Instead, cost function based path planning [2] or precomputed collision paths [19] are used. This work aims to develop a method able to provide collision avoidance based on penalty forces. These forces and torques can be the input of a torque controlled mechanism to avoid collisions or simply used to provide feedback on the magnitude of the forces being generated. These forces and torques can be converted in displacements [5] for position controlled robots. This feature, however, is out of the scope of the present thesis.

2.2 Voxelmap-Pointshell (VPS) Haptic Rendering Algorithm

One of the main features of this work is the use of the Voxelmap-Pointshell (VPS) Algorithm [28] to model geometries, detecet collisions and compute contacts between objects. This is due to its unique traits:

- It allows for the use of any geometry.
- Computational time does not depend on the complexity of the geometry.
- Collisions between multiple objects can be computed within 1 ms [28].
- It is able to compute in a level-of-detail manner, adjusting on the requirements of each simulation.

This force computation and collision detection method is based on the Voxelmap-Pointshell (VPS) Algorithm [28]. Two data structures are used to model each pair of colliding objects: voxelmaps and pointshells. Pointshells, can also be clustered in sphere hierarchies to quickly recognize possibly colliding areas. Both these data structures can be visualized in Figure 2.2 applied to the Stanford Bunny.



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

Figure 2.2: 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). Figure extracted from [28].

Voxelmaps are 3D-grids formed by voxels that contain a discrete value representing their distance to the surface voxel. The voxelmap haptic data structure can be extended to contain floating point surface distance fields. A mixed data structure that merges the advantages of both fields can also be created; fast and accurate penetration and distance computations using modest memory requirements are enabled by this approach.

The pointshell haptic data structure is a point-cloud that samples object surfaces. Each point in the cloud, in turn, has a normal vector pointing inwards of the object surface. The point-cloud is arranged in a point-sphere tree, similarly to [4]. The approach followed by [28] differs in that presented by [4] due to the fact that a down-top building design starting with a high point sampling resolution is seek. This way, points are uniformly distributed. The point clusters are then bound by minimal enclosing spheres [9], creating a hierarchy that allows for fast collision area localization.

Once these haptic data structures have been generated, the VPS algorithm, in charge of computing collisions, traverses the point-sphere hierarchy detecting the likely colliding regions of the pointhshell. The penetration or distance values in these regions are then computed. In the case that the voxelmap overlaps with the pointshell, the VPS algorithm yields the penalty forces corresponding to this con-

tact. As mentioned in Section 1.2, one of the objectives of this thesis is proving a collision computation method that can be used with any type of geometry, regardless of its complexity. An advantageous feature of this algorithm is the fact that the computation speed depends mainly on the sampling of the object and not on its geometry.

All pointshells produced for [28] were generated within 20 seconds and have a size of around 2 MB. Refer to [27] for more information on generation time.

2.2.1 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. [28] provides further insight on the hierarchical traverse algorithm.

In the process of detecting collisions, those in the points in the pointshell belonging to likely colliding regions are checked for their voxel value v in the voxelmap. The points $\mathbf{P}_i$ in the pointshell with a corresponding voxelmap value $v(\mathbf{P}_i) \geq 0$ are colliding points. The force in the pointshell's center of mass $\mathbf{f}_{tot}$ is then computed by adding the colliding points' normals $\mathbf{n}_i(\mathbf{P}_i)$ weighted by their penetration in the voxelmap ($V(\mathbf{P}_i) \geq 0$). Torques $\mathbf{t}_i$ are computed as the cross product between 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. The total torque $\mathbf{t}_{tot}$ is then computed as the sum of these individual torques $\mathbf{t}_i$. This process is summarized in 2.2 and 2.3:

$$\forall i \mid V(\mathbf{P}_i) \geq 0 : \mathbf{f}_i = V(\mathbf{P}_i)) \cdot \mathbf{n}_i \rightarrow \mathbf{f}_{tot} = \sum \mathbf{f}_i, \tag{2.2}$$

$$\forall i \mid V(\mathbf{P}_i) \geq 0 : \mathbf{t}_i = V(\mathbf{P}_i) \times \mathbf{f}_i \rightarrow \mathbf{t}_{tot} = \sum \mathbf{t_i}. \tag{2.3}$$

The voxelmap distance or penetration function $V(\mathbf{P}_i)$ has two components: global and local penetrations, as shown in (2.4):

$$V(P_i) = \underbrace{\mathbf{n}_i \cdot \mathbf{e}_i}_{local} + \underbrace{v(P_i) \cdot \sigma}_{global}. \tag{2.4}$$

The local penetration ($\mathbf{n}_i \cdot \mathbf{e}_i$) is is computed as 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: therefore, it represents the depth of the point within the voxel. On the other hand, the global penetration ($v(\mathbf{P}_i) \cdot \sigma$) is the value resulting from the product between the voxel value in which the point lies ($v(\mathbf{P}_i)$) and the voxel size ($\sigma = s$). As the resolution increases ($s \rightarrow 0$), the influence of the local penetration decreases.

Figure 2.3 illustrates the already presented variables for two overlapping bodies.

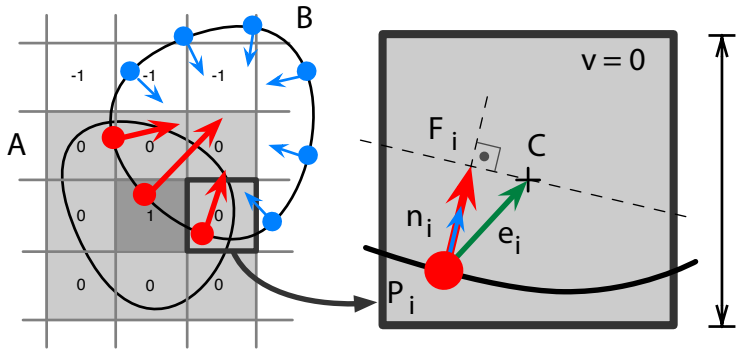


Figure 2.3: Signed distance field (Voxelmap). The surface voxels, the ones overlapping with the gometry have 0 value. Voxels in the outer layers receive $-v$ value, where v is the number of the outer layer count at the studied voxel. Conversely, inner layers are given v value, where v is the inner layer count. Figure by Mikel Sagardia.

When the penetration between voxelmap and pointshell takes a negative value ($V(\mathbf{P}_i) \leq 0$) the distance between these two haptic structures is being measured. In that case, $\max(|V(\mathbf{P} = \mathbf{Q}) \leq 0|)$ represents the distance between the two objects, and $\mathbf{Q}$ the pointhsell point closest to the counterpart voxelmap object.

In has to be noted that, as explained in [34], the quality of the force magnitudes is influenced by the voxelmap resolution while computation time is affected by the number of pointshell points that has to be checked for collision. It is therefore advised, that higher pointshell and voxelmap resolutions are only used in likely colliding areas.

Chapter 3

Generic Mechanism Model

One of the main objectives of this framework is to develop a module that given the joint angles or displacements of each Degree-of-Freedom (DoF) of the mechanism computes the poses of all the frames attached. That is, solving the forward kinematics of the mechanism. In order to achieve this, a configuration file that provides all the necessary information related to the mechanism is built. The module provides a parser for this configuration file that acts as an intermediate component between the mechanism, which provides the angles and displacements of the DoF, and the collision computation algorithm, which receives the pose of every object. Figure 3.1 displays how this module is integrated within the framework.

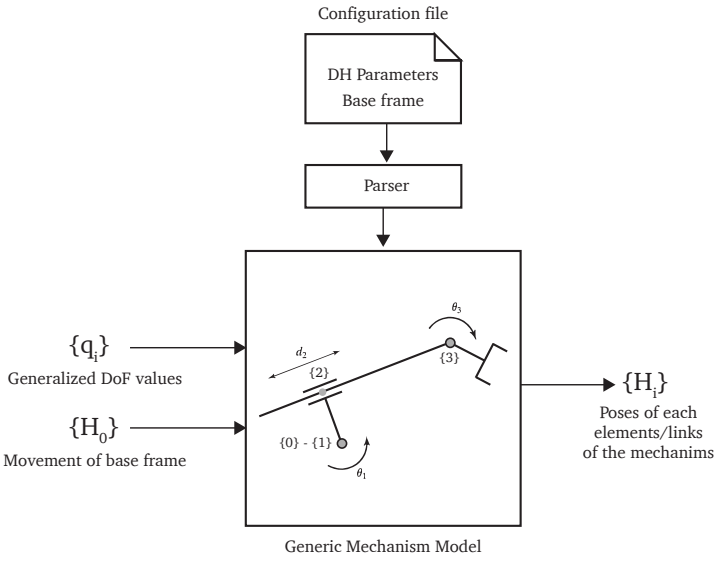


Figure 3.1: Integration of the Generic Mechanism Model module.

The configuration file, where the mechanism is described, can also be used to compute the forces and torques acting on the joints given the forces and torques acting on each object.The focus of this thesis, however, lies on the kinematics. This configuration file specifies the kinematics of a robot by giving the values of four quantities for each link following the Denavit-Hartenberg notation.

Most manipulators are structed from joints that exhibit just one DoF. In the case of a joint having n DoF it can be modelled as n joints of 1 DoF connected with $n-1$ joints of zero length. Therefore, without any loss of generality, only manipulators that have joints with a single DoF will be considered. Robots, specially humanoids, might be formed by more than one kinematic chain; this configuration file contemplates mechanisms containing any number of chains.

3.1 Building the Configuration File to Model the Mechanism

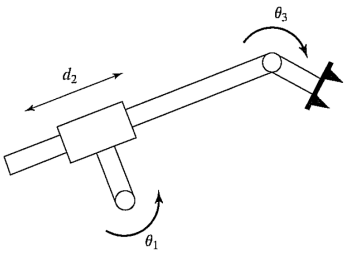
The aim of this section is to describe the steps required to build the configuration file used to later solve the forward kinematics problem and explain what represents each of the parameters involved.

An example mechanism, presented in Figure 3.2, is used throughout this section to exemplify how the configuration file is built.

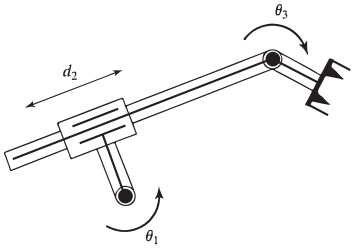
3.1.1 Numbering the Mechanism

The fist step requires the user to number the points in the mechanism where coordinate frames will be attached. A point must be chosen to be used as reference in the mechanism. This will be the point number 0. This point has to be a fixed point in the mechanism and must never be attached to a moving joint. Should it be coincident with a joint, the reference point will be attached rigidly to the fixed part of the joint and an extra frame would be attached to the moving part so that it moves along with it.

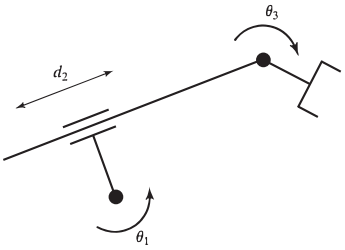
All the joints and end effector points in all the links that constitute the multiple chains of the robot have to be assigned a natural number (starting with 1) so that no number is repeated. The joint angles/displacements (DoF of the mechanism) of the frames located on these points will be the input to compute the transformation matrices, so an orderly and logic numbering is advised.



(a) Example mechanism



(b) Example mechanism overlapping with mechanism sketch



(c) Sketched mechanism

Figure 3.2: Example mechanism acquired from [5].

Next, links are numbered so that link number i is between frame i and $i+1$. Frame i is attached rigidly to link i. Finally, all the remaining points whose transformation matrix has to be computed are numbered continuing the previously started numbering sequence. The points that fall in this group could be fixed points, or intermediate link points whose position is of interest.

Figure 3.3 displays how would this process be carried out in the example mechanism introduced by Figure 3.2.

3.1.2 Affixing Frames to Links

Once all the relevant points have been correctly numbered, as detailed in Section 3.1.1, frames are attached to these points according to the following convention:

- The **Z**-axis of frame i $\mathbf{Z}_i$ is coincident with the joint axis i.
- As a general rule, $\mathbf{X}_i$ points along a_i (the distance from $\mathbf{Z}_i$ to $\mathbf{Z}_{i+1}$ measured along $\mathbf{X}_{i-1}$) in the direction of joint i to joint $i+1$. In the case of the two axes intersecting, assign $\mathbf{X}_i$ to be normal to the plane containing the two axes.

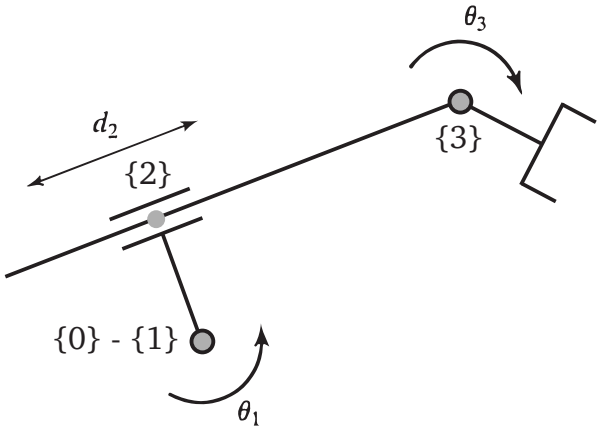


Figure 3.3: Frame numbering for the example mechanism.

