Design and Implementation of a Graphic 3D Simulator for the Study of Control Techniques Applied to Cooperative Robots

Article *in* International Journal of Control Automation and Systems · December 2015

DOI: 10.1007/s12555-014-0278-y

International Journal of Control, Automation, and Systems (2015) 13(6):1476-1485 ISSN:1598-6446 eISSN:2005-4092
DOI 10.1007/s12555-014-0278-y http://www.springer.com/12555

Design and Implementation of a Graphic 3D Simulator for the Study of Control Techniques Applied to Cooperative Robots

Claudio Urrea* and Jean Paul Coltters

Abstract: A new methodology is proposed for the design and implementation of three-dimensional simulations of manipulator robots, in individual as well as cooperative tasks, using several programming and 3D-design tools. To obtain easily the dynamic-mathematical models of movement for the represented systems, regardless of the kind of manipulator robot to be considered, the development of a computer algorithm is also presented. Besides, through the use of the Bullet library, we incorporated the detection of collisions between constitutive elements of the system, considering elements external to the robots, collisions between links of the same robot, or between two robots. Using a C++ programming platform, we also designed and implemented a flexible and intuitive graphic user interface. The dynamic-mathematical performance of the cooperative interaction between two actual robots, designed and implemented in the Departamento de Ingeniería Eléctrica of the Universidad de Santiago de Chile (DIE-UdeSantiago de Chile), was modeled, plotted and three-dimensionally simulated, by using the new algorithm proposed for this purpose. Finally, the performance results are presented and analyzed.

Keywords: 3D, graphic interface, graphic simulators, manipulators, SCARA robot.

1. INTRODUCTION

At present, robots have features that allow them to perform their tasks quickly and precisely, and with a high repeatability. Manipulator robots have become an ever-increasing common tool in the industry [1]. Moreover, when robots work in a cooperative way, they acquire the ability to perform tasks that otherwise would be impossible to be executed individually [2]. All these technological implementations are the result of a continuous scientific effort, the fruit of years of research in the fields of design and construction of such robots, including the development of many control systems for those machines, which day by day have become more sophisticated.

A large part of this scientific research is supported by computer simulations, powered by the steady increase of the processing capabilities of PCs and the introduction of new tools and platforms in the field of computer science. Computer simulations are an essential tool for the observation of dynamic behaviors, tuning controllers, and other purposes aimed to assist the decision making process in many disciplines, allowing multiple tests of actual systems to be carried out with ever better approximation, allowing both industry and researchers to make significant savings in the expenses involved in having real systems subjected to testing.

There are some applications like MATLAB that have very powerful simulation tools, besides graphic tools like "Simulink 3D Animation" that even allow showing three-dimensional models created by programs like Blender and relating them effectively with variables of MATLAB, either from WorkSpace or Simulink, to be plotted in specific positions and with specific rotations. However, usually the ease of developing a specific application in a high-level language implies a lower computing performance and less flexibility for the programmer. This makes tasks like simultaneous simulations of objects become complex issues, having, e.g., to multiply N times the work diagram or the code lines.

Prior to the work presented in this paper, the implementation of the kinematics of a hexapod robot [3] using the MATLAB/Simulink software was studied, along with three-dimensional visualization (see Fig. 1). In spite of the fact that this previous work achieved the expected results, the need to develop a new methodology for the design and implementation of three-dimensional simulations of robotic systems became necessary due to limited simulation performance, lack of flexibility for customizing the MATLAB toolkits at low level, lack of a fully customized user interface, and limitations when using commercial software.

This article shows a new methodology for the design and implementation of 3D computer simulations for

Manuscript received July 10, 2014; revised October 21, 2014; accepted January 2, 2015. Recommended by Associate Editor Sukho Park under the direction of Editor Euntai Kim.

This work has been supported by Proyectos Basales y Vicerrectoría de Investigación, Desarrollo e Innovación (VRIDEI), Universidad de Santiago de Chile, Chile.

Claudio Urrea is with the Electrical Engineering Department, Universidad de Santiago de Chile, Av. Ecuador 3519, Estación Central, Santiago, Chile (e-mail: claudio.urrea@usach.cl).

Jean Paul Coltters was with the Electrical Engineering Department, Universidad de Santiago de Chile, Av. Ecuador 3519, Estación Central, Santiago, Chile (e-mail: jean.coltters@usach.cl).

* Corresponding author.

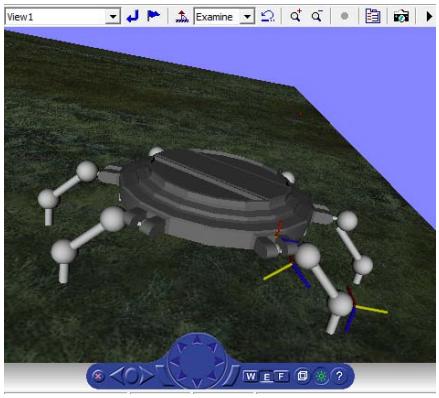


Fig. 1. Hexapod robot in Simulink.

cooperative manipulator robots in a low-level programming language (C++). We present the development of a simulation entirely programmed in this language, allowing to adopt the flexibility and performance features belonging to this kind of programming language.

