An Integrated Robot Simulation Workspace for Real Time Collision Free 3D Laser Scanning Applications

Alexandru Dumitrache **Theodor Borangiu**

University Politehnica of Bucharest, Romania
Centre for Research & Training in Robotics and CIM (CIMR)
(e-mail: alex@cimr.pub.ro, borangiu@cimr.pub.ro)

Abstract: This paper presents a methodology, related techniques and tools for robot modelling and simulating, with focus on real-time collision detection and avoidance. Applications of the methods presented include collision-free path planning for 3D scanning, and collision avoidance in real-time robot tasks for robust operation. The techniques presented were integrated into a real-time robot simulation environment, with rigid body dynamics capabilities, used for development of complex robot applications.

Keywords: Robot simulation; Collision avoidance; Robustness; 3D scanning

1. INTRODUCTION

This paper presents a series of techniques for modelling a simulating a virtual robot environment, together with peripheral equipment (sensors, conveyor belt, grasping end-effector and 3D scanning sensor) using collision detection primitives and rigid body dynamics.

One good reason for developing a simulated robot environment is the limited support for collision detection offered by commercial robot controllers. In the present research, Adept Technology controllers and their programming environment V^+ allow the user to define maximum 4 obstacles of simple shapes: square, sphere, cylinder or frustum: (Adept Technology, Inc., 2004). Only the tool centre point is tested against these obstacles, and experiments showed that only the final destination of a motion instruction is actually checked. Therefore, if one would define a square obstacle between two points A and B, the robot would move from A to B passing through the obstacle without any warning.

A 6-d.o.f. robot had to be used in a 3D object scanning application, which employs automatic trajectory planning, optimized with respect to certain performance constraints. The trajectory planner had to know whether a certain robot configuration is collision-free or not, and the answer to this kind of queries should be given by the collision detection module of the simulation software.

Aside from being able to run robot programs in a simulated environment, it is also possible to use the developed application as a supervising module for real-time collision avoidance of robot applications outside the simulation. The geometry of the robot and workspace is either extracted from CAD data or acquired by 3D scanning.

A 3D scanning sensor is modelled either accurately, using ray tracing, or fast, using geometric intersections between rays and triangular meshes.

To extend the generality of the software, the following peripheral equipments are also modelled:

- part grasping is simulated with friction forces;
- presence and distance sensors are implemented with collision primitives;
- a conveyor belt is also implemented.

1.1 Motivation of the research

Using a simulation environment for developing complex robot programs has several advantages:

- There is no risk of damaging expensive equipment due to collisions. One of the applications of this simulation is automatic path planning in 3D scanning, and the laser probe used for scanning is an expensive device.
- The behaviour of the system can be analysed in ideal conditions. For the robot, this means ideal control loops, lack of vibrations and ideal kinematics computation. For the 3D sensor, there are no surface reflections, external light sources or sensor noise.
- The effect of known perturbations can be studied. These include misalignments between mechanical components, vibrations, and variations in robot parameters. A ray tracing simulation of the 3D sensor can model surface reflections and external lights.

There are also a number of disadvantages, the most important being the computational power involved for accurate simulations in real-time, the second one being the difficulty for simulating the less-than-ideal conditions of the real system.

Also, the robot simulator presented in this paper has also high educational value from two points of view:

- Every student can have access to a virtual robot and experiment his programs without the risk of damaging the equipment;

- Being an open source software, anybody can study the system, learn how it works, and improve it.

2. RELATED WORK

The problem of collision detection has been extensively studied, and many algorithms are available in the literature: there are general algorithms for dealing with arbitrary polygonal meshes with no particular structure (also called *polygon soups*), and particular algorithms, which exploit properties such as convexity or temporal coherence for faster queries.

Usually, the general notion of collision detection encompasses the following elements:

- *proximity detection*, which refers to the minimum distance between two solids
- *collision detection*, which detects whether two solid bodies touch each other
- *collision response*, which computes the changes in motion of the solid bodies after a collision

In 3D computer aided design (CAD), the following queries are usually performed:

- *clash* (intersection) detection: detect whether two bodies intersect each other
- *tolerance verification*: detecting whether two objects are closer than a given tolerance
- *distance computation*: computing the minimum distance between two objects

Traditionally, collision detection (CD) is discrete: it tests for overlapping between two static instances of moving objects; however, CD routines might ignore collisions between two fast moving objects (e.g., they may not notice a bullet passing through a narrow wall). In contrast, continuous collision detection techniques (CCD) are guaranteed to find the collision which has occurred between two given static instances of the 3D scene, although they require more processing power than CD. Recent research efforts concentrate on optimizing continuous collision detection, and also on applying it to deformable objects, which is useful for more realistic simulations.

Two older surveys are available in (Lin and Gottschalk, 1998) and (Jimnez et al., 2001); however, they only present non-continuous collision detection.

2.1 Software implementations of collision detection

There are two main classes of publicly available libraries which implement collision detection:

- rigid body dynamics engines, used in video games;
- standalone libraries for collision / proximity queries.

Rigid body dynamics simulation packages

There are numerous rigid body dynamics engines which implement state-of-art collision detection algorithms. They may be used only for collision queries, with some overhead.

Rigid body engines available under proprietary licenses include NVidia Physx (formerly known as AGEIA and Novodex), Intel HAVOK, Newton Game Dynamics and True Axis. There are also engines available under public licenses, such as BSD, ZLib and GPL, including Open Dynamics Engine, Bullet, JigLib and Tokamak. Most of the above engines can be wrapped in a unified abstraction system, like PAL, OPAL and GangstaWrapper. A comparison between engines supported by PAL is in (Boeing and Bräunl, 2007).