- For join i revolute: $\mathbf{X}_i$ is chosen so that it align with $\mathbf{X}_{i-1}$ when θ_i(the angle from $\mathbf{X}_{i-1}$ to $\mathbf{X}_i$ measured about $\mathbf{Z}_i$) $= 0$ and origin i so that d_i (the distance from $\mathbf{X}_{i-1}$ to $\mathbf{X}_i$ measured along $\mathbf{Z}_i$) $= 0$. In the example mechanism joints 1 and 3 are revolute.
- For joint i prismatic: $\mathbf{X}_i$ is chosen so that $\theta_i = 0$ and i's origin is chosen at the intersection of $\mathbf{X}_{i-1}$ and joint axis i when $d_i = 0$. In the example mechanism joint 2 is prismatic.
- $\mathbf{Y}_i$ axis is assigned to complete a right-hand coordinate system.

This convention, however, does not result in a unique attachment, there exists more than one valid configuration, since there are two choices of direction for $\mathbf{Z}_i$. Moreover, in the case of intersecting axes, two choices for the direction of $\mathbf{X}_i$. Figure 3.4 displays how frames are attached in the example mechanism.

3.1.3 Link Parameter Determination

Once every frame has been attached to each link in the chain, the four quantities that describe each link are defined. Two describe the link itself and two describe the link's connection to a nightboring link. In the usual case of a revolute joint, θ_i is called the joint variable, and the other three quantities would be fixed link parameters. For prismatic joints, d_i is the joint variable, and the other three quantities are fixed link parameters. No frame should have more than one variable parameter.

A table is filled for every frame attached to the mechanism indicating the values of the four quantities that are used to describe each link:

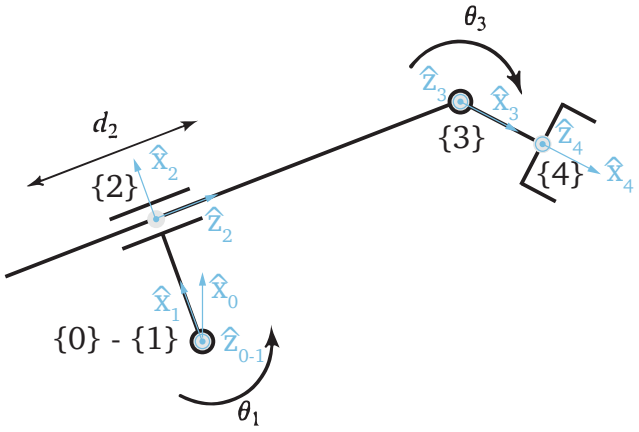


Figure 3.4: Affixing frames to the example mechanism.

i	a_{i-1}	α_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	L_1	0	d_2	0
3	0	90°	0	θ_3
4	L_3	0	0	0

Table 3.1: Denavit-Hartenberg parameter table for the example mechanism (Figure 3.2).

- a_{i-1} = the distance from $\mathbf{Z}_{i-1}$ to $\mathbf{Z}_i$ measured along $\mathbf{X}_{i-1}$
- α_{i-1} = the angle from $\mathbf{Z}_{i-1}$ to $\mathbf{Z}_i$ measured about $\mathbf{X}_{i-1}$
- d_i = the distance from $\mathbf{X}_{i-1}$ to $\mathbf{X}_i$ measured along $\mathbf{Z}_i$
- θ_i= the angle from $\mathbf{X}_{i-1}$ to $\mathbf{X}_i$ measured about $\mathbf{Z}_i$

Table 3.1 showcases the values for the Denavit-Hartenberg parameters for the example mechanism. On the other hand, Figure 3.5 visually displays what magnitudes these values represent.

3.1.4 Configuration File Specification

This section explains how the source code of the configuration file is input. The structure has been designed so that there exist a number of different environments

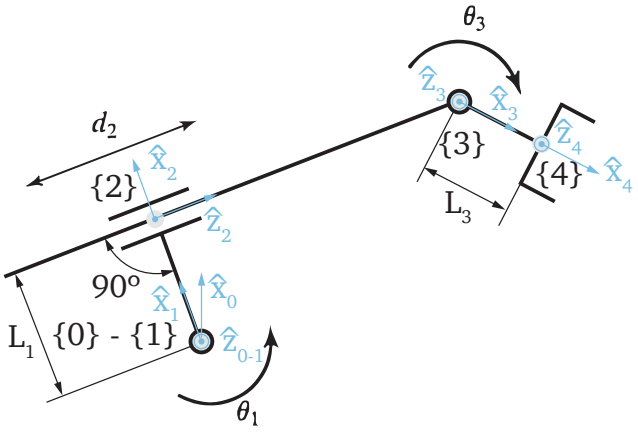


Figure 3.5: Determining Denavit-Hartenberg parameters for the example mechanism.

each of which is used to specify different features of the mechanism. Within each environment a set of commands to input values and choose between different options is available. As a reference to understand the environments and commands detailed in this section the configuration file for the example mechanism is presented in Listing 3.1.

Listing 3.1: Configuration file for the example mechanism.

```
# Length and angle units
units
    length  mm
    angle   degrees
end_units

# Frame of each DoF
mapping
    dof 0       1
    dof 1       2
    dof 2       3
end_mapping

# DH Parameters (L1 = 240, L3 = 160)
chain
    frame       1       0       0       0       theta
    frame       2       240     0       d       0
    frame       3       0       90      0       theta
```

```
    frame          4          160      0          0          0
end_chain
```

Comments:

- Any number of comments can be made using the `#` token at the beginning of the line. Any content following this tag will be ignored.

Unit Specification (optional):

- Environment tags:
  - `units`: This tag specifies the beginning of the `units` environment. All the commands available for the `units` environment can be used after this tag.
  - `end_units`: It indicates the end of the `units` environment.
- Commands:
  - `length unit_name`: It specifies the length units to be used. Possible inputs: mm, cm, m(default), inches.
  - `angle unit_name`: It specifies the angle units to be used. Possible inputs: degrees, radians(default).

Mapping:

- Environment tags:
  - `mapping`: This tag specifies the beginning of the `mapping` environment. All the commands available for the `chain` environment can be used after this tag.
  - `end_mapping`: It indicates the end of the `mapping` environment.
- Commands:
  - `dof dof_id frame_id`: It is used to create a mapping between the array that will contain the values of the degrees-of-freedom and the frame they correspond to. This way, when the array containing the values for each degree-of-freedom is fed to compute the transformation matrices, parser function will be aware of which frame they are attached to.

    * `dof_id`: The id of each degree-of-freedom corresponds to its position in the array. The numbering should start at 0, sequentially continuing with the following positive integers.
    * `frame_id`: It specifies the frame to which the corresponding degree-of-freedom is mapped to. This value has to be a possitive integer.

Chain Creation:

- Environment tags:
  - `chain`: This tag specifies the beginning of the `chain` environment. All the commands available for the `chain` environment can be used after this token.
  - `end_chain`: Indicates the end of the `chain` environment.
- Commands:
  - `reference reference_frame_id`: A positive integer is used to indicate the reference frame to be used for the chain. The reference frame to be used has to be previously defined for another chain or be the global reference frame 0. This command can be omitted when the chain's reference frame is the global reference frame 0.
  - `frame id a_i-1 alpha_i-1 d_i theta_i`: A new frame is created with the specified id and the four parameters that describe the link to which the corresponding frame is attached. These parameters, detailed in Section 3.1.3, have to follow the following rules:
    * `id`: A unique positive integer. Corresponds to the number given in the process of numbering the mechanism.
    * `a_i-1`: A constant real number.
    * `alpha_i-1`: A constant real number.
    * `d_i-1`: For frames attached to prismatic joints, this quantity is variable. This is specified inputting `d`. (See frame 2 on Listing 3.1). Otherwise input a real number.
    * `theta_i-1`: For frames attached to revolute joints, this quantity is variable. This is specified inputting `theta`. (See frames 1 and 3 on Listing 3.1). Otherwise input a real number.
  - Notes:
    * The order in which frames inside a chain are fed is important, as each frame will use the previous as reference.

    - Note: Fixed frames or other points attached to links can be modelled like chains of an only frame, being the parameters constant values. The reference for these points can be specified using the reference token as explained before.
    - Note: The reference frame must not be provided.

General remarks:

- The parser is case sensitive. Commands must be input in lowercase as specified below.
- The parameters that follow the commands can be separated by any number of spaces or tags. All the parameters for the specified command should however remain in the same line. A line break indicates that a new command will be input.
- The configuration file can be saved using any extension. However, .txt or .config is advised.

Chapter 4

Multi-Body Collision Computation Applied to Mechanisms

The main goal of the framework is to detect the collisions between any number of specified bodies while at the same time computing the forces and torques that arise from that contact. This collision computation, in turn, has satisfy some requirements. The first one is to be flexible and extendable, that means, it should be able to be used by users of different needs and requirements in different humanoids or bodies with the minimum amount of setup and modification. Another requirement is that the collision detection should happen in fewer than 1 ms. In the case of the visual modality it is usually enough to employ a 60 Hz rate for video playback, however, for applications where haptic feedback is required, a rate of at least 1000 Hz is usually necessary. This fact has a both a biological and physic ground; while the eye perceives a series of discontinuous frames as fluid movement, starting at a rate of 30 Hz, a much higher rate is needed for the touching perception to realistically feel a virtual contact. Moreover, the haptic system becomes unstable for higher delays when the same stiffness is trying to be achieved. This is due to the active system's energy gain. 1 ms requirement even nowadays is sharp, so very efficient methods are required in which computationally expensive operations have to be avoided. To achieve such speed the parallel computing becomes essential to relieve the large amount of operations and checks performed per loop. The code should as well be written in C++ programming language following the rules of object oriented programming (OOP). Finally, it is desirable that the structure produced is optimized to handle complex mechanisms, such as humanoid robots, frequently used at DLR.

The fulfilment of those requirements set this new project apart from the previously existing framework. The previous structure did also allow for multi-body collision computation, it was, however, not optimised for more flexibility focussing

on mechanisms, in which objects and threads can be grouped for better performance. The result of this project is code that is both flexible and adaptable to different requirements, while at the same time remain efficient to robustly perform in a standard computer.

4.1 Previous Structure

The current work has inherited and built upon many features from the previously implemented version of the multi-body framework. Figure 4.1 displays the previous structure. A configuration file was used to build the scene, in which the voxelmap and pointshell files needed for the computation were specified. These interfaces in charge of storing the voxelmap and pointshell haptic structures are defined as `Objects`. The `Object` class represent the bodies in the scene that are being checked for collision. On top of the structure the `Algorithm` class gathers all the `Objects` in the environment and provides a method to receive the state information of the objects (`receiveStateMatrix()`); this data includes the position or the safety margin needed for the computation of contacts. It also provides a method that outputs the information relative to collisions between all the `Objects` (`sendCollisionMatrix()`) where forces and torques are displayed.

The main feature of this approach is that there exists a thread behind every possible `Object` pair in charge of detecting and computing collisions using the VPS algorithm, explained in Section 2.2. The `start()` and `stop()` methods available in the `Algorithm` class respectively, start and stop the threads that handle the computation of collisions. The threads function asynchronously with respect to the main thread that provides the state matrix for all objects and reads the contact information every loop, independently of whether each thread has been able to compute the collision. This can be understood as if at every moment in time an overall picture of the collision state is computed. On the other hand, if a synchronous thread configuration were to be used, every iteration of the main thread should wait until every subthread completed the computation, which could potentially require extended overall computation time. An example of how this asyncronous behaviour is achieved when computing collisions between a pair of objects, is displayed in Figure 4.2. A collision computation loop between two objects involves:

1. Calling the `Object.receivePose()` method in order to feed the poses of both the objects to the collision computation thread.

2. Calling the `Algorithm.computeCollisionForce()` method in order to compute the forces that arise between the two bodies with the new supplied poses.