2. METHOLOGY

The development of simulation environments in C++, along with the use of some C++ specific libraries and frameworks, arises as a proposal to face the demand for higher complexity in simulations, derived from the need to consider multiple factors at the same time (trajectories, collisions, multiple robots, data acquisition, three-dimensional visualization, user interfaces, etc.). In general terms, when those multiple factors are integrated simultaneously, performance problems arise, in addition to difficulties in compatibility, modularity and inner cohesion.

The proposed methodology for the design and implementation of three-dimensional simulations presented in this work is not restrictive, since it establishes a working frame general enough to be applied in different kinds of 3D mathematical or physical simulations. Focusing on the idea of fully developing a software with no commercial elements, we also take advantage of some basic tools and open source frameworks for C++. The set of procedures and tools that compose this methodology are:

- **Pre Calculated Models:** Using the Octave software, which is an OpenSource alternative to MatLab, the different mathematical model's expressions that will be simulated and evaluated in the runtime of the simulation are obtained. These expressions with the corresponding variables and parameters of the dynamic model are translated between Octave and C++ languages with Octave's regular expression feature.
- **3D Graphic with OpenGL:** Using the OpenGL *API* to provide the C++ programmed simulation with a command interface that has high 3D graphic capabilities and high flexibility.
- **Graphic User Interface:** Employing QT SDK (Open Source) for C++ the development process for user interfaces is achieved in a simpler manner using C++ code without any external software.
- **3D Modeling:** Employing Blender software to represent of three-dimensional models that cannot be codified in a simple way in OpenGL. These models will be imported to the simulation with the C++ ASSIMP (Open Source) library for importing 3D-model mesh (vertex, edges and faces) into a C++ class information.
- **3D Collision Detection:** Although 3D collisions detection can be implemented in C++ between different 3D primitives, it is more practical to use an already well documented (Open Source) framework to handle this topic.
- **Software Design:** To keep programming as modular as possible through object-oriented programming, so as not to limit possibilities or increase the complexity of the works to be simulated.

3. MODELED ROBOTS

The robots modeled in this work were designed and implemented in the Robotics Laboratory of the Departamento de Ingeniería Eléctrica of the Universidad de Santiago de Chile. Next, we will give a brief description of those robots.

The robots can be classified as SCARA (Selective Compilant Assembly Robot Arm). These robots usually perform faster and are simpler than the similar Cartesian robots but require a more complex control system. SCARA robots are commonly used for screw tightening, moving elements between process lines, assembling equipment, peg in hole tasks, stacking materials, inspection of finished products like devices with touch panels among many others.

3.1. SCARA robot # 1

This robot, shown in Fig. 2, is a SCARA-type robot with 6 degrees of freedom. Of these six degrees of freedom, three are used for positioning the end effector, using two revolving joints and one prismatic joint; the other three joints are meant for opening, closing and orientation of the clamp at the end of the arm. For control and supervision concerns, the manipulator has an interface developed in LABVIEW.

Of the three joints used for positioning the robotic arm in a desired place within the workspace, two are powered by direct current motors, controlled via Pulse Width Modulation, and the third is powered by a direct current

Fig. 2. SCARA robot # 1 with 6 DOF.

Fig. 3. Clamp of the SCARA # 1 manipulator.

Fig. 4. SCARA robot # 2.

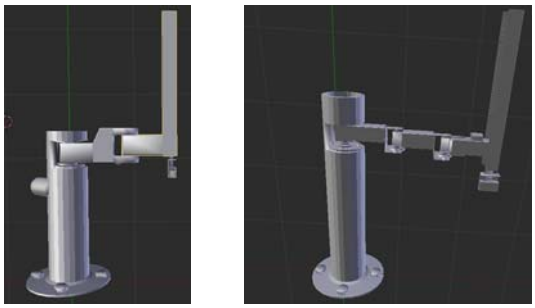
servo motor. The three remaining joints that control the opening, closing and orientation of the clamp (see Fig. 3) use direct current servo motors, allowing high-accuracy control.

3.2. SCARA robot # 2

This manipulator is very similar to the SCARA robot No. 1 (see Fig. 4), but it presents many differences with respect to a classic SCARA robot [4-8], since with the aim of being over-actuated, it has a joint configuration of the RRRPRP kind, instead of the RRPRRP configuration of a classic SCARA robot. By being over-actuated, this robot can be used to investigate fault-tolerant control techniques, among other issues. For control and supervision concerns, the manipulator has an interface developed in MATLAB.