Libraries for discrete collision queries

Such libraries can be grouped in two classes:

a) Limited to convex polyhedra:

- GJK - Gilbert, Johnson and Keerthi distance routine; runs in expected constant time (Van den Bergen, 1999); implemented in Bullet Physics;
- I-COLLIDE: exact collision detection for large environments (Cohen et al., 1995); uses Lin-Canny Closest Features Algorithm; used in "Impulse" rigid body simulator (Mirtich, 1996);
- SWIFT: supports also bodies made of convex pieces. Possible queries: clash, distance and contact determination (Ehmann and Lin, 2000);

b) For arbitrary non-convex polyhedra (*polygon soups*):

- RAPID: uses OBBTree, a hierarchy of oriented bounding boxes (Gottschalk et al., 1996);
- PQP: intersection tests, distance query and tolerance verification (Larsen et al., 1999);
- V-COLLIDE: optimized for a large number of objects; uses 3 tests: *n-body*, OBB tree, and exact (Hudson et al., 1997);
- SWIFT++: for arbitrary polyhedral models. Queries: clash, tolerance, distance and contact determination (Ehmann and Lin, 2001);
- V-CLIP: Voronoi Clip algorithm; similar to Lin-Canny, less complex and more robust (Mirtich, 1998);
- OPCODE: memory-optimized AABB-tree (Terdiman, 2001);
- GIMPACT: Supports concave triangle meshes and deformable models. Implemented in ODE and Bullet.

Libraries for continuous collision detection (CCD)

- FAST: Performs CCD for general, rigid polyhedra (Zhang et al., 2006);
- CATCH: CCD for articulated models (Zhang et al., 2007); uses FAST for rigid body CCD, SWIFT++ for distance queries, and QHull for 3D convex hull.

3. ROBOT SIMULATION

For the robot arm, the simulation should be able to render it in any user-defined position. The user should be able to control either the joint angles for each articulation, or the Cartesian position together with the orientation given by ZYZ' Euler angles. Higher level positioning will also define the tool and base transformations.

3.1 Kinematic simulation

At the lowest level, 4×4 homogeneous transformation matrices are defined as primitive data types. They include translation and elementary rotations (around X, Y and Z). The next level include direct and inverse kinematics of the robot, which transform from a joint space configuration to a Cartesian position and vice-versa.

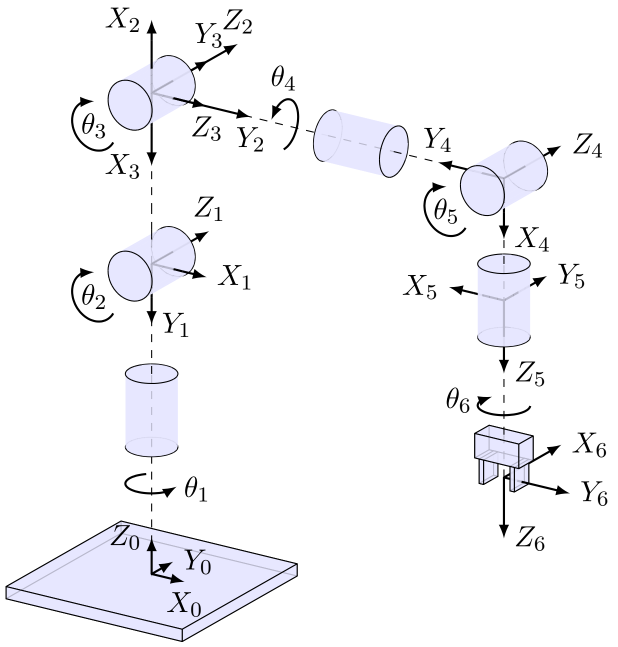


Fig. 1. Reference frames for 6-DOF robot arm

Table 1. Denavit-Hartenberg parameters for the 6-DOF robot

Link	a_i [mm]	d_i [mm]	α_i [deg]	$theta_i$ [deg]
1	75	335	-90	θ_1
2	270	0	0	θ_2
3	-90	0	90	θ_3
4	0	295	-90	θ_4
5	0	0	90	θ_5
6	0	80	0	θ_6

Direct and Inverse Kinematics The direct kinematics for the robot arm function is obtained by using the Denavit-Hartenberg (Spong et al., 2005) convention. The first step is to assign individual reference frames to each link from the kinematic chain, which includes the six robot arms and the laser probe (Fig. 1). The direct kinematics function is the product of the individual homogeneous transformations (Davis, 2001) for each robot link $i = \overline{1..n}$. An individual matrix, called T_i^{i-1}, is the transformation from the $(i-1)^{th}$ link reference frame to the i^{th} link reference frame. The 0^{th} link is the robot base, and the last link is the end effector. Knowing the Denavit-Hartenberg parameters a_i, d_i, α_i and θ_i for each joint $i = \overline{1..n}$, with θ_i being the joint variables, the individual transformations T_i^{i-1} can be written:

$$T_i^{i-1} = \mathcal{R}_Z(\theta_i)\ \mathcal{T}(a_i, 0, d_i)\ \mathcal{R}_X(\alpha_i) \tag{1}$$

where $\mathcal{T}(x, y, z)$ is the homogeneous translation, and $\mathcal{R}_A(\phi)$ is the homogeneous rotation around axis A with angle ϕ.

The direct kinematics transforms are 4×4 matrices:

$$T_{DK} = T_6^0 = T_1^0\ T_2^1\ T_3^2\ T_4^3\ T_5^4\ T_6^5 \tag{2}$$

The World coordinate system used here, $X_0Y_0Z_0$, is right-handed, with the X_0Y_0 plane being horizontal, X_0 axis pointing forward, Z_0 axis pointing upwards, and the origin being at the base of the robotic arm.

3.2 3D rendering

For 3D rendering, the relative transformations for each link i with respect to its parent link $(i-1)$ have to be known, and they are given by Eq. 1 when the individual joint angles θ_i are known. This requires every geometric link to have its reference frame assigned according to the Denavit-Hartenberg convention.

For rendering the mesh of link i, the transformation to be passed to OpenGL is:

$$T_{GL}(i) = T_i^0 = T_1^0\ T_2^1 \cdots\ T_i^{i-1} \tag{3}$$

The meshes used for rendering (Fig 2) were imported from the CAD files, available on the manufacturer's web site, and were simplified to 5000 faces per mesh using MeshLab, with the function *Quadric Edge Collapse Decimation* (Cignoni et al., 2008).

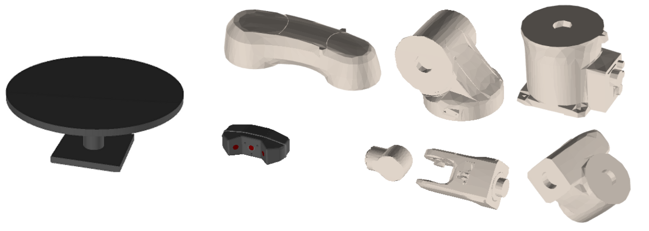


Fig. 2. Geometric models of the robot links, rotary table and the 3D scanning sensor

3.3 Grasping simulation

A useful simulation should have a high degree of generality, and should not be limited to a single application. A simulation which models correctly the interaction between the robot and its environment even when there are programming mistakes is more useful because it lets the user to debug its application inside the virtual environment, without risking to damage expensive equipment.