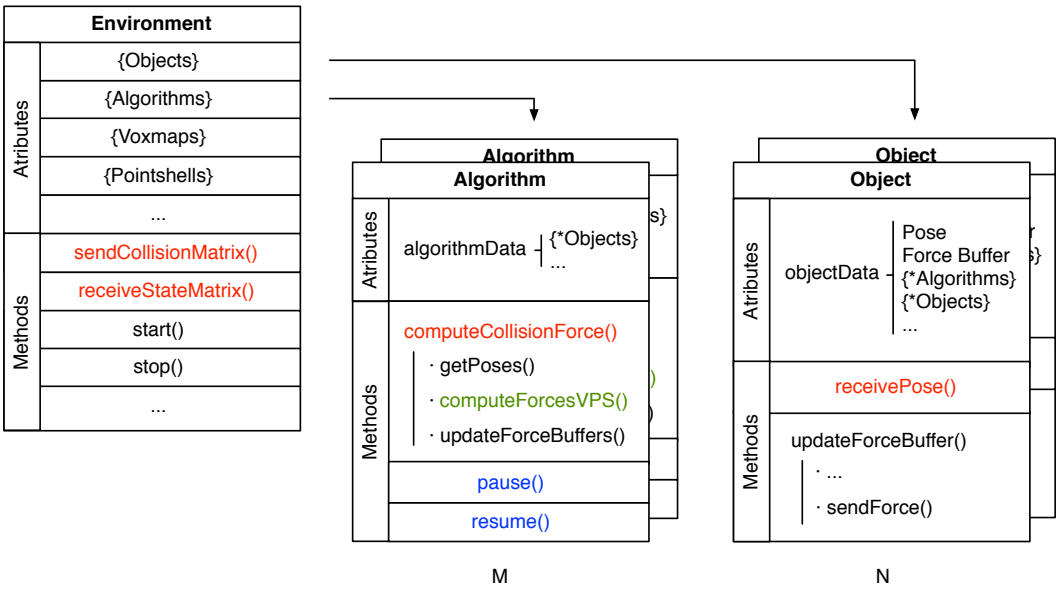


Figure 4.1: Previous class structure for the multi-body framework. The structure is composed of three main classes: `Environment`, `Algorithm` and `Object`. The `Object` class is initiated for each body in the scene. The `Algorithm` class is the link between two objects that can collide and is in charge of the collision detection. `Environment` contains all the others while tracking the status of each. Figure by Mikel Sagardia.

3. Calling the `Object.send()` and `Object.updateForceBuffer()` methods in order to provide the new computed force to the haptic device and update the buffer of the object respectively.

Having decided to use this configuration where every `Object` pair utilize a new thread to compute collisions some problems did arise in larger projects. The initial idea of computing collisions in parallel might seem like an overall positive idea, but as given in (2.1), the number of pairs and consequently threads increases quadratically with the number of `Objects`. For 20 `Objects` 171 threads are required. It was observed that projects that demanded the use of a large amount of threads did not perform satisfactorily. The implemented structure, focuses on the optimisation of the framework for mechanisms. Instead of using completely independent threads, these are grouped in serially computed calls as explained in Section 4.2.

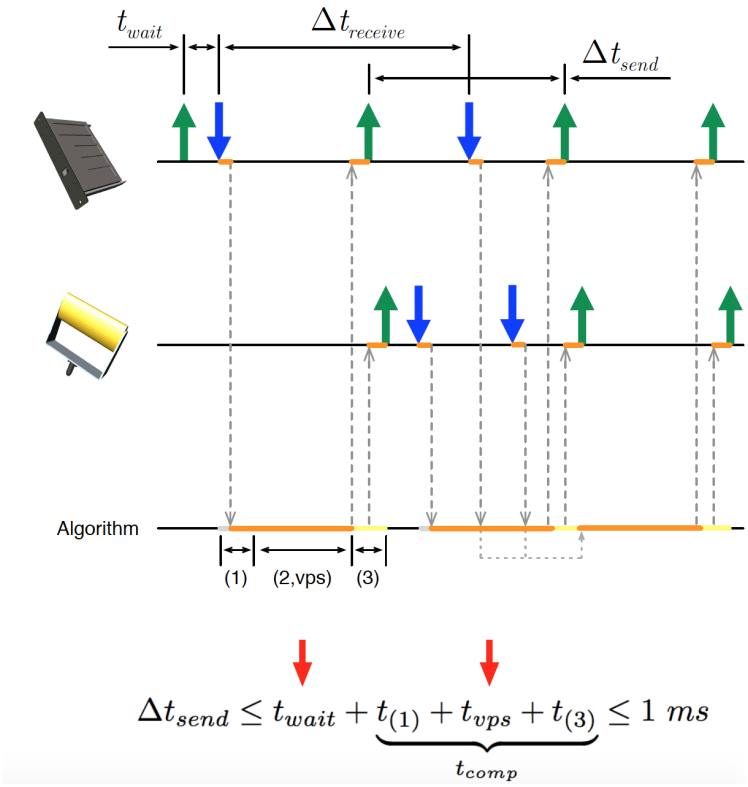


Figure 4.2: Computation process in the previous structure for a single object pair (electronic module and hand): The blue arrows represent packets of information with the position of objects being received. The green arrows depict the forces and torques from the contact being sent. It can be observed that the `Algorithm` thread functions independently from the information reception from the main thread. Δt_{send} is the time period between two consecutive computed forces being sent. $t_{receive}$ is the time period between two consecutive poses being received for an object. $t_{(1)}$ is the time delay between the poses being received and the start of the VPS collision detection computation. t_{vps} is the time required to compute the collisions for the pair of objects. Finally, $t_{(3)}$ is the time delay until both forces computed are sent. Figure by Mikel Sagardia.

4.2 Implemented Structure

This section aims to detail the structure designed to perform the multi-body collision computation. The classes created as well as the main attributes and methods of these classes are presented in Figure 4.3. All the code produced has been written in C++ following the object-oriented programming (OOP) paradigm and has been stored in a library. This library can be loaded into any application to provide the

multi-body collision computation capabilities. As mentioned in Section 6.2, in the future a configuration file along with a parser for this file could be used to setup the whole environment instead of having to create an application.

There are two main classes in the multi-body collision computation class structure: `ObjectStateDB` and `ObjectPairThreadDB`. The `ObjectStateDB` class is a database containing all the `ObjectState` objects (they store all the information relative to the objects in the environment). This class is also able to access and modify all the attributes of the objects it contains. The `ObjectPairThreadDB` class is a database containing all the `ObjectPairThread` objects. Each `ObjectPairThread` object is a thread that computes collisions for the pairs it has been assigned. The `ObjectPairThreadDB` class is able to start and stop all the threads that compute collisions for every object pair.

The overall workflow between classes occurs as follows: First the objects are created specifying its voxelmap and pointshell haptic data structures. Afterwards these objects are added to the object database from which the main functions are called to perform modifications on these objects. Once the object database contains all of the objects that are to be used in the scene, the relations table is created. This table contains the contact state between each possible pair belonging to the object database. The contact state is described by the forces and torques existing between the two bodies, the maximum penetration, and the number of contact points between the two haptic data structures.

The next step is the creation of the required amount of object groups in order to ease the collision detection process. Each one of the object groups will be a separate thread that will run along the other threads belonging to each of the object groups. To make this possible, in each object group possible colliding pairs are added so that there is no collision pair repeated. This way, only possible colliding pairs are computed saving time. Any time during the computation, pairs can be added or removed depending on whether the collision between two target objects is possible or not. At the same time, that these threads are running, the main thread checks for new poses of each object, sending them to the objects in the database when needed. The total force on each object is also computed, taking into account the force yielded on each and every colliding pair it is involved. These forces can then be sent to the corresponding haptic device or processed to add friction or enable a god object mode.

The main classes in charge of enabling the collision computation within the multi-body framework are detailed in sections 4.2.1 to 4.2.10.

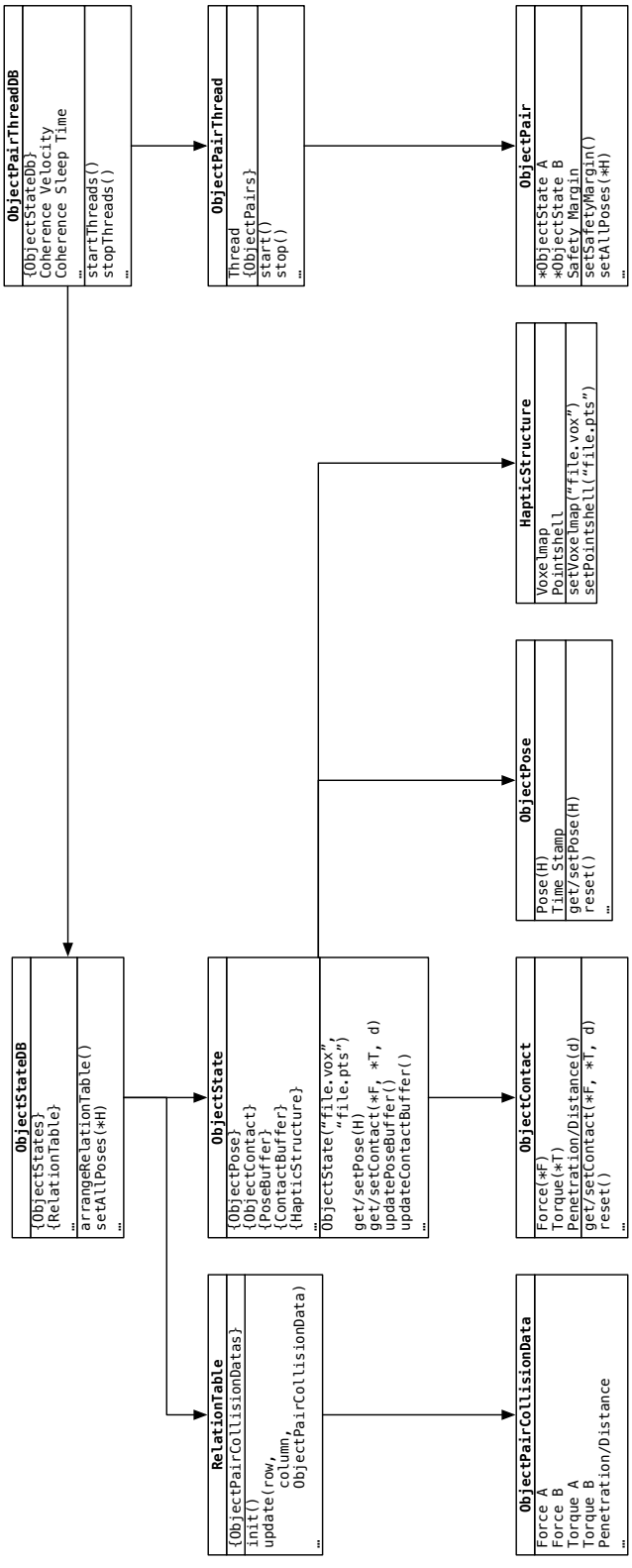


Figure 4.3: Class structure of the multi-body collision computation module.

4.2.1 `ObjectState`

The ObjectState class contains the `ObjectPose`, `ObjectContact` and `HapticStructure` classes, providing methods to access and modify the values of their data members. The different values of the `ObjectPose` and `ObjectContact` classes are sequentially stored in a data buffer. The position data buffer allows for the computation of the mean and instant, linear and angular velocity of each object. This information can later be used to predict motion and avoid future possible collisions.

4.2.2 `HapticStructure`

As explained in Section 2.2, two haptic structures -voxelmpas and pointshells- are used to represent the geometry of each body and compute the forces and torques that might arise from collisions. This class is in charge of storing the voxelmap pointshell structures for their future usage in the collision detection algorithm.

4.2.3 `ObjectPose`

Each object in the database is represented by two fundamental data. One of them describes all related with its position and kinematics while the other details the dynamics: the contact points torques and forces acting on it. The `ObjectPose` class is in charge of storing the kinematic variables of the object in an orderly and accessible way. The objects in this framework obey the rigid body physical laws, thus, the position and orientation is perfectly described using a translation vector from the reference frame to the geometric center of the object and a rotation matrix that represents the rotation of a frame attached to the rigid body with respect to the reference frame. This information is stored in a 3×4 transformation matrix, also used to compute the velocity as its first derivative.

4.2.4 `ObjectContact`

The contact state is constituted by the forces and torques acting on the body and a variable that stores whether the object has collided with any object during the current iteration. One object might be colliding with more that one object at the same time, yielding different forces for each colliding pair. The forces and torques acting on each object pair are stored in the `RelationTable` class. The force and torque stored in this class is the total force and torque acting on a certain object on an instant of time.

4.2.5 `ObjectStateDB`

This class contains all the necessary components to build a database of `ObjectState` objects. It is arranged so that the `ObjectStateDB` class has the main methods to be able to set up all the variables and operate on them. Each object is in turn composed of two subclasses, `ObjectPose` and `ObjectContact`; the former provides information relation to position and the latter related to to the forces and torques that might arise from collision. This class is referenced by pointers in other classes to be able to access all the information related to each object.

4.2.6 `ObectPair`

This class provides the method to compute collisions for the given object pair. As a result, forces and torques as well as the number of contact points and the distance or penetration, in the case of the two objects overlapping, are obtained.

4.2.7 `ObectPairThread`

The `ObjectPairThread` class stores a group of `ObjectPair` objects to compute collisions. This class has methods to add and remove `ObjectPair` objects. It also provides functions to start, stop and pause the thread that is in charge of computing the contacts for each `ObjectPair`.