4. THREE-DIMENSIONAL MODELING OF ROBOTS

Blender is a software specifically meant for three-dimensional design and animation, widely used all over the world. The software can be used to represent industrial robots by means of a physical shape that can be approximated in an acceptable way through a set of basic three-dimensional bodies like cubes, parallelepipeds, cones, spheres, etc., each one having its own position, rotation and dimensions. In Fig. 5 we can see the entire robots rendered in Blender.



(a) # 1. (b) # 2.

Fig. 5. Three-dimensional models of manipulators.

4.1. Modeling with dimension flexibility

The two 3D robotic models, as shown in Figs. 6 and 7, are formed by several kind of parts, specifically divided into base floor, base, base top, first segment, first joint, second segment, second joint, etc. This division is required since from the user interface 3D developed simulation will allow modifying properties for example:

- Segment size (any)
- Base height
- Clamp dimensions
- Robot width (whole robot)

The modification of these properties without a proper separation between parts would cause visual deformations when changing the robot body size in an arbitrary way. For example, if the three parts of Fig. 7 were a unique whole 3D model for the robot base, and the simulator user wanted to scale the robot base height to five times the normal, even the screws of the base would grow five times. Similar problems occur when changing segment size or other parts.

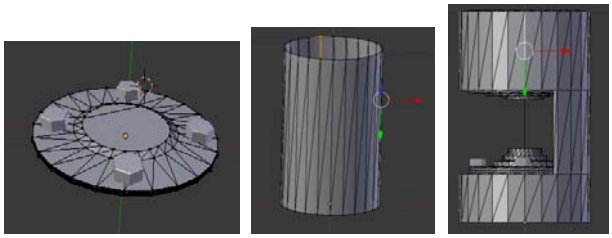


Fig. 6. Base of manipulator #1.

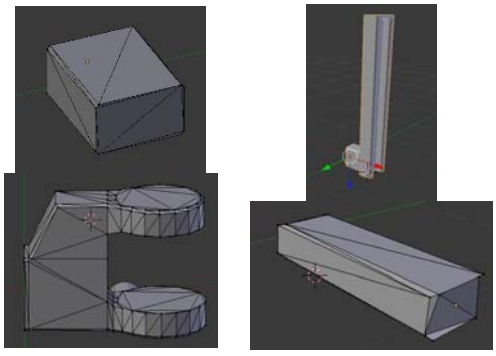


Fig. 7. Arm of manipulator #1.

4.2. Showing the 3D models in OpenGL

The Blender-made 3D models are saved as .obj format files and read with the C++ library Assimp, which stores the information of each part of the robots inside the simulation runtime memory, and is ready to be sent to the OpenGL interface functions.

After the Assimp library imports the code, the information is not ready to be drawn, since translations, rotations and scaling operations must be performed on each 3D part depending on the current direct kinematics of each robot and the size of each configurable part of the robot's body, so this algorithm was also developed for the simulation.

5. DYNAMIC MODELS

To ensure that the mechanical models employed in this work are correct, we implemented a programmed algorithm inside Octave based on the Lagrange-Euler method [9], which gats the dynamic equations of movement from each robot's Denavit-Hartenberg (D-H) table and as output produces C++ code with the inverse dynamic matrix information that was compiled along with the rest of the simulator and evaluated at runtime.

5.1. Dynamic model generation algorithm

This algorithm is entirely done in Octave and its results are added as native C++ code inside the simulator before the simulator compilation. This approach was used due the requirement of using mathematical operations like the derivative to work with the dynamics equations, and the need of symbolic expressions. Using Octave as a C++ framework is another possible approach that could be studied if in the future it is required to add more types of robots in the simulator runtime.

The general expressions used for the dynamics of a robot are the inverse dynamic model and direct dynamic model, shown in (1) and (2), respectively.

$$\tau + J^T \cdot F_{Ext} = M(q)\cdot\ddot{q} + C(q,\dot{q}) + G(q) + K\cdot\dot{q}, \quad (1)$$

$$\ddot{q} = M^{-1}(q)\cdot(\tau + J^T \cdot F_{Ext} - C(q,\dot{q}) - G(q) - K\cdot\dot{q}), \quad (2)$$

where:

$M(q)$: Inertia matrix of the manipulator.
$C(q,\dot{q})$: Centrifugal and Coriolis effects vector.
$G(q)$: Gravitational forces vector.
q : Joints position vector
q : Joints speed vector
K : Friction vector
τ : Joint torques vector.
F_{Ext} : External forces acting on the arm.
J^T : Transposed Jacobian of the end effector.

Our approach is to export to C++ code just the *M*, *C*, *G* and *K* matrices individually and then operate them in runtime as shown in (1) and (2). The matrices, as mathematical expressions that depend on the robot's properties (mass, segments size, center of mass position), joints speed and position, can be ported to an equivalent C++ robot class methods with position and speed arguments that returns each of the evaluated matrices.