With grasping simulation, the robot can interact with objects in the virtual environment according to physics laws, i.e. by means of contact forces and friction. Therefore, a good grasping simulation can show the following effects:

- Contact slippage due to high accelerations in robot motion and inertial movements;
- Effects of grasping on soft bodies;
- Instability when only two contact points are present.

When implementing grasping simulation, it is compulsory that the robot moves on a smooth trajectory, with acceleration and deceleration profiles. A naive implementation, which would simply enforce the position of each link according to the kinematics model, will render correctly the robot, but will fail to keep the grasped part in the gripper due to very high accelerations that are necessary to move the robot suddenly in a single time step.

For a two-fingered gripper, two forces are necessary to keep the gripper closed and holding the object, and a third one to keep the gripper fingers centred with the end-effector axis. The implementation uses two slider joints, each joint connecting the finger and the gripper base.

4. COLLISION HANDLING TECHNIQUES

4.1 Collision detection

In the robot simulation presented here, collision detection is employed in two situations:

- between the robot links themselves;
- between the robot and other objects.

The first situation has to make sure that the robot trajectory would not cause collisions between the robot links themselves. For 6-DOF robots, it is easy to program a trajectory which will collide, for example, the end effector with the first link.

When performing queries to the collision library, the pairs of consecutive robot links should be excluded, since their geometries are always in contact. Performing collision queries between them will only return superfluous results and slow down the simulation.

4.2 Collision avoidance

This step is performed by a trajectory planner, which knows the initial state, the final state, and can perform collision queries for intermediate states.

The simplest robot programs do not include collision avoidance; they simply move the robot between predefined or computed locations, without any validation. They are usually not robust, since they are based on the assumption that the trajectories are computed in such a way that collisions are not possible.

4.3 Collision response

This step is only a visual feedback which shows that a collision happened. In rigid body dynamics engines, contact forces or impulses are applied to the simulated bodies in order to simulate the effect of collisions.

The collision detection module will return, for a pair of bodies, a set of N contact points. For simple models like sphere-plane, only a contact point is returned. For box-plane, there are usually 4 contact points. However, for high resolution meshes, the collision detection routine may report a large number of contact points (maybe hundreds).

In ODE, there are two possible solvers for advancing the simulation in time: `Step` and `QuickStep`. These solvers take into account the joints between bodies (e.g. the robot links) and the contact joints which appear only when two bodies collide. The method `Step` is the most accurate method according to Boeing and Bräunl (2007), but its time complexity is $O(m^3)$ where m is the number of constraints. `QuickStep` is an iterative method with $O(m*n)$ complexity (n is the number of iterations per time step), and is comparable with the solvers used currently in video games; however, experiments showed it is not accurate enough for simulated the grasping process and it was unstable with vertical stacks of boxes.

For a fluent simulation, the number of contacts has to be reduced, since the $O(m^3)$ complexity would not be adequate with hundreds of contact points. In areas when accuracy is important (e.g. grasping simulation), there should be enough contacts; for a contact between two parallel boxes, good results are obtained with 4 contact points. However, when the robot is colliding with the floor, the accuracy of the collision response is not important, and only one contact point is sufficient.

5. PERIPHERAL DEVICE SIMULATION

5.1 Presence sensor

The presence sensor is a device with Boolean output which is activated when another part is in the sensor proximity. The naive implementation simply checks if the position of all scene objects is in a desired interval. However, this approach will not work correctly when it has to detect parts of different sizes, or parts with non-trivial geometry.

An implementation using a rigid dynamics package can model the sensitive volume of the sensor as a solid, allow it to collide with the objects of interest, and ignore the collision response step, letting the object and the sensor geometries to penetrate each other.

5.2 Distance sensor

Optical distance sensors, which use triangulation, can be simulated using *ray geometries*. The implementation will check their intersections with all the possible objects of interest. For more realistic simulations in concave areas, one would create an extra ray, coming from the reflection point through the sensor's lens. If the second ray would hit another object, the measurement is discarded.

5.3 3D scanning sensor simulation

The 3D scanning sensor, also known as the *profile scanner*, projects a narrow stripe of laser light onto the surface being digitized. A 2D camera, placed at a known angle with respect to the laser plane, records the image of the laser stripe and computes the local geometrical shape of the surface. For better accuracy and improved visibility in concave regions, two or more cameras may be used with the same laser beam.

For reconstructing a full 3D model of the workpiece, the profile scanner has to be swept around the part. The most precise way is to use a coordinate measuring machine (CMM). The sensors can also be mounted on robot arms, which are more flexible in positioning and orienting the sensor, but also less accurate.

There are two methods for 3D sensor simulation:

- *Ray tracing simulation*, which creates the image as seen by the camera, and computes the sensor output with an image processing algorithm, which detects the laser stripe and computes 3D coordinates with non-linear equations. This simulation was presented in detail in (Borangiu et al., 2008b), and can be used for testing the image processing algorithms in ideal conditions or controlled perturbations.
- *Geometric simulation*, which computes the intersection between the sensor's field of view and the objects on the scene. It can be two orders of magnitude faster, therefore suitable for realtime simulations.

5.4 Ray tracing simulation of 3D scanning sensor

The sensor was modelled with POV-Ray, which uses a *Scene Description Language* for describing the objects, lights and cameras that interact in a virtual environment by means of ASCII-based input files. This capability allows easy interfacing with many programming environments.

Camera modelling and triangulation

The laser probe emits a laser beam focused into a plane, such as when it intersects a surface, it casts a line or a curve. This laser beam is approximated with a point light source, constrained to pass to a narrow opening.

The cameras used in the laser probe are modelled as two standard perspective cameras which may be implemented in POV-Ray by entering their parameters such as position, orientation and focal length. In the following text, only one of the two cameras will be described, as the other one is identical and symmetrical to the first one.

Let XYZ be the reference frame of the laser probe as in Fig. 3(b), and let xyz be the reference frame of the CCD array from the camera (Fig. 3(a) and 3(b)). Referring to Fig. 3(b), the camera position and orientation with respect to the laser device are given by the scalars a, b and ϕ.