4.2.8 `ObectPairThreadDB`

This class stores all the `ObjectPairThread` objects that may have been created and is able to start, stop and pause all of them at the same time. This class sets an important difference with the previous structure, where every object pair needed a thread for the computation of contacts. Using this configuration, the user chooses to distribute the pairs in threads in the way he pleases according to the criteria being employed.

4.2.9 `ObjectPairCollisionData`

This class contains all collision data related to each pair of objects that can collide: the closest contact point, the penetration depth (positive when the object pair is colliding, negative otherwise) and the forces and torques on both objects. The total force and torque, computed as a vector sum of all the forces and torques of all the `ObjectPairCollisionData`, is $\mathbf{0}$ at all times.

4.2.10 `RelationTable`

This table stores `ObjectPairCollisionData` objects for all the possible colliding pairs. At each moment in time, this table contains the colliding state of all the objects. Even if it represents a table, objects are internally stored in an array that can be accessed with the two indices of the table: row and column. It has been defined as a 2D table instead as a database due to the fact that it relates every object with all the other in groups of two.

4.3 Implemented Collision Computation

An important feature that differentiates the implementation of the multi-body collision computation framework presented in this thesis from the previous, is the way in which parallel computing is handled. Previously, every object pair was assigned a different thread for collision detection. The implementation presented in this work allows the user decide how many threads are to be created and which object pair checks are going to be performed which threads. This structure has been chosen due to the flexibility it offers when working with mechanisms.

Checking for collisions is a computationally expensive operation; for this reason strategies to avoid unnecessary checks have been applied. First of all, the user must choose a maximum velocity modulus value (v_{max}). In this thesis $v_{max} = 1\,\mathrm{m/s}$ was used; it was empirically verified that no moving object on the trials performed did exceed this limit.

The user has to decide how to distribute the object pair checks in threads so that the collision computation is ready to be started. Each thread works in parallel with all the rest of threads. On the other hand, collision checks for each object pair within a thread are performed sequentially. Collision computation for a given pair of objects in a thread is performed as follows: Once a thread has been started, it waits until a new pose is received. When a new pose is received for the first time the VPS algorithm is called. This algorithm outputs the force ($\mathbf{F}$), torque ($\mathbf{T}$) and penetration or distance (δ). The quotient between the distance and the maximum velocity yields the minimum amount of time (Δt) in which collisions are not going to happen. Knowing the time required for the average cycle (t_s), the quotient between (Δt) and (t_s) yields the number of cycles there is no need to check for collisions; therefore, the thread is idled during this time. Finally, when this time has expired the process is repeated until the thread is stopped.

4.4 Integration of the Multi-Body Collision Computation within the Framework

In this work two separate modules have been developed in order to achieve (self-) collision detection for any type of mechanism. One of them, the generic robot model module explained in Chapter 3, is in charge of solving the forward kinematics problem for the mechanism. The other, the multi-body collision computation explained in this chapter, aims to compute contacts between the objects in the environment. This section illustrates how these two modules are integrated within the rest of the framework.

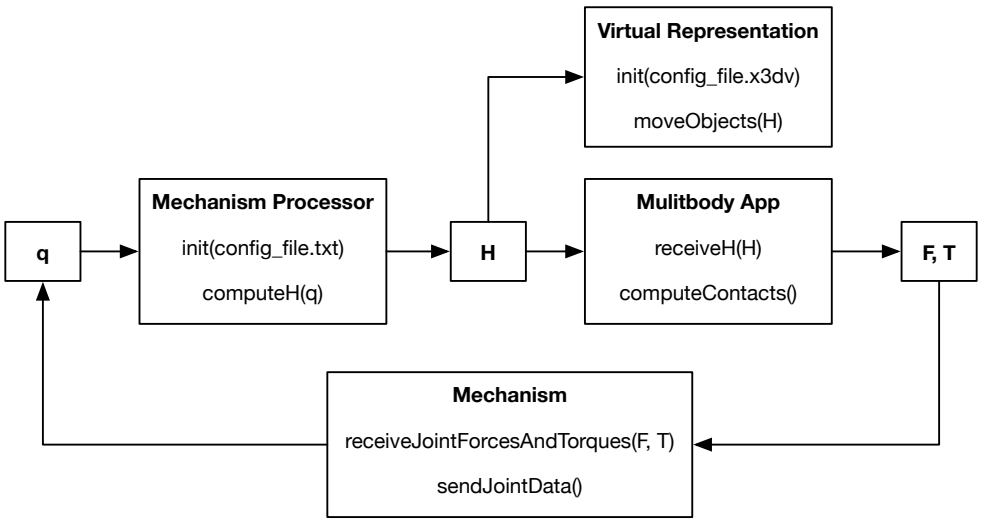


Figure 4.4: The global workflow for the current multi-body framework.

The overall workflow between all the components of the framework is displayed in Figure 4.4. First, the mechanism provides the rotations and translations of each one of its degrees-of-freedom. The mechanism processor module is in charge of transforming this information into the transformation matrices of each of the objects that compose the mechanism. This transformation matrices are then sent to the visualisation of the scene and to the threads that are computing collisions in the multi-body application (the implementation of multi-body collision computation). Afterwards, the object forces and torques yielded are converted into joint forces and torques and

sent to the mechanism. Finally, these forces and torques produce a movement on the mechanism intended to avoid collisions that is sent back to mechanism processor completing the cycle.

Chapter 5

Experiments and Results

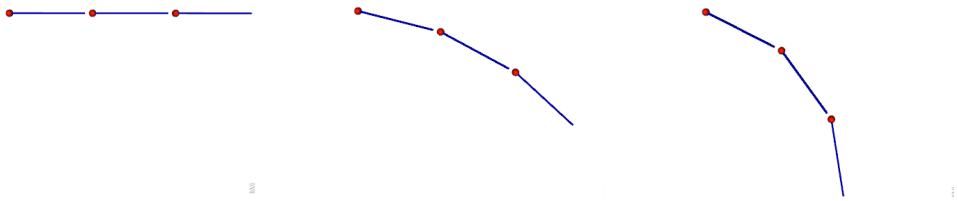
In this chapter, all the experiments performed to check the validity of the framework are detailed. As mentioned in Section 1.2, it is important for the work to be valid to fulfil some requirements:

- Be valid for any type of geometry.
- Be valid for any type of mechanism.
- Operate robustly in real-time settings.

The trials carried on test that this requisites are satisfied at all times. Chapters 3 and 4 detail two distinct parts of this framework: the former, aims to provide a simple way to deal with mechanism kinetics, while the latter explains the collision computation process. These two parts have been tested separately for an in-depth analysis of their behaviour and later included into the final test, aiming to exam the performance of the framework altogether.

5.1 Generic Robot Model

In this experiment, the geometry and the kinematics behaviour is acknowledged to the framework using a configuration file as detailed in Chapter 3. The file specifies the position of each frame attached to each link and whether those frames are allowed one degree-of-freedom rotation or translation. For this test, a simple robot arm formed by 3 links and 3 articulated joints has been used (see Figure 5.1a). The mechanism starts at a horizontal rest position, where all the joint angle values are zero ($q_i = 0, \forall i$). Then, the degrees-of-freedom begin to be articulated following a known *connecting rod-crank* movement as detailed in (5.1). For this purpose no voxelmap



(a) Initial pose. (b) Intermediate pose. (c) Lower end pose.

Figure 5.1: Simple mechanism in motion; three links and joints. Movement corresponds to (5.1)

or pointshell structures are needed due to the fact that no collision is to be computed.

Positive results for this trial must satisfy the already mentioned requirements in addition to providing an accurate depiction of the position of the mechanism during the whole simulation process. The reason for a known movement pattern to be used is that it allows for simple proof of validity based on the already existing mathematical and empirical data.

$$\begin{aligned} \forall i : q_i &= \arcsin\left(0.5 \cdot \sin\alpha\right), \\ \alpha &= f(t). \end{aligned} \tag{5.1}$$

5.1.1 Results

This test has been carried out on a 3.2 GHz Intel Xenon 5060. Using the configuration file showcased in Listing 5.1. During the simulation process carried out at a 1 ms rate the links have produced the expected motion never surpassing the 1 ms threshold, always remaining around 20 μs. Some screenshots of its movement are presented in 5.1. This first test has provided solid data to believe that the *Generic Robot Model* interface behaves as expected fulfilling the imposed requirements. Due to the simplicity of the geometries and the test settings further analysis on more complex geometries and demanding environments has been performed, as detailed in the following sections of this chapter.

Listing 5.1: Configuration for a simple link chain, introduced in Figure 5.1.

```
# SIMPLE CHAIN

units # Lengh/Angle units (SI).
        length  m
```

```
        angle    degrees
end_units

mapping # Frame of each DoF
        #dof     dof_id   frame_id    dof_offset    dof_scale
        dof      0        1           0.0           1.0
        dof      1        2           0.0           1.0
        dof      2        3           0.0           1.0
end_mapping

chain
        #frame   id       a_i-1    alpha_i-1    d_i      theta_i
        frame    1        0        0            0        theta
        frame    2        1.1      0            0        theta
        frame    3        1.1      0            0        theta
end_chain

link
    id 1
    center 1.0 0.0 0.0 0.5  0.0 1.0 0.0 0.0  0.0 0.0 1.0 0.0
end_link

link
    id 2
    center 1.0 0.0 0.0 0.5  0.0 1.0 0.0 0.0  0.0 0.0 1.0 0.0
end_link

link
    id 3
    center 1.0 0.0 0.0 0.5  0.0 1.0 0.0 0.0  0.0 0.0 1.0 0.0
end_link
```

5.2 Force and Torque Computation

Complicated scenarios make it difficult to verify that the forces and torques obtained are indeed accurate. For this purpose, a simple test where prior mathematical calculations yield the correct direction and sense for both forces an torques has been devised.

The Voxelmap-Pointshell (VPS) Haptic Rendering Algorithm [28] computes the forces and torques for each pair of voxelmap and pointshell haptic structures acting on the pointshell in the voxelmap's frame. Afterwards, forces and torques acting on the voxelmap are deducted from the force and torque equilibrium between both objects. (5.2) displays how this is achieved in the case of object A being the voxelmap and B the pointshell.

$$\begin{aligned} \mathbf{F}_B &= {}^W\mathbf{F}_B = {}^W\mathbf{R}_A \cdot {}^A\mathbf{F}_B, \\ \mathbf{T}_B &= {}^W\mathbf{T}_B = {}^W\mathbf{R}_A \cdot {}^A\mathbf{T}_B, \\ \sum \mathbf{F} = 0 &= \mathbf{F}_A + \mathbf{F}_B \rightarrow \mathbf{F}_A = -\mathbf{F}_B, \\ \sum \mathbf{M}_A = 0 &= \mathbf{T}_A + \mathbf{AB} \times \mathbf{F}_B \rightarrow \mathbf{T}_A = \mathbf{F}_B \times \mathbf{AB}. \end{aligned} \tag{5.2}$$

An experiment, using the the framework presented in Chapter 4, involving two rods has been conducted to check that the forces and torques produced respond to the laws of classical mechanics. One of the rods spins around its center-of-mass while the other remains fixed in its initial position. At one point of the first rod's movement contact arises between the two bodies and therefore forces and torques along with it. These values have been studied on the center-of-mass of each rod as seen in Figure 5.2

5.2.1 Results

The magnitude of penalty forces is related to the penalty values, which in the case of VPS are the penalty values of colliding points. It is not in the scope of this thesis investigating it. However, the direction and sense of these determine whether the collision is being computed correctly or not. Positive results for this test are defined as the obtention of forces and torques with reasonably similar or equal direction and same sense as those depicted in Figure 5.2.

Both rods, depicted in light blue, have been conferred a certain degree of transparency to ease the perception of the penetration between the two bodies as it can be observed in Figure 5.3.

The experiment has been been conduted on a 3.2 GHz Intel Xenon 5060. The moving rod has been modelled using 19296 solid voxels while the static rod employed 8152 pointshell points. The average simulation rate was 0.2 ms. Moreover, as shown

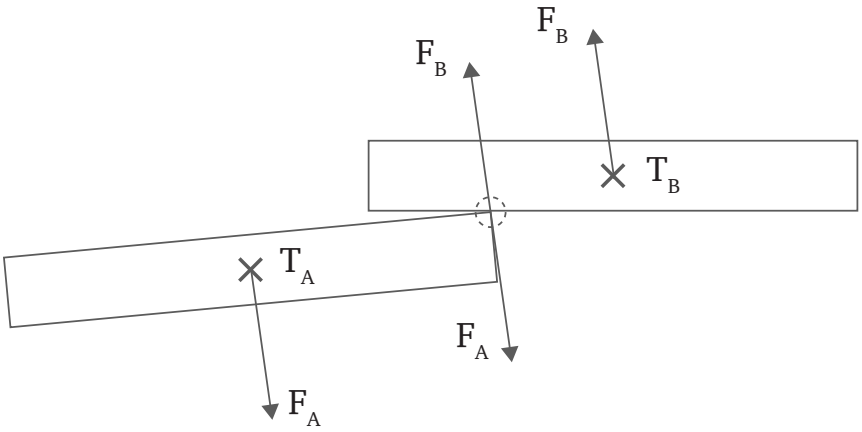


Figure 5.2: Rod force and momentum equilibrium.