The inputs of the Octave algorithm are the D-H and the center of mass tables, which give information about the position and orientation of the links, joints and centers of mass of the robot, besides the total number of degrees of freedom that the robot has. The degrees of freedom have a direct relation with the number of rows in the table; position and orientation are functions of the robot's length parameters and the joint variables (angles in the case of revolving joints, or lengths in the case of prismatic joints).

The Lagrange-Euler method requires having expressions that determine the kinetic and potential energies involved in the robot; those expressions are derived from the position and speed of the different segments or, in an equivalent way, from their respective centers of mass. The Denavit-Hartenberg table [2-4,8-10,13] is used to determine the position and orientation of each joint of the robot, but holds no information about the centers of mass; hence it cannot be used as a unique input that represents all the robot's morphology information needed for the application of the Lagrange-Euler method.

In order to simplify the determination of centers of mass positions we used an additional type of table very similar to Denavit-Hartenberg's, The new table uses the same 4 D-H parameters to represent in a simple manner the transformation between multiple coordinate systems. The difference between those tables lies in the fact that in this new table each row represents the relationship between the reference system placed in joint # i and the coordinate system placed in the center of mass # i, while in the D-H table each row indicates the transformations

Table 1. Denavit-Hartenberg Table of SCARA # 2.

Joint # i	θ_i	d_i	α_i	a_i
0	θ_1	L_{Base}	L_1	0
1	θ_2	0	L_2	0
2	θ_3	0	L_3	π
3	0	θ_4	0	0
4	θ_5	L_5	0	0

Table 2. Centers of mass of SCARA # 2.

Center of mass # i	θ_i	d_i	α_i	a_i
0	θ_1	L_{Base}	L_{c1}	0
1	θ_2	0	L_{c2}	0
2	θ_3	0	L_{c3}	0
3	0	θ_3-L_{c4}	0	0
4	0	L_{c5}	0	0

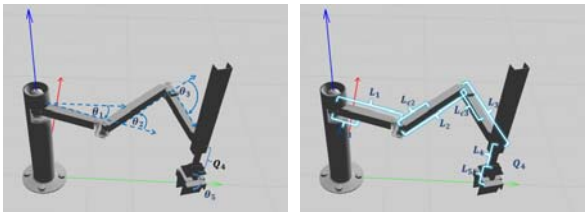


Fig. 8. SCARA robot # 2, joint angles, segment lengths and center of mass positions.

between two successive joints, that is, joint # i and joint # i +1. The D-H and center of mass matrices for the different positions of the SCARA robot # 2 are shown in Tables 1 and 2.

where:

L_{Base}: Height of the manipulator base where the first joint is located.

L_i: Length of segment # i of the robot arm.

L_{ci}: Center of mass position along the length of arm segment # i.

θ_i: Angle of joint # i.

After getting the position of each center of mass, the usual Lagrange-Euler method is applied to obtain the *M*, *C*, and *G* matrices. The direct kinematic models are obtained as a consequence of using the Denavit-Hartenberg tables; the position of each joint is one of the first steps of the calculation of dynamic models by means of the developed algorithm.

The resulting matrices' (*M*, *C* and *G*) inner expressions are translated to C++ code with Octave's regular expressions feature. The matrices are operated as (1) for the inverse dynamics, and for the direct dynamics along with the numeric evaluation an inverse matrix is calculated using the Eigen C++ library, while in runtime each model is iterated according to the simulation needs through an order 4 Runge-Kutta method.

6. INVERSE KINEMATICS

The inverse kinematics of the SCARA robot # 1 is calculated by decoupling the positioning and orientation equations of the end effector.

The inverse kinematics of the SCARA robot # 2 is calculated using homogeneous transformation matrices [1,4-6,8,10-12].

Considering that the matrix transformation between the base and the end effector has the following form:

$$ {}^{0}T_{N} = \begin{pmatrix} R_{11} & R_{12} & R_{13} & p_x \\ R_{21} & R_{22} & R_{23} & p_y \\ R_{31} & R_{32} & R_{33} & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3} $$

And ${}^{0}T_{N}$ can be decomposes as:

$$ {}^{0}T_{N} = {}^{0}A_{1}\,{}^{1}A_{2}\,{}^{2}A_{3}\,{}^{3}A_{4}\ldots{}^{n-1}A_{n}. \tag{4} $$

The previous equation can be written as

$$ ({}^{n-2}A_{n-1})^{-1}\ldots({}^{1}A_{2})^{-1}({}^{0}A_{1})^{-1}\cdot{}^{0}T_{N} = {}^{n-1}A_{n}. \tag{5} $$

The inverse kinematics model for the end effector is obtained after working with the resultant relations of (5) applied to each joint. For simplicity the inverse kinematic is obtained in this work considering that $\theta_2 = \theta_3$.