Using these notations, let $P = (P_X; P_Y; P_Z)$ the point of reflection of a laser ray in the XYZ reference frame, and let $p = (p_x; p_y)$ be the coordinate of the pixel at which the ray was detected on the CCD matrix, in xy reference frame. Knowing the pixel coordinates p, the location of the 3D point P can be expressed using the triangulation equations (4):

$$\begin{aligned} P_X &= 0 \\ P_Y &= \frac{a}{f\ \sin\left(\phi - \arctan\dfrac{p_y}{f}\right)}\, p_x \\ P_Z &= \frac{a}{f\ \tan\left(\phi - \arctan\dfrac{p_y}{f}\right)} + b \end{aligned} \tag{4}$$

where$f = \dfrac{H}{2\tan\gamma}$ if the unit length is considered to be 1 pixel, i.e. the distance between two adjacent pixels on the CCD array.

The area in the plane determined by the laser rays, i.e. YZ plane in Fig. 3(c), is a trapezoid determined by z_{min}, z_{max}, y_{min} and y_{max}, whose expressions are given in Eq. 5. The scanning range is thus given by z_{min} and z_{max}, and the length of the laser line L_L that is effectively being analyzed is dependent of the distance of the scanned object with respect to the laser probe, and varies from $L_{Lmin} = 2\ y_{min}$ when the workpiece is close to the probe, to $L_{Lmax} = 2\ y_{max}$ when the workpiece is far from the laser probe.

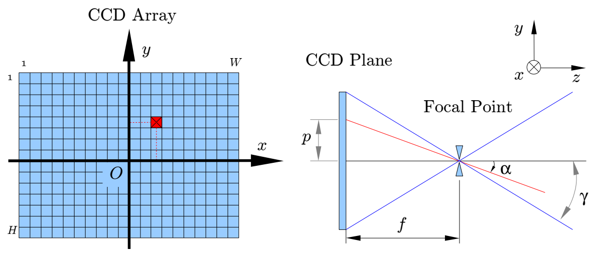
$$\begin{aligned} y_{min} &= \frac{a\ \tan\gamma}{\sin(\phi+\gamma)} & z_{min} &= \frac{a}{\tan(\phi+\gamma)} + b \\ y_{max} &= \frac{a\ \tan\gamma}{\sin(\phi-\gamma)} & z_{max} &= \frac{a}{\tan(\phi-\gamma)} + b \end{aligned} \tag{5}$$

5.5 Geometric simulation of 3D scanning sensor

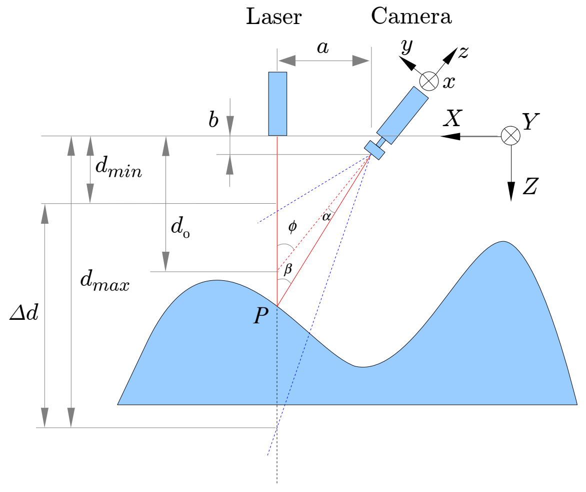
A much faster alternative for simulating the 3D scanning sensor is to use a number of rays which cover the entire field of view of the sensor, and which will intersect with the scanned geometry. The simulation only has to compute the intersection points between each ray and the rigid bodies on the scene, and choose for each ray the point which is closest to the origin of the laser source (Fig. 4 a).

The simulation will perform queries for intersection between rays and other solid geometry (sphere, box, triangle mesh) using the collision detection library.

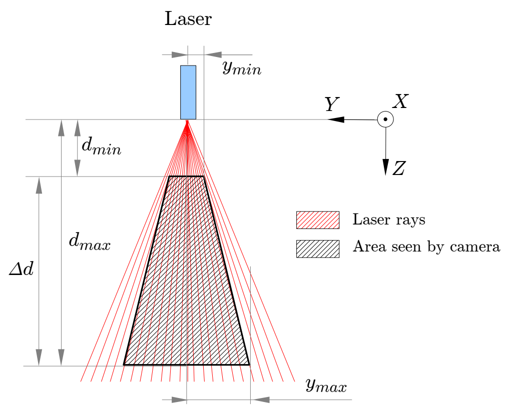
The 3D data obtained by the laser sensor is expressed relative to a local reference frame on the laser sensor,