Figure 5.3: Virtual representation of the two rods.

in Figure 5.4, the direction of the forces (in red) and torques (in blue) satisfy all the initial requirements and providing positive evidence. Having already tested the collision computation for two bodies the next step is to check the simultaneous collision detection and subsequent computation for multiple bodies that is object of this work.

(a) Contact above moving rod.

(b) Contact below moving rod.

Figure 5.4: Resulting VPS forces and torques for the two rods.

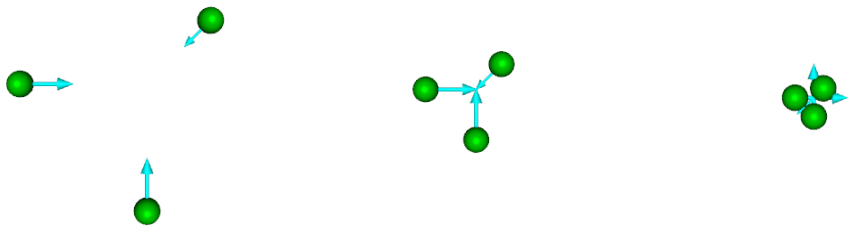
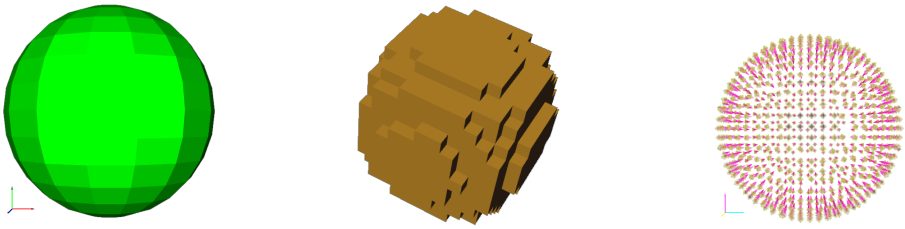


Figure 5.5: Spheres approaching to contact at three different points of their movement cycle. The cyan vectors indicate the direction and magnitude of the velocity of each of the spheres.

5.3 Multiple Collision Detection

In order to check that multiple simultaneous collisions can be handled, a scenario where three spheres are given a motion so that they meet at some point has been produced. These spheres move with uniform linear motion starting at three different corners of an invisible square box containing them. When these bodies hit one of the invisible walls of this box change their sense of movement; this way, they meet again at the center of the box. The module of the velocity of two of these sphere is twice as much as the module of the remaining one. For this reason, the first loop two balls will meet at the center of the box while the following all three will collide at this point. Figure 5.5 displays some screenshots of their representation using InstantPlayer at different points of their movement cycle.

To facilitate the interpretation of results, different colors have been employed to



(a) Original geometry. (b) Voxelmap. (c) Pointshell.

Figure 5.6: Voxelmap and pointshell haptic data structures used to model the spheres. Voxelmap: 61 voxels in **X**, 61 voxels in **Y** and 61 voxels in **Z**. Pointshell: 1066 pointshell points.

display the different variables that are involved in the contact of these three bodies. Velocities are presented in cyan, forces in red and torques in blue. In addition to that, spheres turn red when the number of colliding points is greater than zero, i.e., contact between bodies exists.

For this scenario, where spheres posses 0.05 m radius, a resolution of 0.005 m has been used for both the voxelmap and the pointshell. In the case of the voxelmap, this haptic structure has been produced with 20 outer layers, so as to improve the computation of distances. Both haptic geometries are showcased in Figure 5.6

Three possible colliding pairs exist:

- Sphere 1 with sphere 2.
- Sphere 1 with sphere 3.
- Sphere 2 with sphere 3.

Even if one single thread might be enough to perform the collision detection of these three pairs, each of them has been assigned a separate thread that continuously checks for collisions to test the multi-threaded behaviour of the framework. Splitting the collision checks in different threads increases the overall performance due to the fact that repetitive tasks are executed in parallel rather than sequentially. This is true up to a point: when the number of threads exceed the processor's capacity the efficiency drastically decreases, meaning that some checks should be grouped and checked sequentially. As explained in Chapter 4, one of the mayor points of this framework is to allow the user to decide how to distribute the colliding pairs according to the maximum achievable efficiency in the machine being used or to what it

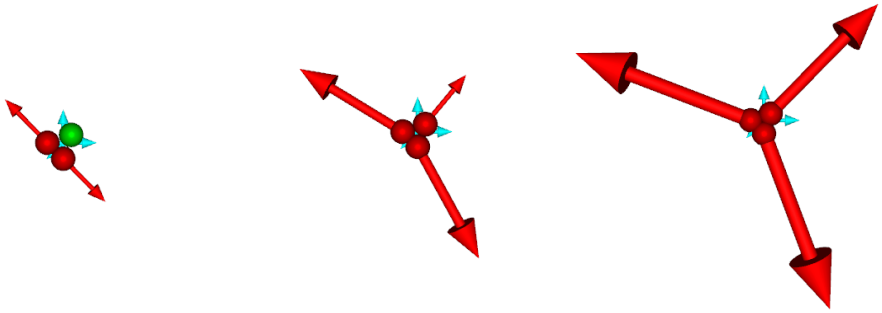


Figure 5.7: Spheres colliding at three different stages. The red arrows represent the forces that arise from contact.

best suits his or her purpose.

These simple geometries should collide producing forces that can be interpreted effortlessly in order to prove their validity. A successful simulation would involve being able to detect collisions of tangent and overlapping bodies at all times simultaneously. In addition to that, forces, torques, velocities and the contact state have to be computed and adequately represented without exceeding the 1 ms restriction.

5.3.1 Results

The achieved representation of the contact variables and velocity can be viewed in Figure 5.7. At the different stages of the motion of these three bodies the collisions are successfully detected and the forces and torques computed accordingly.

In addition to the correct computation of all contact variables, this process must satisfy some requirements to be able to be tested along with real time machines. The trial has been carried out on a 3.2 GHz Intel Xenon 5060. Figure 5.8 displays the results obtained.

5.4 HUG Collision Detection and Avoidance

The final tests includes all the modules built for this framework. A simulator has also been created to reproduce the movements of the real DLR's HUG haptic device and be able to display collisions and torques being sent to this robot. HUG is a bi-manual haptic device consisting of two light-weight robots (LWRs). Information relative to HUG can be found in Appendix A.

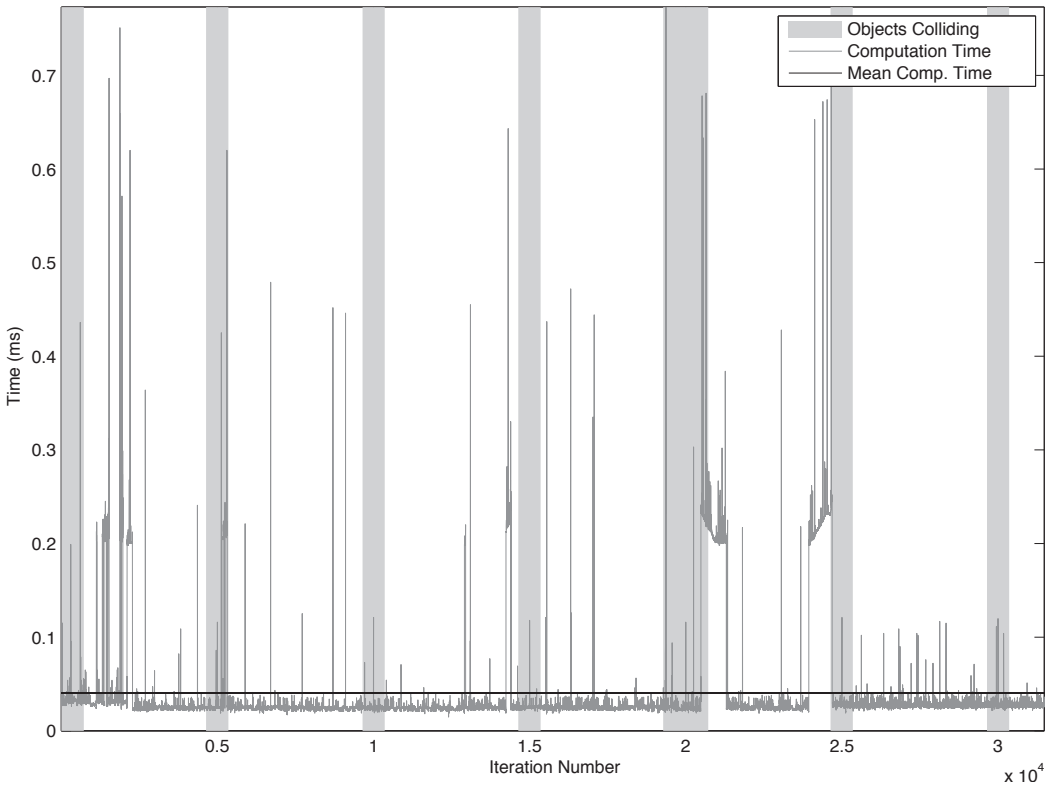
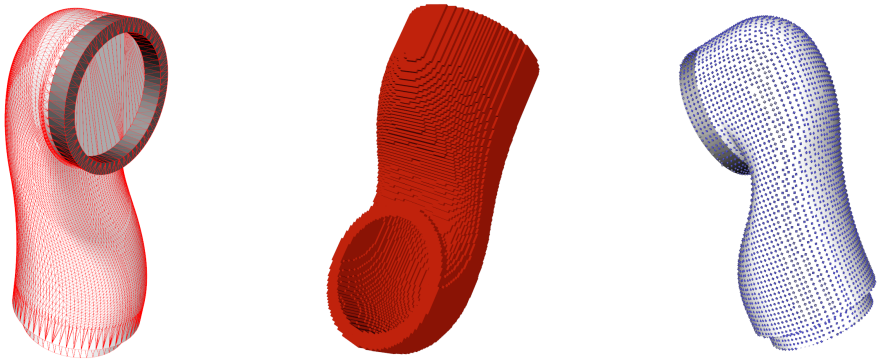


Figure 5.8: Computation time for spheres colliding with no safety margin.

In order to test the behaviour of HUG under test conditions, 14 objects have been used to model its geometry, 7 for each arm. Figure 5.9 displays the original geometry for link 1 along with the voxelmap and pointshell haptic data structures used to model links 2 and 1 respectively. Potentially 78 object pairs should be checked for collision according to (2.1). However, the contiguous objects of each arm do not need to be checked for collisions, since the joint angle limits prevents them from colliding, yielding a total of 66 object pairs. Figure 5.10 shows both the real HUG and the simulator developed to visualize HUG.

The multi-body collision detection framework that has been developed in C++ language works along a Simulink project in charge of controlling the real HUG robot. HUG runs on a realtime machine with a compiled C code. The system model is accessed with the Simulink GUI. Figure 5.11 shows the Simulink block that acts as an interface between the HUG model and the multi-body collision detection framework.

This Simulinik block receives the robot data, that includes the joint angles, then



(a) Link 1 meshed. (b) Link 2 voxelmap. (c) Link 1 pointshell.

Figure 5.9: Original link 1 geometry and pointshell and voxelmap data structures for links 1 and 2 respectively: Figure 5.9a shows the original geometry of link 1. Figure 5.9b shows the voxelmap for link number 2 with a mesh resolution of 0.002 m. Figure 5.9c shows the pointshell for link number 1 with a mesh resolution of 0.004 m.

(a) Real HUG. (b) HUG simulator.

Figure 5.10: Real and simulated HUG. Figure 5.10a shows a human operating the real HUG. Figure 5.10b shows the simulator developed to visualize HUG.

sent to the mechanism simulator. The mechanism simulator then computes the tranformation matrices given these joint angles thanks to the mechanism processor module explained in Chapter 3. Afterwards, this matrices are used to check collisions between all the object pairs, and the forces and torques yielded are transformed into joint torques in order to be sent back to the Simulink application. Having received this torques, they are transformed empirically into an acceptable torque range for the robot through a gain and limited to a maximum to avoid possible threats to the overall safety of the test. This torques can be turned on and off depending on whether they are supposed to be sent to the robot to test its collision avoidance behaviour or not. Finally, a roundtrip delay is computed for the whole process to check the frequency at which it can fluidly perform.