7. TRAJECTORY GENERATION

It's generated using user input that specifies the end effector position and orientation in multiple time frames,

Fig. 9. Trajectory manager.

this way the precision of the trajectory will have direct relation to the amount of trajectory points specified. Using the inverse kinematics of each robot, each point of the end effector trajectory in world space is transformed into its corresponding joint space representation.

The discrete vectors at joint space are then translated into continuous position and speed joint curves employing third degree polynomial interpolations. The joint trajectory T will be generated as shown for the desired group of joint space trajectory points $Q_{j,i} = Q_j(t_i)$ and $\dot{Q}_{j,i} = \dot{Q}_j(t_i)$ for I = 1…n, where j is the joint number. T(t) can be written as:

$$ T(t) = \sum_{i=1}^{n-1} \begin{bmatrix} P_{1,i}(t) \\ P_{2,i}(t) \\ P_{3,i}(t) \\ \vdots \\ P_{m,i}(t) \end{bmatrix}, \tag{6} $$

where each polynomial $P_{j,i}(t)$ has the following values

$$ P_{j,i}(t) = \begin{cases} a_1^{j,i}\cdot t + a_2^{j,i}\cdot t^2 + a_3^{j,i}\cdot t^3 & \text{if } t_i \le t \le t_{i+1} \\ 0 & \text{if } t < t_i \vee t < t_{i+1} \end{cases} \tag{7} $$

and the arguments of the polynomial $P_{j,i}$ are calculated as follows:

$$ \begin{pmatrix} a_1^{j,i} \\ a_2^{j,i} \\ a_3^{j,i} \\ a_4^{j,i} \end{pmatrix} = \begin{pmatrix} 1 & t_i & t_i^2 & t_i^3 \\ 1 & t_{i+1} & t_{i+1}^2 & t_{i+1}^3 \\ 0 & 1 & 2t_i & 3t_i^2 \\ 0 & 1 & 2t_{i+1} & 3t_{i+1}^2 \end{pmatrix}^{-1} \cdot \begin{pmatrix} Q_{j,i} \\ Q_{j,i+1} \\ \dot{Q}_{j,i} \\ \dot{Q}_{j,i+1} \end{pmatrix}, \tag{8} $$

where:

t_i : Time of point # i interpolated by the trajectory generation system.

$P_{j,i}$: Polynomial that interpolates the trajectory for joint Q_j between times t_i and t_{i+1}.

$a_k^{j,i}$: Coefficient number k of polynomial $P_{j,i}$.

These whole operations are hidden beneath an interface in the simulator that allows the generation of trajectories easily:

8. SCREEN DRAWING OF BACKGROUND AND ROBOTS: OPENGL

Using OpenGL5, low-level directions for drawing vertices, lines and points on the screen are processed; we can draw most of the graphic content of the simulation, except for the user interaction items like menus, buttons and basic elements of the window. The inputs for the different directions correspond to the position of each of those vertices, lines and points, along with background lighting parameters, colors, width of the edges, and others.

In the design stage, the three-dimensional models are exported to the .obj format; when starting a simulation, those files, that have an easy-to-decode syntax, are translated by the library of C++ Assimp, generating a list of elements like vertices, edges, and faces, which are stored in the RAM memory of the application, and can be drawn later on the screen by OpenGL during the execution cycle of the simulation.

9. PROGRAMING STRUCTURE

In object-oriented programming, classes play a fundamental role, allowing to gather in a strategic way data with related operations, actually grouping them to have a higher degree of order, and also making possible an increase in the capacity for modularity and interaction between the different systems.

The behaviors and data of each element of the simulation are encapsulated in classes. The main systems in the simulation, each one representing an agent associated with its own class within the C++ code, are:

- SCARA1: Class with features and behavior of SCARA robot # 1.
- SCARA2: Class with features and behavior of SCARA robot # 2.
- DataManager: System in charge of storing a position record of the robots, at each moment, for the generation of graphic.
- 3DCamera: Control system for the three-dimensional camera (Position, Orientation, and Zoom).
- PathPlanner: System for design of trajectories with third order polynomial interpolation.
- Framerate: Management system for simulation time.
- BulletControl: System for collision detection.
- Controller: Abstract class for controller polymorphism.
- PID: PID controller class.
- NLController: Nonlinear controller class.

In order to get a seamless flow between interface actions, algorithm processing, and screen drawing, the simulation programming was developed in two threads, separating tasks.

In addition, the internal processes of the processing and drawing thread are more complex than the processes of the user interface thread, which only depends on user actions to trigger events. The content of the processing and drawing thread is detailed in Fig. 11.