(a) CCD sensor and its reference frame



(b) Side view of the laser probe



(c) Front view of the laser probe

Fig. 3. Triangulation process for 3D reconstruction

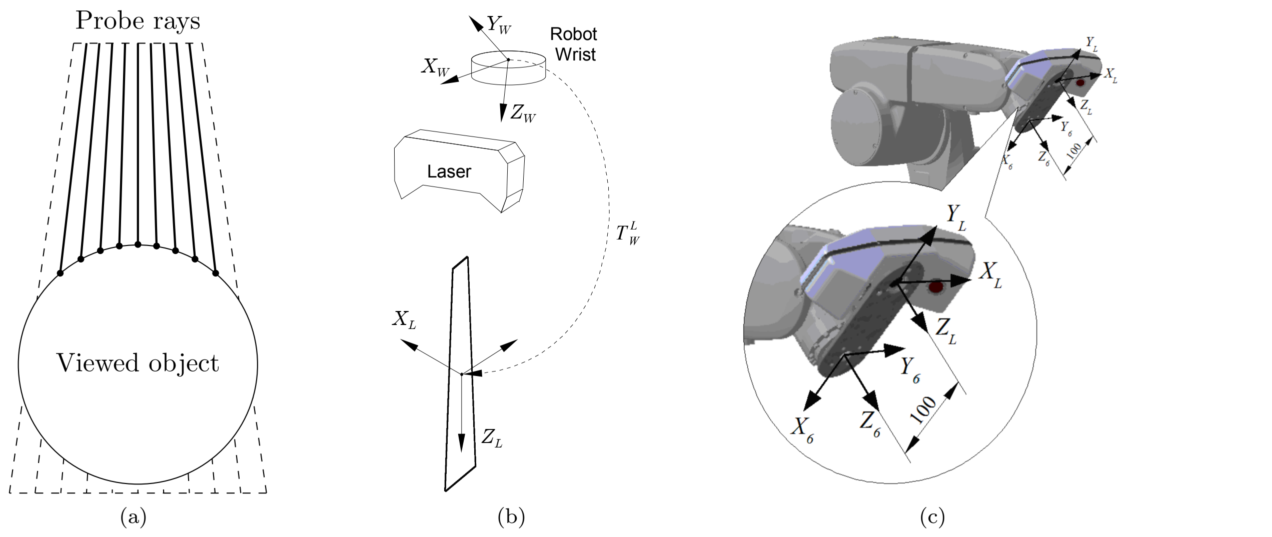


Fig. 4. (a) Geometric simulation of 3D scanning sensor; (b) Tool transformation for aligning the sensor data; (c) Ideal position of laser sensor with respect to the robot end effector

X_L Y_L Z_L (Fig. 4 b). However, for surface reconstruction and 3D robot guidance tasks, the sensor data has to be expressed in a reference frame attached to the robot. This can be achieved by pre-multiplying the 3D scanner data with the transformation matrix representing the position of the robot at the moment of data acquisition, taking also into account the relative position of the sensor with respect to the robot tool centre point (TCP).

The position of the sensor with respect to robot TCP is modelled using a *Tool transformation* (Fig. 4 b), whose ideal value can be computed from the dimensions of the mechanical fixture (Fig 4 c).

In the physical 3D scanning system, the tool transformation encompasses the misalignment between the sensor and the robot, and is determined using a calibration procedure (Borangiu et al., 2009a). Also, the laser measurements are synchronized with the robot position by means of a trigger signal, therefore the sensor can acquire data while the robot is moving. This operation mode is called *dynamic scanning*. The other mode is *stop-and-look*, where the sensor is only allowed to take measurements when the robot does not move. This mode is much slower, but the measurements are more accurate.

5.6 Rotary table

The table can be simulated with a cylindrical body attached to a hinge joint. Objects are attached to the table using friction forces, and this behaviour is already implemented in the rigid body simulation engines, without requiring extra effort from the application developer.

For simulating the calibration algorithms for the laser sensor, the table can be placed in the virtual scene with a known eccentricity, or with a known angular deviation from the ideal rotation axis. This will model the lack of accuracy in low-cost positioning devices and are helpful to test the ability of the calibration algorithms to compensate for these inaccuracies. Calibration algorithms for the 3D sensor and eccentric rotary table were presented in (Borangiu et al., 2008a).

5.7 Conveyor belt

While a conveyor belt can be simulated as a set of small elements moving on a closed curve, and using friction for interacting with transported objects, this is very expensive from a computationally point of view. A simpler implementation may use a stationary box object with a moving surface, by setting a special flag for the contact joints between the belt and the other bodies. This approach still models friction between bodies and conveyor belt, and no special care should be taken in order to keep the belt moving continuously.

For rendering the conveyor belt, a static texture can be used, and change the only texture coordinates within OpenGL drawing routine. This approach is fast because the texture is sent to the graphics card only once, and inside the rendering loop, only the transformation matrix is updated.

6. REAL TIME SIMULATION ISSUES

This section presents the bottlenecks which could affect real-time operation, and solutions for overcoming them.

6.1 Programming language overhead

The simulation presented in this work was implemented using Python scripting language, which is interpreted and dynamically typed. While being a flexible language with a concise syntax, every variable access generates a lookup in the dictionary of names, and this results in a large speed penalty. For a real-time system, bottlenecks can be rewritten either in C/C++, or can be implemented in Cython using static typing in the innermost loops. With Cython, very small changes to Python code can result in performance equal to the C equivalent of the same algorithm (Seljebotn, 2009).

6.2 *Network delay*

The network delay represents a perturbation in the communication between the robot and PC workstation, when using the latter as a watchdog for avoiding collisions.

Robot position query can be performed from PC terminal every 32 milliseconds via Ethernet. A major cycle of the robot controller has 16 milliseconds, therefore a position query takes two robot cycles in the most favourable case.

However, when network delays occur, the PC would not have the possibility to slow down the robot quickly enough. Therefore, a protection mechanism has been employed: if the time since last message received from the watchdog is higher than normal, the monitor speed on robot controller decreases gradually, even until a full stop if network delays are very high. However, the method is not yet reliable for high speed operations, and this will addressed in future work.

6.3 *Collision detection speed*

It may seem surprising that the collision itself is not actually a bottleneck in the current simulation, even if each mesh has 5000 triangles on average. The collision detection engine (OPCODE), as implemented in the ODE library, uses a memory optimized AABB tree and is able to run the collision queries for the entire scene in less than 10 milliseconds on a Core2Duo CPU, therefore being able to achieve 100 frames/second without considering the 3D rendering, dynamics simulation, motion planning and user interface tasks. The entire simulation runs at 30 frames/second on the same computer.

7. APPLICATIONS

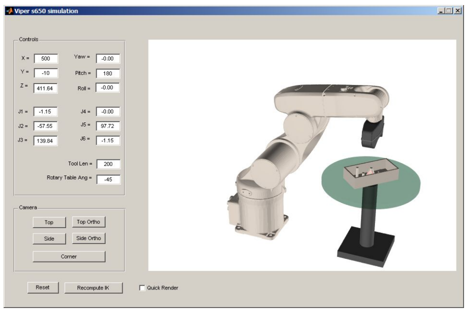
This section presents applications developed with the help of collision detection methods from in the simulation engine presented in this paper.

7.1 *Simulation of 3D scanning process*

The screen shot of the simulation software is displayed in Fig. 5(a). The user can control either the position and orientation of the robot in Cartesian mode, relative to either robot base or rotary table, or the individual joint angles. The software may simulate a continuous movement of the laser probe over the workpiece, compute the images that would be seen by the two cameras, and generate a point cloud from processing them (Fig. 5(b)-(d)).

The simulator has two modes of operation: static and dynamic simulation. In the *static* mode, the robot maintains the position of the laser sensor fixed, and the two CCD image sensors show a simulated image. In *dynamic* mode, the user can specify complex scanning trajectories which will be followed. The result from the laser sensor is analyzed and a point cloud model, representing the scanned virtual part, is created.