5.4.1 Results

In a first step, collisions were successfully detected and displayed in the simulator built for this purpose. Figure 5.12 displays a virtual self-collision of HUG having introduced a safety margin of 7.5 cm. Once the collisions were consistently detected the computed joint torques were sent to HUG to observe the self-collision avoidance behaviour. The robot did avoid potential collision thanks to the use of the safety margin and the torques generated in the joints that avoided any further penetration increase on the fictitious (due to the safety margin) penetration arouse.

Due to the fact that the simulations are executed at less than 1 ms, a vast amount of data is gathered in a short amount of time if every variable is stored in every single iteration. For this reason, data has been collected at different sampling rates: 500 (1 sample every 500 iterations), 10 (1 sample every 10 iterations) and 1 (all possible samples). Figures 5.13, 5.14 and 5.15 display the results obtained on a 3.2 GHz Intel Xenon 5060.

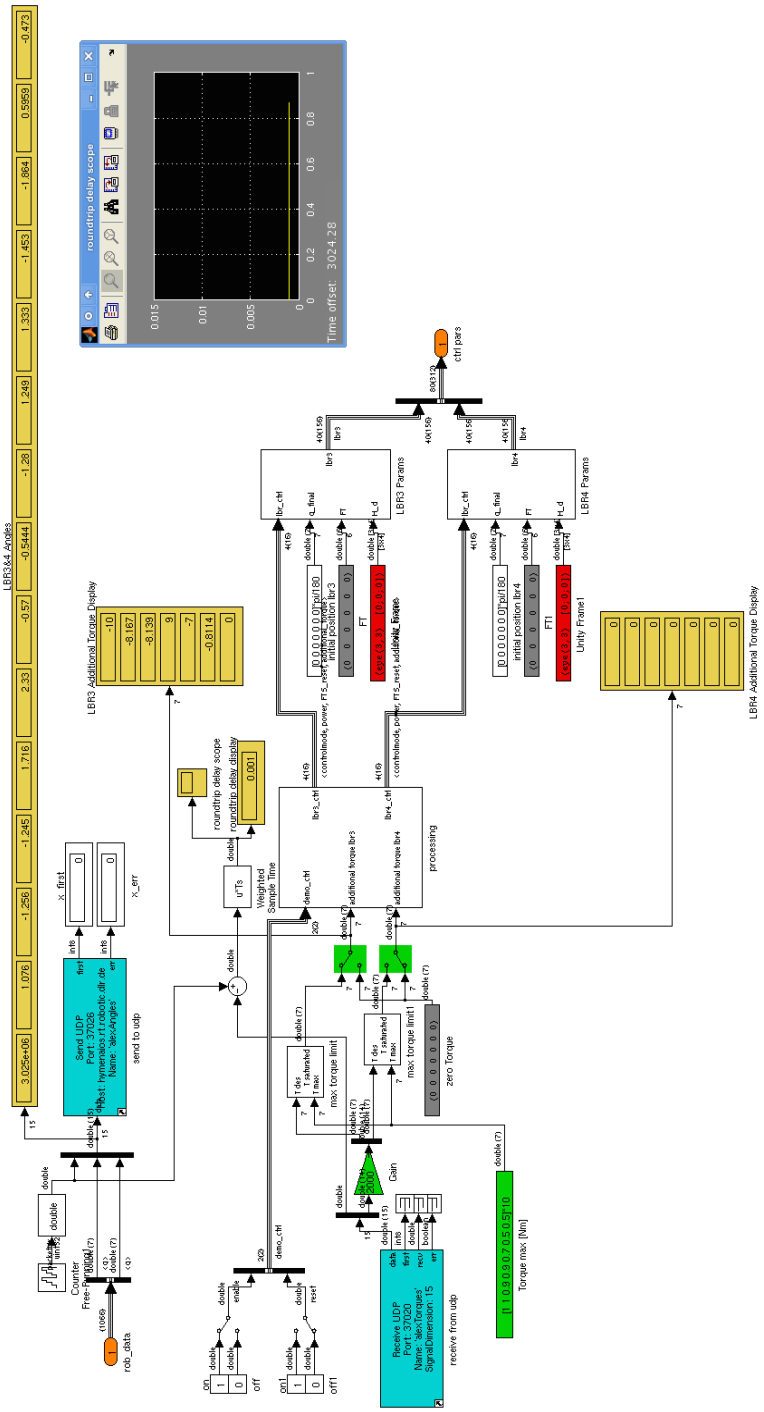


Figure 5.11: Simulink interface: In light blue the input and output ports to receive the joint torques and send the joint angles are depicted. The yellow boxes output the information received or being sent from these ports (in blue). The green shapes account for the safety of the robot, for instance, providing switches to turn on and off the torques sent. The grey and red boxes provide constant values. The graph contained in the blue window pane displayed the roundtrip delay of the whole process. Finally the orange ellipses represent the main input (robot data) and output (control parameters) of the project.

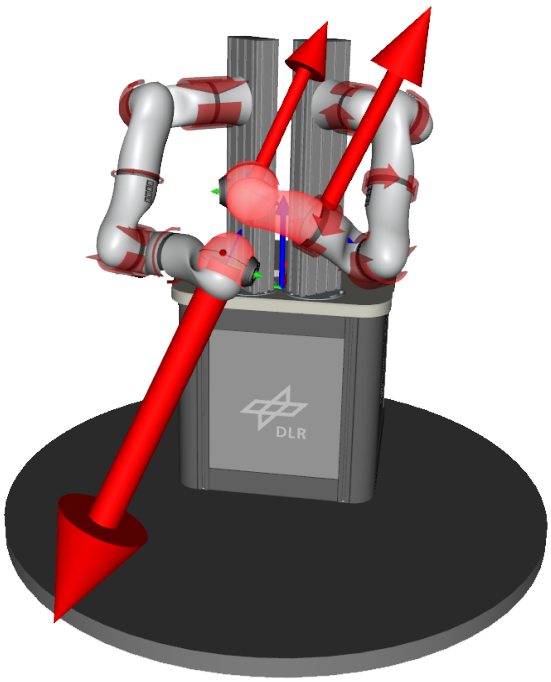


Figure 5.12: Visual representation of a virtual collision in DLR's HUG robot: Colliding objects are represented in bright red, forces are given by the red arrows and torques are depicted as rounded arrows spinning around the joints; their width indicates the magnitude of their value.

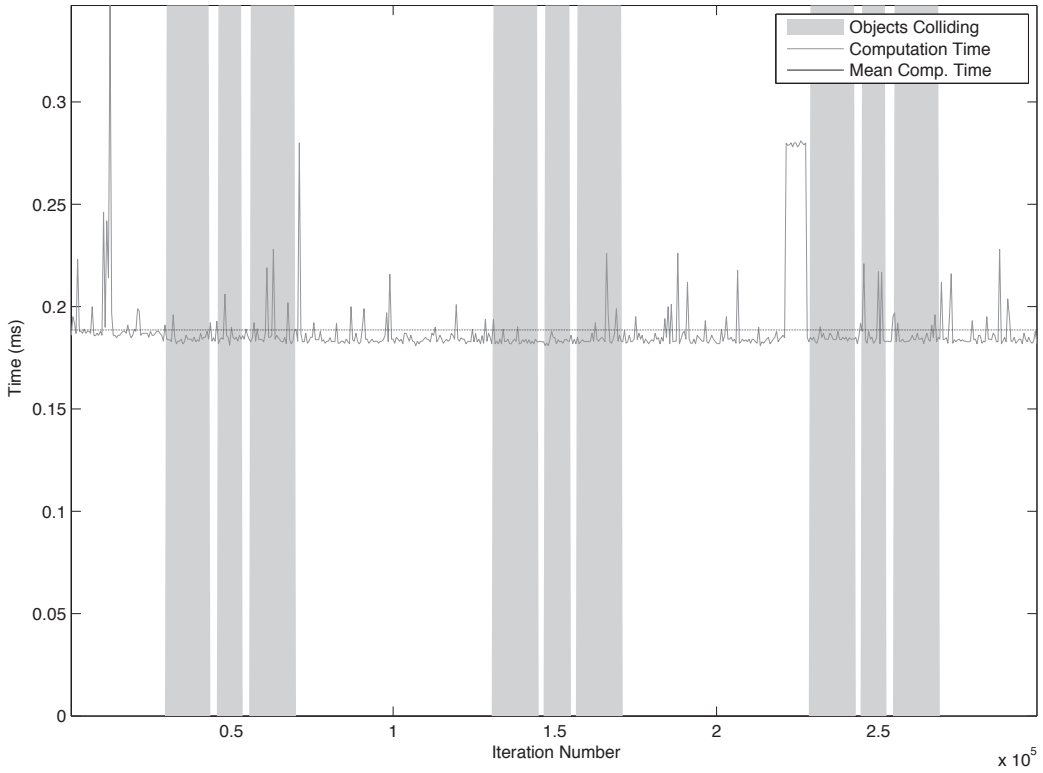


Figure 5.13: Computation time for DLR's HUG collision detection sampling 1 every 500 iterations.

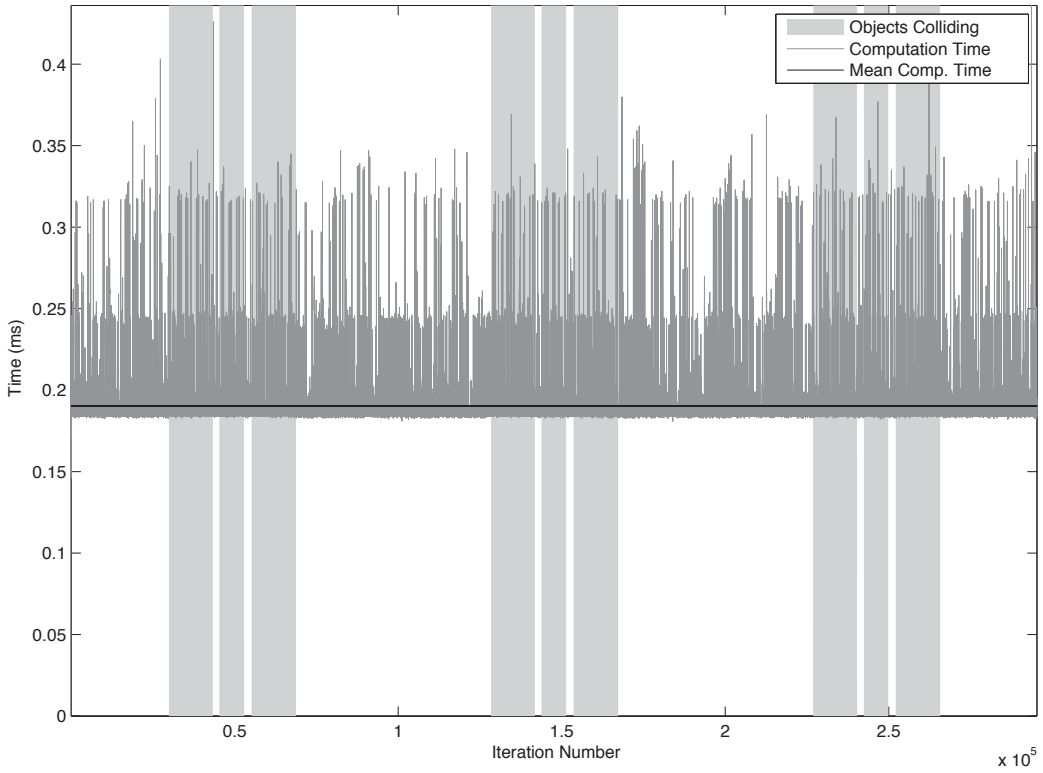


Figure 5.14: Computation time for DLR's HUG collision detection sampling 1 every 10 iterations.