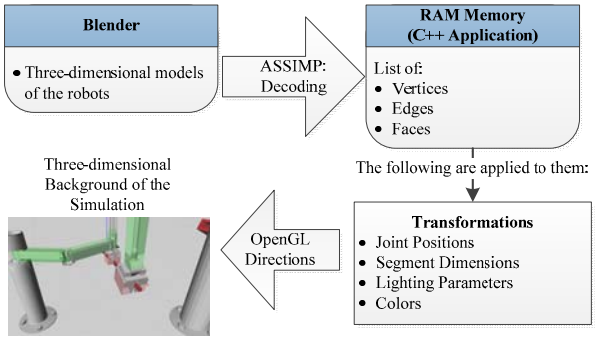


Fig. 10. Process of drawing the robots on the screen.

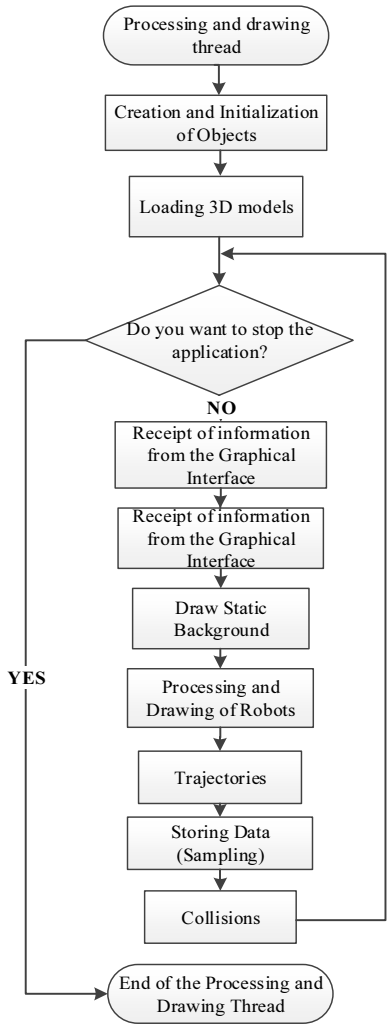


Fig. 11. Simulation threads.

10. COLLISION DETECTION AND PHYSICAL CONSIDERATIONS

Using Bullet we can incorporate into our C++ virtual bodies with advanced collision systems without having to program the mathematical collision detection algorithm for each shape; the Bullet framework has a variety of standard shapes, starting from simple bodies like cubes, to complex three-dimensional meshes that can be used for collision detection and rigid body dynamic simulation.

10.1. Bullet rigid bodies

Bullet bodies use collision detection and all the common dynamics calculations related to rigid bodies movement and collision, considering forces, torques, impulses, collisions, elasticity, inertia, etc. These kinds of bodies are fully controlled by the bullet framework. And will collide and respond according to their rigid body parameters. These bodies are good for simulating collisions and body interactions for simple objects, but the authors still prefer to simulate the intrinsic robot interaction, due to its complexity, with the calculated mathematical model.

10.2. Collision detection bodies

These kinds of bodies do not necessarily represent an actual rigid body, they are used to detect collisions between custom shapes and inform the rest of the C++ programmed software in any way it requires. The collisions, when detected, will send callback to sections of the software that are subscribed to them.

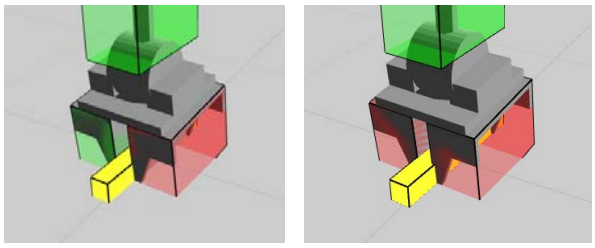
10.3. Use of bullet in this project

Bullet is normally used to fully simulate physical environments, but this is not the case of this work, where bullet was not used to produce the dynamic behavior of the robots since it is generated by iterations of Lagrange-Euler equations, but was used instead as a supporting tool to incorporate other less complex rigid bodies adjacent to the robots that can interact with them and provide additional features to make the simulation more interesting. Bullet was used to do the following tasks in this simulator:

- Add free rigid bodies to the virtual world that have their own mechanical models that allow them to interact between them and make the scene more interesting.
- Add rigid bodies over each of the robots segments and body parts, so that when the robot (product of our implemented model movement) collides with a free object there is a force interaction. The rigid bodies added to the robot segments are slightly modified by code to be fixed to each segment's position and orientation regardless of the interaction forces.
- Deliver force feedback when the robot is colliding with an external free object, bullet Rigid Bodies returns contact force and position information to the rest of the code that is introduced in the dynamic model's external force input.
- Give contact information to the rest of the C++ for further flexibility.

Although Bullet holds all these capabilities, much of it had to be programmed and modified, to be able to fully control de bodies from outside the bullet framework. Specifically, to achieve a hybrid behavior, to change between a rigid body behavior and a collision detection behavior when certain conditions are met.