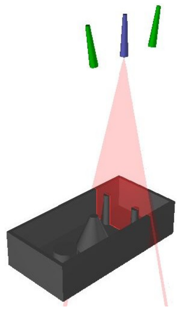
The simulator is also capable of exporting animations with the scanning system following a predefined program.

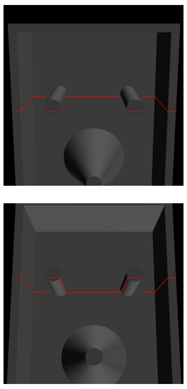

Viper s650 simulation

Controls

X = 500
Y = -10
Z = 411.64
Yaw = -0.00
Pitch = 180
Roll = -0.00
J1 = -1.15
J2 = -57.55
J3 = 139.84
J4 = -0.00
J5 = 97.72
J6 = -1.15
Tool Len = 200
Rotary Table Ang = -45

(a) Screen shot of the simulator


(b) Laser probe


(c) Images from the two cameras

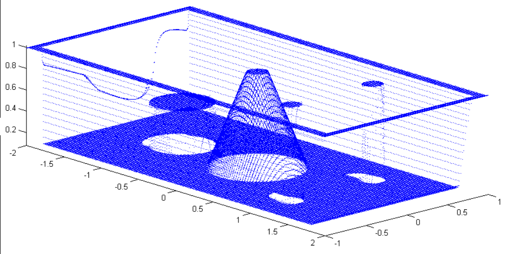

(d) Computed point cloud

Fig. 5. Laser probe simulator

7.2 *Collision-free motion planning for 3D scanning*

A heuristic planning algorithm for the robot arm and rotary table was presented in Borangiu et al. (2009b). The 7-DOF mechanism (robot + table) is redundant, and this property can be exploited to satisfy additional constraints. The rotary table angle, θ_R, may be chosen freely; once this angle is fixed, the remaining 6-DOF can be uniquely chosen from the possible inverse kinematics solutions.

The algorithm supports additional constraints specified as real-valued functions f_i, with $0 \leq f \leq 1$, which evaluate any static robot pose, with the following meaning:

- $f_i = 0$: the constraint is not satisfied;
- $f_i = 1$: the constraint is fully satisfied;
- $0 < f_i < 1$: the constraint is only partly satisfied.

If there are many constraints, their functions can be multiplied, resulting a metric for evaluating any static robot configuration, with the same interpretation:

$$f = \prod_{i=1}^{m} f_i \tag{6}$$

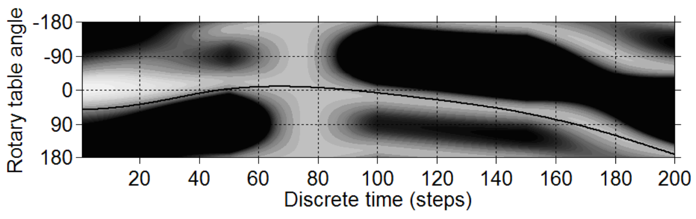
where m is the number of constraints.

If only one constraint is not satisfied ($f_i = 0$), the specific robot configuration is avoided, since $f = 0$.

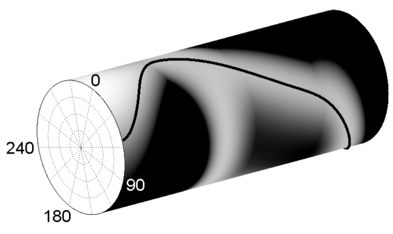
A set of constraints which keeps the robot away from its joint limits, ensuring a natural configuration, is:

$$f_j^*(\theta_j) = \left(\sin \left(\frac{\theta_j - \theta_j^{\min}}{\theta_j^{\max} - \theta_j^{\min}} \cdot \pi \right) \right)^{\gamma_j} \tag{7}$$

where $1 \leq j \leq 6$ is the joint number.



(a)



(b)

Fig. 6. Grayscale configuration map for static constraints: (a) plane view; (b) cylindrical view

Static constraints can be represented graphically as graylevel configuration maps, where the pixel intensity is the value of the metric function (6). An example is given in Fig. 6 (a) and (b), where the two dimensions of the map are the rotary angle θ_R and the discrete time t. The planner attempts to find a path from a start configuration (leftmost column in the map) to a final configuration (rightmost column in the map) which, while obeying all the constraints at least partly, has to be as smooth at possible, and should not exceed the maximum angular speed and acceleration values for the rotary table. The robot motion is not constrained explicitely, but high speeds in robot motions can be avoided by adding static constraints which do not allow the robot to reach singular configurations.

Dynamic constraints for the rotary table are specified using scalar weights for angular speed and acceleration, k_ω and k_a. The heuristic algorithm, called *Ray Shooting*, attempts to try various constant-acceleration paths (rays) and selects the lowest cost path at every time step. This strategy ensures low variations in the angular speed of the rotary table, which allows scanned parts to sit on the table without any additional fixture. The planning algorithm also attempts to reduce scanning time and increase scanning accuracy by avoiding near-singular configurations.

A constraint suitable for avoiding collision detection depends on the *minimal distance* between two rigid bodies, d_{min}. If d_{min} is less than a threshold d_{min}^{low}, the constraint is not satisfied, and this configuration is forbidden. If d_{min} is higher than d_{min}^{high}, the constraint is fully satisfied:

$$f_C(d_{min}) = \begin{cases} 0, d_{min} < d_{min}^{low} \\ sin\left(\dfrac{d_{min} - d_{min}^{low}}{d_{min}^{high} - d_{min}^{low}} \cdot \pi\right)^{\gamma_C} \\ 1, d_{min} > d_{min}^{high} \end{cases} \quad (8)$$

The exponent γ_C controls the constraint intensity for d_{min} between $[d_{min}^{low}...d_{min}^{high}]$: higher values rejects values closer to d_{min}^{low}, while lower values are more permissive.

The parameters d_{min}^{low} and d_{min}^{high} can be chosen the same for every pair of possibly colliding bodies or can be adjusted for each pair. For example, the distance between the laser sensor and 4th robot link is by design 10 milimeters, so there's no point in setting a higher value for d_{min}^{high}. However, if one has to keep the distance between the laser sensor and the rotary table at least 20 mm, and preferably 50 mm, then d_{min}^{low} and d_{min}^{high} should be set to 20 and 50 respectively.