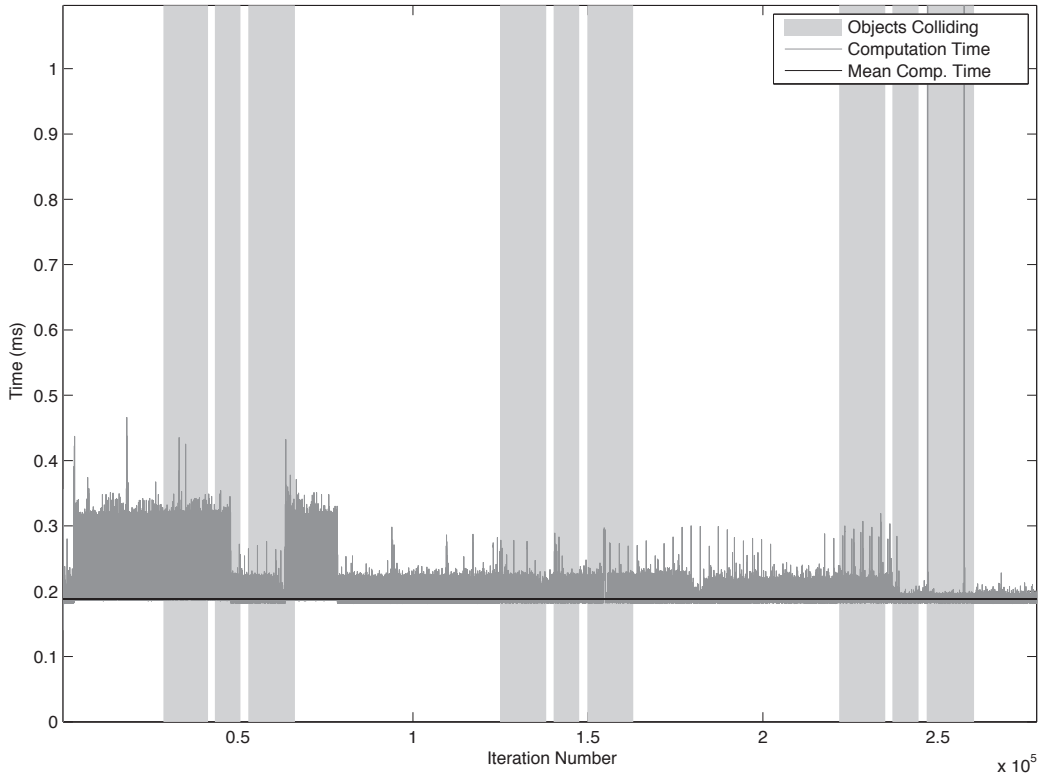


Figure 5.15: Computation time for DLR's HUG collision detection sampling every iteration.

Chapter 6

Conclusions and Future Work

The goal of this thesis was to develop a framework that enables fast collision computation for any type of mechanisms. Collision detection and later handling are one of the fundamental demands on robotic systems to interact with their environment in a safe manner. As detailed in Chapter 2, there already exist many methods with this purpose, and even DLR's previous implementations that compute and handle collisions. However, to the best of my knowledge, none at the date of the production of this work were fast or flexible enough to satisfy the demanding requirements of the Insitute of Robotics and Mechatronics at DLR. These requirements were:

- Being valid for any type of geometries
- Being valid for any type of mechanisms
- Operating robustly in real-time settings, that is in 1 kHz

The output of this work is a multi-body collision detection framework based on the Voxelmap-Pointshell (VPS) Algorithm [28], able to function in 1kHz and improved for mechanisms. Additionally, a generic robot kinematics and dynamics simulator was implemented. These modules, combined, enable self collision computation of complex robot systems and collision avoidance with the environment. Methods, implementation and validation of the experimental data are presented in this report.

The fast and robust collision computation provided by the Voxelmap-Pointshell (VPS) Algorithm modified DLR implementation, was the chosen method for the process of detecting contacts and computing forces between single pairs of objects. Being the multi-body core a problem that increases quadratically in complexity with the number of objects, part of this project's work was to develop a method to handle collisions checks in the most efficient way possible, for which the use of parallel

computing in the CPU has been necessary.

Another major topic of the thesis was developing a method for inputing the structure of a given mechanism to ease the computation of forward kinematics. This method is also capable of computing joint actuator's forces or torques given the forces and torques on each link that arise from collision. Through this implementation the user is enabled to work with several different mechanisms simply by writing a configuration file and working directly with the data that need to be received from and sent to the mechanism.

The framework described in this work has been extensively tested through different means. At an early stage, the analysis was focused on proving that each distinct part that compound this work functioned as expected. This was achieved through the implementation of simple tasks that could be mathematically checked for validity. Afterwards, a simple test involving three spheres was performed. Once every component had performed reliably under the testing conditions, an experiment for a real robot was implemented. First, a simulator was developed to verify that a trial on the real mechanism would result in safe human-machine interaction. To track the human's movement in the simulator a commercial Kinect [23] camera has been used. Appendix B provides further insight on the topic. To track the users position using the real HUG, Vicon camera system has been used. Only when all parts of this framework were tested with positive results, it was integrated on the actual Haptic User Gerät (HUG).

6.1 Conclusions

Evidence has been gathered to support that this improvement of a multi-body collision computation framework and its application to robot (self-)collision avoidance is in the right direction to devise a future where a safe human-machine and machine-machine interaction is possible. There are still many concerns that have to be accounted for, however; collision detection is one of the major and fundamental concerns today [8].

All the initial tests have yielded positive results, not only behaving to the mathematical model but also meeting and even outperforming the already mentioned requirements. It was shown that the HUG device, consisting of two LWRs, can safely avoid self collisions and collision with complex objets in the environment. All this, with 16 objects in the scene with around 70000 triangles altogether and below 1 ms. The successful outcome is due to the hard work of the many people involved

in this project, that have provided many of the basic tools for this framework to successfully perform.

As it may occur in many long-lasting projects, the final output differs slightly from the initial intention. This is due to the inclusion of new ideas, unanticipated needs or events. However, this fact can have a positive impact on the outcome; solving the problems that arise and finding answers to unforeseen requirements result in a more robust and flexible production. In the present work, the following modifications with respect to the initial plan have been performed:

- The initial goal of the project was to test the framework on the humanoid robot Justin (see Figure 6.1) the bimanual DLR haptic device HUG was used instead [15]. All results obtained with HUG are translatable to Justin. The framework can be implemented following the exact same steps required for HUG in Justin.
- The original idea was to test the framework checking for collisions between all the objects that integrate a certain mechanism. Nevertheless, when the human is added to the group of objects checked for collisions, human-machine collision detection is enabled. To track the position an movement of the human, Kinect[23] camera sensor has been used. New methods have been created to read from the data stream of the device and transfer this information to the geometry modelling the human shape. Further information on this topic can be found in Appendix B.

Figure 6.1: DLR's Justin humanoid robot.

The framework, however, has some limitations. In this project a framework has been developed where the real and virtual representations have to be synchronized

for a correct computation of results. If the virtual model differs considerably in position and orientation from the real mechanism, the output would be inaccurate. Therefore, careful calibration as well as a rigorous configuration file are required. The results are likewise subject to the accuracy of devices being used; inaccurate position tracking would result in collisions not being detected or false positives. The speed of the computer's processor being used to handle the VPS collision detection can also have an influence on the results; faster processors would allow for higher sampling rates improving the accuracy of collision detection and the haptic feedback in the case of forces being computed. The user, has to take into account all of these variables and establish a safety margin (which act as a safety layer around the robot links) that secures correct collision computation for the object application.

6.2 Future Work

To the date of writing, the framework has already been implemented for the HUG robot. In the future, more robotic mechanism could test this framework. Presumably, DLR's Justin [7] could carry into effect the self-collision detection module. Not limited to humanoid robots, this framework can prove to be valuable in several different scenarios where collision checks between bodies are required.

Being such a versatile collision computation module, the number of scenarios where it could be tested is virtually unlimited. Assembly lines could include it to increase the performance of robots interacting on the same objects. Not only collisions can be detected, but also the exact point where contact has been produced opening the door for new applications.

As for the framework in itself, there are some improvements that could be performed. A configuration file to setup the entire environment could be implemented, in the fashion [28] uses for its setup. This would bring the user control over all the parameters without the need of mastering any specific programming languages. Even for those proficient in C++ a configuration file to setup the project could save considerable time.

Currently, the mechanism parser along with the mechanism processor are capable of computing the forward kinematics of a given robot and the actuator forces and torques given the forces and torques acting on the links of each chain. However, there are many more possibilities in this field of robotics. For instance, given the linear and angular velocities of a certain mechanism the inertia forces and torques can be computed or given a path the module could compute the forces and torques

required to follow the given trajectory as detailed in [5].

DLR's VPS implementation [28] requires the existence of penetrations to compute forces and torques. In reality, contact occurs without perceivable penetration between bodies. The God-Object Method [25] bridges this gap, enabling the VPS method to compute forces and torques with the required penetration but then rendered as if they were occurring in the surfaces of the objects. This method has been tested in many other applications with positive results and could prove to be useful for this particular work. Furthermore, some other modules to provide additional functionalities could be implemented on the framework, similar to the kinetic energy damping design or the emergency stop algorithm [6] implements.

Bibliography

[1] Vicon camera system. http://www.vicon.com/System/TSeries, May 2015.

[2] Lucian Balan and Gary M. Bone. Real-time 3d collision avoidance method for safe human and robot coexistence. In *International Conference on Intelligent Robots and Systems*, 2006.

[3] C. Borst, C. Ott, T. Wimböck, B. Brunner, F. Zackarias, B. Bäuml, U. Hillenbrand, S. Haddadin, A. Albu-Schäffer, and G. Hirzinger. A humanoid upper body system for two-handed manipulation. In *Proc. of IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, 2007.

[4] J. Barbič and D.L. James. Six-dof haptic rendering of contact between geometrically complex reduced deformable models. *IEEE Transactions on Haptics*, 1(1):39 –52, jan.-june 2008.

[5] John J. Craig. *Introduction to Robotics*, volume Third Edition. Pearson Education International, 2005.

[6] Alexander Dietrich, Thomas Wimböck, Holger Täubigy, Alin Albu-Schäffer, and Gerd Hirzinger. Extensions to reactive self-collision avoidance for torque and position controlled humanoids. In *IEEE International Conference on Robotics and Automation*, 2011.

[7] German Aerospace Center (DLR). Humanoid space justin. http://spectrum.ieee.org/automaton/robotics/humanoids/space-justin, April 2015.

[8] Chister Ericson. *Real-time Collision Detection.* Morgan Kaufmann Publishers, 2005.

[9] Kaspar Fischer and Bernd Gärtner. The smallest enclosing balls of balls: Combinatorial structure and algorithms. *International Journal of Computational Geometry & Applications*, 2004.

[10] Kinect Fisioterapia. Human standard proportions. http://fissioterapia.blogspot.com.es/2015/01/hombre-de-virtuvio-y-la-historia-de-la.html.

[11] E. Freund and J. Rossmann. The basic ideas of a proven dynamic collision avoidance approach for multi-robot manipulator systems. In *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1173–1177, October 2003.

[12] Elmer G. Gilbert, Daniel W. Johnson, and S. Sathiya Keehrthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 1988.

[13] S. Gottschalk, M. C. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *Proceedings of ACM SIGGRAPH '96*, 1996.

[14] S. Haddadin, A. Albu-Schäffer, A. De Luka, and G. Hirzinger. Collision detection & reaction: A contribution to safe physical human-robot interaction. In *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3356–3363, September 2008.

[15] Thomas Hulin. Hug - the dlr haptic user gerät. http://www.dlr.de/rmc/sr/en/desktopdefault.aspx/tabid-8750/.

[16] Thomas Hulin, Katharina Hertkorn, Philipp Kremer, Simon Schätzle, Jordi Artigas, Mikel Sagardia, Franziska Zacharias, and Carsten Preusche. The dlr bimanual haptic device with optimized workspace (video). In *Proc. of IEEE International Conference on Robotics and Automation*, 2011.

[17] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *The International Journal of Robotics Research*, pages 90–98, Spring 1986.

[18] O. Khatib, K. Yokio, O. Brock, K. Chang, and A. Casal. Robots in human environments. In *Proc. of IEEE First Workshop on Robot Motion and Control*, pages 213–221, 1999.

[19] James Kuffner, Koichi Nishiwaki, Satoshi Kagami, Yasuo Kuniyoshi, Masayuki Inaba, and Hirochika Inoue. Self-collision detection and prevention for humanoid robots. In *Proc. of IEEE International Conference on Robotics and Automation*, May 2002.

[20] M. Lin. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, U.C. Berkeley, U.C Berkeley Department of Electrical Engineering and Computer Science, 1993.

[21] H. Liu, X. Deng, and H. Zha. A planning method for safe interaction between human arms and robot manipulators. In *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1814–1820, August 2005.

[22] B. Martinez-Salvador, A. P. del Pobil, and M. Pérez-Francisco. A hierarchy of detail for fast collision detection. In *Proc. of IEEE International Conference on Intelligent Robots and Systems*, pages 745–750, 2000.

[23] Microsoft. Kinect for xbox 360. `http://www.xbox.com/en-US/xbox-360/accessories/kinect`.

[24] B. Mirtich. Vclip: Fast and robust polyhedral collision detection. In *ACM Transactions on Graphics*, July 1998.

[25] Michael Ortega, Stephane Redon, and Sabine Coquillart. A six degree-of-freedom god-object method for haptic display of rigid bodies. Technical report, i3D-INRIA, 2006.

[26] S. Quinlan. Efficient distance computation between non-convex objects. In *Proc. of IEEE International Conference on Robotics and Automation*, pages 3324–3329, 1994.

[27] Mikel Sagardia, Thomas Hulin, Carsten Preusche, and Gerd Hirzinger. Improvements of the voxmap-pointshell algorithm - fast generation of haptic data-structures. In *53. IWK - TU Ilmenau*, 2008.