For example a short algorithm for holding objects with the clamp was implemented, that simulates the conditions of friction, position, and orientation required by the clamp and an object so that the clamp can lift the object



(a) Immediately before holding the object. (b) Beginning to hold the object.

Fig. 12. Process of holding an object with the clamp.

(see Fig. 12). The algorithm checks, among other aspects, if the free body is colliding with both ends of the clamp, and if the orientation of the object is appropriate.

Although Bullet has tools for calculating the contact forces when rigid bodies are colliding, contact interaction is complex when dealing with two different frameworks that are interacting. It is a very complex task to design, with accuracy, the interaction between the mathematical model of the robots and the Bullet rigid bodies, so most of the collisions shown by the simulator only show a collision warning and have no force interaction. The force interaction between these frameworks is currently being studied in a new work.

Taking this into consideration, as an approximation (if enabled inside the simulator) the Bullet physics engine will feed the mathematic model of the robots with an external force due to interactions with other bodies (2), this is used to simulate collision with bodies on the ground and to simulate load sharing forces between robots carrying the same object.

11. RESULTS

11.1. Graphic interface

Thanks to the designed graphic user interface, the simulations are configurable –within a specified range of options– by means of the previously designed user interface. This graphic interface is very effective to assist in customizing several aspects of the simulation without the need to modify the internal C++ code of the simulator. The interface has multiple windows, each of which allows customizing or activating different sectors of the desired simulation (see Fig. 13). Those windows have flexibility features somewhat inspired by the Blender interface. Most of them have variable size and can be

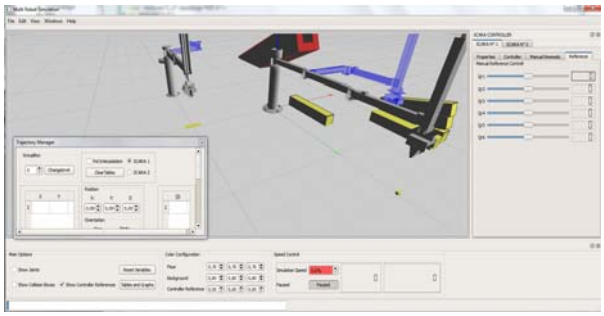


Fig. 13. User interface.

placed in different positions to meet user's preferences. The buttons, the horizontal bars and other components included in this graphic interface allow the interaction with the user using the SIGNAL-SLOT system of QT Creator, generating an effective interaction between the simulation's internal processes and the user interface. This communication allows real-time modifications of the internal systems' parameters, in addition to the call for the execution of the systems' processes. QT Creator allows the implementation of multiple windows, each one for a specific function.

11.2. Collisions and bullet library

The physics and collisions library in Bullet allows to include the ability to detect collisions between the different segments of the robot in the simulation environment, besides simulating free objects in the background. When collisions are detected, multiple red lines intersecting at the collision point are displayed. Furthermore, for better visualization the option to show the boxes that approximate the volume of each segment for the detection of collisions is implemented, shown in green color if the segment is not colliding and in red if it is actually colliding. The option Show Collision Boxes of the graphic interface enables the visualization of those polyhedral.

Collision detection becomes very useful if we plan to include some features not commonly analyzed in previous simulation work, like volume and shape of the robots.

Fig. 14 shows a collision between two robots, where the collision points are drawn in red. Those points can be stored, and may be employed to implement algorithms or as a consequence of them, as in the case of bouncing, etc. The collision information also has a vector indicating its direction, allowing further implementations of forces as consequences of the collisions.

It is seen that C++ has great programming flexibility, which becomes quite remarkable when compared with previous simulations written [9], for example, in the M language of MATLAB. This flexibility, along with the libraries and software tools that were used, enables a high degree of simulator customizing. The object-oriented programming used in its design allows developing every system separately and modifying them easily through changes in the simulator's source code. In this way we can include drastic changes like the incorporation of new kinds of robots, new types of controllers, improvements in aspects of the simulated physic models or in the user interface, etc.

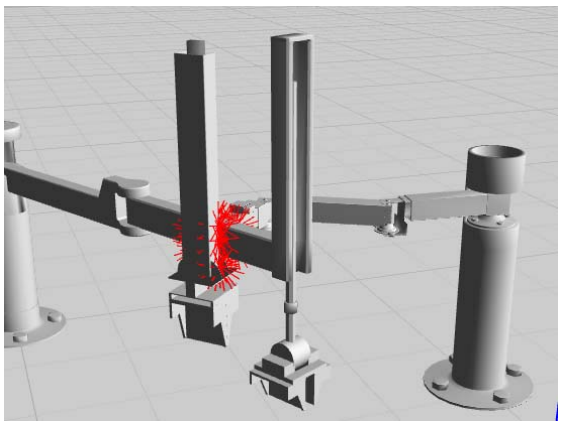


Fig. 14. SCARA # 1 and SCARA # 2 robots colliding.