7.3 Real-time collision avoidance in robot tasks

In most robotic tasks, the robot is operated in two modes:

- From the manual control pendant (MCP) of the robot
- In automatic mode, where scanning trajectories are generated by the control software

In manual mode, the robot is usually moving at low speeds and the user is assumed to be careful not to cause collisions. However, a robust user interface shouldn't rely on correct user input; it should not allow the user to produce damage to the system no matter what the user input might be.

In automatic mode, the robot moves along a programmed trajectory, which is computed from user-input data, or from parameters computed automatically using sensors or vision equipment. However, users may make mistakes, and autodetection may produce incorrect results.

Collision detection during manual operation

The proposed protection scheme (Fig. 7) is to have a dedicated task for realtime collision checking during robot operation. The protection task runs on the PC, continuously monitoring the robot position and velocity.

In manual mode, no user program is allowed to move the robot or change its speed, due to internal protection mechanisms implemented in the robot controller.

When the robot is heading to a colliding situation, the only actions that could be taken from a user program are:

- Give visual feedback on the MCP;
- Give audible feedback to the user;
- Assert the emergency stop signal (in extreme cases).

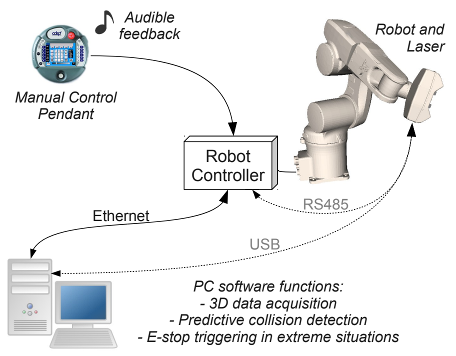


Fig. 7. Protection scheme for collision-free manual mode operation

The first option is useless if the user is not looking at the MCP. The second option uses the control pendant's internal speaker to emit warning beeps. If a collision is imminent, the only action allowed by the robot controller is to assert the emergency stop signal.

A more useful approach would have been to automatically reduce the robot speed when a collision is imminent; however, in order to implement it, safety mechanisms from the robot controller would have to be bypassed.

Warnings should be given only when two possibly colliding bodies become closer. When the user moves the robot away from the colliding situation, no warning should be given.

To implement this, the protection module should know also in which direction the robot is moving. User-level programs do not have access to the buttons pressed on the control pendant. However, in manual mode it is relatively easy to predict the robot motion, since the robot can be moved using one of the following motion types:

- Cartesian translation (any direction in 3D space);
- End-effector rotation (around any fixed axis);
- Joint motion (rotate only one robot joint at a time).

Therefore, a predictive collision detection mechanism can be used. The predictor has to detect the motion type:

- A joint interpolated motion;
- A Cartesian motion (translation and/or rotation).

A model for describing and predicting joint-interpolated motions is given by:

$$J_i^{(t+1)} = J_i^{(t)} + s\ \delta j_i, \quad i = \overline{1,n} \tag{9}$$

where the robot has n independent joints, $J_i^{(t)}$ is the absolute position of i^{th} joint at time t, δj_i are weights which represent the relative speed of the i^{th} joint and s is the speed factor. This model represents general joint interpolated motions; however, in manual mode, only one joint is moving at a time.

Therefore, the model parameters are $[\delta j_i]$, $i = \overline{1,n}$. They remain constant during the motion and can be identified by linear regression. In contrast, s may change freely throughout the motion, due to acceleration.

Cartesian motions are described by linear interpolation in X, Y and Z, and spherical linear interpolations (slerp) in orientation. Therefore, the rotation axis remains constant throughout the Cartesian motion. A model for predicting linear motions in Cartesian space, where the end-effector is allowed to change its orientation, is:

$$\begin{aligned} X^{(t+1)} &= X^{(t)} + s\ \delta x \\ Y^{(t+1)} &= Y^{(t)} + s\ \delta y \\ Z^{(t+1)} &= Z^{(t)} + s\ \delta z \\ \theta^{(t+1)} &= \theta^{(t)} + s\ \delta\theta \\ R^{(t)} &= \mathcal{R}_{[rx,ry,rz]}(\theta^{(t)}) \cdot R^{(0)} \end{aligned} \tag{10}$$

where (X, Y, Z, R) is the Cartesian end-effector position and orientation (R is a 3×3 rotation matrix), $R^{(0)}$ is the initial end-effector orientation (at $t = 0$), and $[rx, ry, rz]$ is the rotation axis throughout the motion, which is constant. The notation $\mathcal{R}_{[axis]}(angle)$ is a rotation matrix specified by its axis and angle.

The model parameters, which remain constant throughout the motion, are $[\delta x, \delta y, \delta z, \delta\theta, rx, ry, rz]$. These parameters remain constant throughout the motion and can be identified by nonlinear minimization.

The decision for the motion type (Cartesian or joint) is taken by trying to fit both models and select the one which gives lower residuals.

Transformations between Cartesian and joint spaces are given by direct and inverse kinematics functions.

Collision detection during automatic operation

Even if the trajectory planner is programmed to generate collision-free paths, nobody can be certain that there are no programming mistakes in the robot software. In semi-automatic modes, trajectory planning is performed solely on the robot controller, which does not check for collisions; however, collision checking can be done before sending a motion instruction to the robot. Of course, this assumes the robot program runs from the PC terminal.

The collision detection mechanism described in this section is designed to be as general as possible, in order to be useful regardless of the particular robot application. The implementation is a watchdog task, which analyzes the subsequent motion transparently, while the program is running. If a collision becomes imminent, the following actions can be taken:

- User feedback (visual or auditive);
- Gradually reduce monitor speed [1] (this can be performed even while another program is running);
- Trigger the emergency stop (only in extreme cases).

In automatic operation, only one program is normally allowed to *move* the robot. However, there may be additional program tasks which can *watch* the robot motion, i.e. read the current position within a loop, and also retrieve the destination of the current motion. Therefore, the watchdog task knows in advance the robot trajectory, and no prediction is necessary.

A schematic view of the protection scheme for automatic operation is given in Fig. 8.