[28] Mikel Sagardia, Theodoros Stouraitis, and Joao Lopes e Silva. A new fast and robust collision detection and force computation algorithm applied to the physics engine bullet: Method, integration and evaluation. In *EuroVR 2014 - Conference and Exhibition of the European Association of Virtual and Augmented Reality*. The Eurographics Association, 2014.

[29] A. De Santis, A. Albu-Schäffer, C. Ott, B. Siciliano, and G. Hirzinger. The skeleton algorithm for self-collision avoidance of a humanoid manipulator. In *Proc. of IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, 2007.

[30] F. Seto, K. Kosuge, and Y. Hirata. Self-collision avoidance motion control for human robot cooperation system using robe. In *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 50–55, August 2005.

[31] O. Stasse, A. Escande, N. Mansard, S. Miossec, P. Evrard, and A. Kheddar. Real-time (self-)collision avoidance task on a hrp-2 humanoid robot. In *Proc. of*

IEEE International Conference on Robotics and Automation, pages 3200–3205, May 2008.

[32] H. Sugiura, M. Gienger, H. Janssen, and C. Goerick. Real-time self collision avoidance for humanoids by means of nullspace criteria and task intervals. In *Proc. of IEEE/RSJ International Conference on Humanoid Robots*, pages 575–580, December 2006.

[33] Holger Täubig, Berthold Bäuml, and Udo Frese. Real-time swept volume and distance computation for self collision detection. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[34] Rene Weller, Mikel Sagardia, David Mainzer, Thomas Hulin, Gabriel Zachmann, and Carsten Preusche. A benchmarking suite for 6-dof real time collision response algorithms. In *ACM Virtual Reality and Software Technology*, 2010.

[35] Robin Wolff, Carsten Preusche, and Andreas Gerndt. A modular architecture for an interactive real-time simulation and training environment for satellite on orbit servicing. In *15th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, 2011.

Appendices

Appendix A

Haptic User Gerät (HUG)

HUG, the DLR Haptic User Gerät [16] has been developed with the aim to achieve the most realistic force-feedback for DLR's sophisticated haptic applications. HUG is a bimanual haptic device composed of two Light-Weight Robot (LWR) arms (see Figure A.1). The DLR LWR is composed by a series of links united by revolute joints being very light in weight and flexible. Its sensory equipment has been especially designed to work in the field of human interaction. It mimics the human arm in both size and power. Furthermore, gripper tools can be connected thanks to its standard robot interface flange.

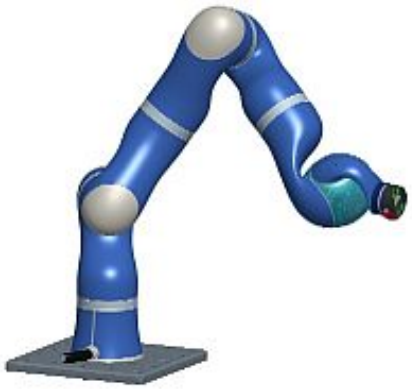


Figure A.1: Light Weight Robot representation.

Each LWR arm weighs just 14 kg being able to work with loads up to 14 kg. All the electronics are integrated into the robot arms. Each of the Light Weight Robot's joints has a motor position sensor and a sensor for joint position and joint torque. This enables the user to operate the robot position, velocity and torque controlled, resulting in a highly dynamical system with active vibration damping. The hardware specifications of this LWR arm are detailed in Table A.1.

Total Weight	14 kg
Max. Payload	7 kg
Max. Joint Speed	120°/s
Sensor angular resolution	20"
Maximum Reach	936 mm
Nr. of Axes	7 (R - P - R - P - R - P - R)
Motors	DLR-Robodrive
Gears	Harmonic Drive
Sensors (each Joint)	2 Position, 1 Torque Sensor
Brakes	Electromagnetic Safety Brake
Power Supply	48 V DC
Control	Position-, Torque,- Impedance Control
Control Cycles	Current 40 kHz; Joint 3 kHz; Cartesian 1 kHz
Electronics	Communications by optical SERCOS-Bus

Table A.1: Hardware Specifications according to A.

HUG has been assembled with two of these robots that are mounted behind the user, such that the intersecting workspace of the robots and the human arms becomes maximal. Equipped with a thorough safety architecture in hard- and software, HUG assures safe operation for human and robot.

Figure A.2: A human operating with HUG.

A particularly advantageous characteristic of HUG is its capability of generating

high interaction forces in a comparably large workspace. Various hand interfaces and additional vibro-tactile feedback devices are available to enhance user interaction. Additionally, sophisticated control strategies improve performance and guarantee stability. To this end, HUG is well suited for versatile applications in remote and virtual environments:

- Telemanipulation of Justin using HUG, shown in Figure A.2.
- Virtual assembly simulations in which stiff collisions and smooth sliding are possible [3].
- Training of astronauts and mechanics [35].
- Rehabilitation.

Appendix B

Towards Human-Robot Interaction

In robotics, a high level of safety and reliability is only ensured if self-collisions and collisions with the environment can be completely excluded. Self-collisions involve two or more geometries of the mechanism overlapping with each other. The robot, however, is only aware of its own geometry through the setup performed by the user. Therefore, when the geometry of the human is rendered as part of the robot's geometry, self-collision avoidance results in human-robot collision avoidance. To achieve this, there are two basic requirements:

- A virtual representation of the human geometry .
- A method to track the position or relative movement of each of the joints of the human whose movement is to be studied.

One common virtual representation for humanoid motion would be an skeletal surface mesh along with a skeletal joint structure. The mesh, provides the outer shape which can be rendered with a texture for more realistic results. The skeletal joint structure, in turn, contains the rotations of each of the degrees-of-freedom found in the human body; these values are then used to animate the mesh accordingly, deforming the mesh in the bent regions displaying elasticity. In robotics, due to the need of computing distances between distinct body parts and penalty forces, this type of representation is unfrequent.The framework presented, makes use of the voxelmap-pointhsell haptic structures [28] that are used to represent and compute collisions between rigid bodies. At present, the deformations that might arise from collisions are not computed. Therefore, the human geometry has been divided in different rigid objects that assembled together represent the human shape. Figure B.1 illustrates how this is performed.

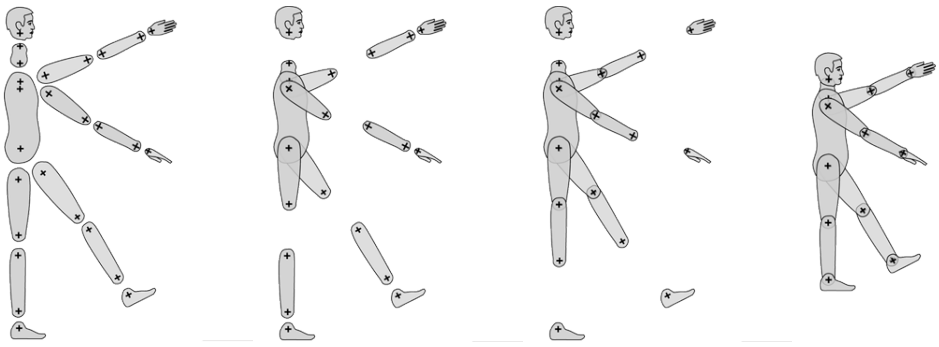


Figure B.1: Assembly of the human geometry.

One of the main problems of creating a rigid geometry to model the human is the difficulty to create a unique representation that is valid for all the different potential users. As stated in Chapter 1, one of the main requirements of this framework is to be flexible, and that implies that different users have to be able to make use of it. For this reason, a module to produce the geometry to model the user's body has been created. The simple version only requires one parameter to be specified; the user's height. All the geometries are then created according to the standard human proportions, which can be found in Figure B.2. In the case that a more accurate representation is needed, the user is enabled to change each of the contours of the geometries.

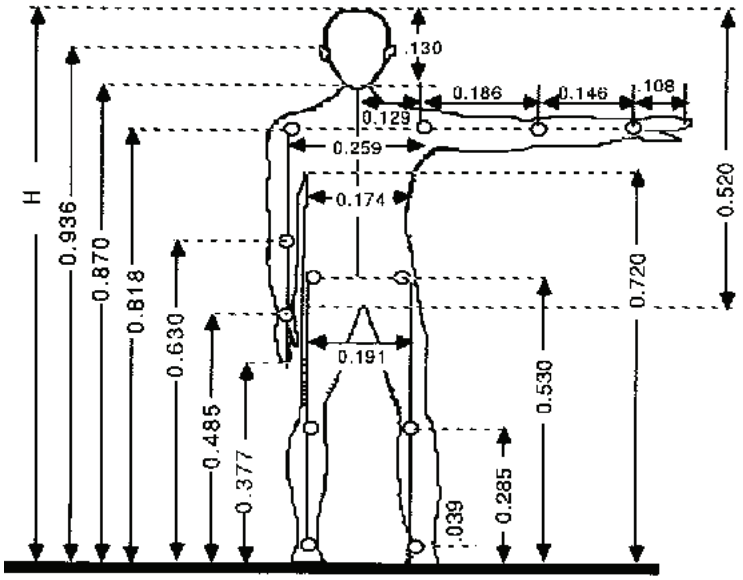


Figure B.2: Standard human proportions according to [10].

body part	geometry	length	contour
torso	ellipsoid	$h \cdot 0.16$	$h \cdot 0.67$
head	cuboid	$h \cdot 0.34$	$h \cdot 0.36$
biceps	ellipsoid	$h \cdot 0.19$	$h \cdot 0.24$
forearm	ellipsoid	$h \cdot 0.15$	$h \cdot 0.20$
hand	ellipsoid	$h \cdot 0.11$	$h \cdot 0.11$
thight	ellipsoid	$h \cdot 0.25$	$h \cdot 0.37$
calf	ellipsoid	$h \cdot 0.22$	$h \cdot 0.24$
foot	cuboid	$h \cdot 0.15$	$h \cdot 0.20$

Table B.1: Dimension for the virtual human geometry.

When the human is within the robot's workspace, specially high safety margins are used to protect the human from collisions. The safety margin, as explained in Chapter 5, acts like a protective shell around the geometry that for the collision computation method is no different from the actual geometry. Consequently, when wide safety margins are employed, a highly accurate representation of the body is not strictly necessary; the priority is to bound the human geometry within the desired safety distance. As a result, given the height (h) this framework models the human geometry using cuboids and ellipsoid for the different parts that constitute the human body. Table B.1 displays how this is achieved.

Having created the human geometry the Kinect provides a data stream containing the poses of the different joints of the skeleton when tracking the user. The hands-free tracking is possible thanks to an infrared projector and camera that track the movement of objects and individuals in three dimensions. An instance of the skeleton along with the depth image is shown in Figure B.3.

The joint poses provided by Kinect are used to animate the human geometry by mapping each joint to the corresponding object. This way, the motion of the user being tracked is replicated by the virtual object assembly created to model the user's anatomy. Figure B.4 shows virtual representation of the user interacting with HUG A.

Further experiments will be conducted for human-robot interaction using this framework with Kinect as tracking device. Videos for this thesis have been recorded

Figure B.3: Skeleton and depth image provided by Kinect.

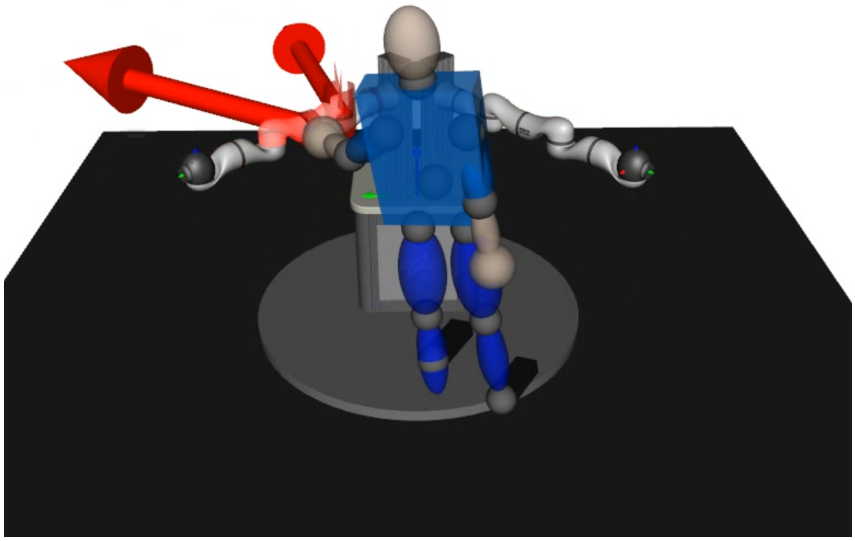


Figure B.4: Simulation of human body interacting with HUG through Kinect tracking.

that show how human-robot interaction is achieved in a simulator, using Kinect as position-tracking device. To the date of the writing, Kinect has only been tested with the HUG simulator using this framework. The experiments carried out on the real HUG used Vicon [1] as position-tracking device for the human body.