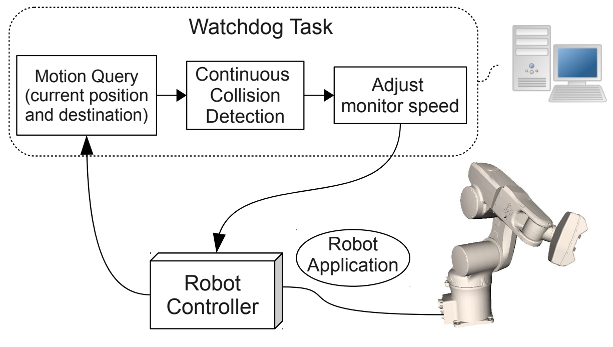

Watchdog Task
Motion Query (current position and destination)
Continuous Collision Detection
Adjust monitor speed
Robot Application
Robot Controller

Fig. 8. Protection scheme for collision-free automatic mode operation

[1] Monitor speed is a global setting of the robot, regardless of program speed

8. CONCLUSION

This paper presented a set of robot modelling and simulation techniques focused on collision queries, which are integrated in a multi-device virtual workspace for simulating robot and 3D scanning applications. A 6-DOF arm is modelled at a kinematics level and rendered using OpenGL, and interactions between the robot and its environment are simulated using rigid body dynamics. Presence and distance sensors are implemented using collision queries between their active space and the scene objects. A 3D scanning sensor is simulated either with ray tracing (accurate, but slow) or with geometric primitives (real-time ideal simulation).

The following applications were developed with the help of the simulation software which implements the discussed collision handling and simulation techniques:

- Simulation of 3D scanning process
- Collision-free path planning for redundant mechanisms, in particular, a 7-DOF system for 3D scanning
- Collision avoidance in real-time robot applications

ACKNOWLEDGEMENTS

This work is funded by the National Council for Scientific University Research, in the framework of the National Plan for Research, Development and Innovation, grant 69/2007.

REFERENCES

Adept Technology, Inc. (2004). Adept SmartMotion Developer's Guide.

Boeing, A. and Bräunl, T. (2007). Evaluation of real-time physics simulation systems. In *GRAPHITE '07: Proceedings of the* 5th *international Conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, 281–288. ACM, New York, NY, USA.

Borangiu, T., Dogar, A., and Dumitrache, A. (2008a). Integrating a short range laser probe with a 6-dof vertical robot arm and a rotary table. In *RAAD 2008 - The 17th International Workshop on Robotics in Alpe-Adria-Danube Region.* Ancona, Italy.

Borangiu, T., Dogar, A., and Dumitrache, A. (2008b). Modelling and simulation of short range 3D triangulation-based laser scanning system. *Int. Journal of Computers, Communications and Control (IJCCC)*, 3(Suppl. Issue - ICCCC'08), 190–195.

Borangiu, T., Dogar, A., and Dumitrache, A. (2009a). Calibration of wrist-mounted profile laser scanning probe using a tool transformation approach. In *RAAD 2009 - The 18th International Workshop on Robotics in Alpe-Adria-Danube Region.* Brasov, Romania.

Borangiu, T., Dogar, A., and Dumitrache, A. (2009b). A heuristic approach for constrained real time motion planning of a redundant 7-dof mechanism for 3D laser scanning. In *INCOM 2009 - 13th IFAC Symposium on Information Control Problems in Manufacturing.* Moscow, Russia.

Cignoni, P., Corsini, M., and Ranzuglia, G. (2008). Meshlab: an open-source 3D mesh processing system. *ERCIM News*, (73), 45–46.

Cohen, J.D., Lin, M.C., Manocha, D., and Ponamgi, M. (1995). I-COLLIDE: an interactive and exact collision detection system for large-scale environments. In *I3D'95: Proc. of the 1995 Symp. on Interactive 3D graphics*, 189. ACM, New York, USA.

Davis, T. (2001). Homogeneous coordinates and computer graphics. URL `www.geometer.org/mathcircles/`.

Ehmann, S.A. and Lin, M.C. (2000). Accelerated proximity queries between convex polyhedra by multi-level Voronoi marching. In *Proc. of IEEE/RSJ International Conf. on Intelligent Robots and Systems*, pp.2101–2106.

Ehmann, S.A. and Lin, M.C. (2001). Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *in Computer Graphics Forum*, pp.500–510.

Gottschalk, S., Lin, M.C., and Manocha, D. (1996). OBB-Tree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the* 23rd *annual Conference on Computer graphics and interactive techniques*, 171–180. ACM, New York, NY, USA.

Hudson, T.C., Lin, M.C., Cohen, J., Gottschalk, S., and Manocha, D. (1997). V-COLLIDE: accelerated collision detection for VRML. In *VRML '97: Proc. of the second Symp. on Virtual reality modeling language*, 117–ff. ACM, New York, NY, USA.

Jimnez, P., Thomas, F., and Torras, C. (2001). 3D collision detection: a survey. *Computers and Graphics*, 25(2), 269–285.

Larsen, E., Gottschalk, S., Lin, M.C., and Manocha, D. (1999). Fast proximity queries with swept sphere volumes. Technical report, Department of Computer Science, University North Carolina at Chapel Hill, USA.

Lin, M.C. and Gottschalk, S. (1998). Collision detection between geometric models: A survey. In *In Proc. of IMA Conference on Mathematics of Surfaces*, 37–56.

Mirtich, B. (1998). V-Clip: fast and robust polyhedral collision detection. *ACM Trans. Graph.*, 17(3), pp.177–208.

Mirtich, B.V. (1996). *Impulse-based dynamic simulation of rigid body systems.* Ph.D. thesis, University of California, Berkeley.

Seljebotn, D.S. (2009). Fast numerical computations with cython. In *Proceedings of the 8th Python in Science Conference (SciPy 2009)*, pp. 15–23. Pasadena, CA, USA.

Spong, M.W., Hutchinson, S., and Vidyasagar, M. (2005). *Robot Modeling and Control.* John Wiley and Sons, Inc., New York.

Terdiman, P. (2001). Memory-optimized bounding-volume hierarchies.

Van den Bergen, G. (1999). A fast and robust GJK implementation for collision detection of convex objects. *J. Graph. Tools*, 4(2), pp.7–25.

Zhang, X., Lee, M., and Kim, Y.J. (2006). Interactive continuous collision detection for non-convex polyhedra. *Vis. Comput.*, 22(9), 749–760.

Zhang, X., Redon, S., Lee, M., and Kim, Y.J. (2007). Continuous collision detection for articulated models using Taylor models and temporal culling. *Proc. of SIGGRAPH 2007*, 26(3), 15.