This simulator could be used with any type of manipulator if the correct Denavit-Hartenberg tables are provided. In case that the dynamics of the robot requires additional equations or rules they could be added in any way as an equation or condition check in C++.

11.3. Data and graphic

The simulation has a data acquisition system allowing the position of the different joints of each robot to be stored; finally showing those in a table and a graphic (see Fig. 15). Those data, by pressing a button, can be exported to Excel for a deeper analysis.

11.4. Speed control

The "Speed Control" section of the graphic interface has a direct relationship with the simulation's speed and the FrameRate management system described in section 8; here we can configure the speed at which the simulation's time will run with respect to the real time. We can configure a speed from 0% (pause) to 400%, where 100% means that the time elapsed inside the simulation will be equivalent to the real time.

In terms of the simulation's performance, we reached an average of 350 images per second and a step time of 0.0002015 seconds, measured by a display included in the simulation. These results were very satisfactory if we want a fluent and highly accurate simulation, with a good margin to add more complexity by incorporating new systems or increasing the simulation's speed.

The simulation's step times were measured by executing the cooperative application described in this same chapter, in real time (100% speed), between both SCARA robots. The equipment used for running the simulator is a low-grade PC (2 GB RAM, Intel Core 2 Duo at 2.8 GHz).

Fig. 16 shows the interface's time supervision and monitoring window where, along with the control op-

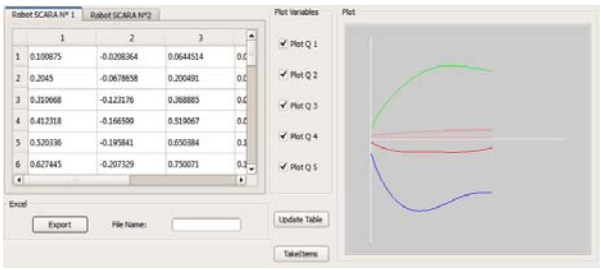


Fig. 15. Data and graphic of the simulation.

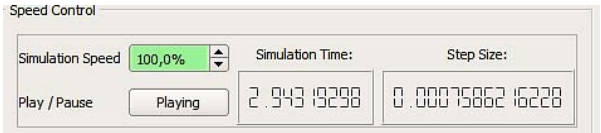


Fig. 16. Simulation speed.

tions already mentioned, information is included on the simulation's elapsed seconds (Simulation Time) and the size of the time steps (Step Size) used by the RungeKutta algorithm and the internal iterations of Bullet, a variable that is dependent on the simulation speed and the processor and hardware capabilities.

11.5. Developed industrial application

In order to validate the simulator, not just as a tool for individual tasks but also for cooperative jobs, we devised a task that requires both robots to move an object and place it in a specified position.

With the purpose of developing an application related to actual applications of manipulator robots, we implemented a hollow inclined parallelepiped (see Fig. 17) where some objects picked up from the ground are placed. Once inside the parallelepiped, by means of gravity those objects are allowed to slide and are transported to another place.

The developed application simulates the handling of bars of considerable lengths, moved from an initial position inside this container element. The difficulty associated with this application lies in the process of planning and generating adequate trajectories to avoid collisions between the robots, the different objects, and between a robot and the object.

Cooperative work sequence:

1) In the neighborhood of SCARA robot # 2 there is a bar that has to be moved.
2) SCARA robot # 2 picks the object from the ground and, without making it collide with the robot itself, moves it next to the SCARA robot # 1.
3) SCARA robot # 1 holds the object in the air, with both robots holding the object at the same time.
4) SCARA robot # 2 releases the object, which is therefore held only by SCARA robot # 1.
5) SCARA robot # 1 moves to a position near the depositing place, with the proper orientation to avoid colliding with the walls of the hollow parallelepiped.
6) SCARA robot # 2 performs movements that allow inserting the object without collisions.
7) SCARA robot # 2 releases the object, which slides – by gravity– inside the parallelepiped container.
8) The sequence is repeated from step 1.

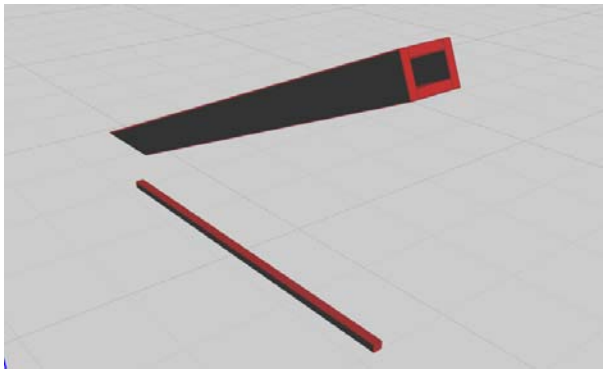


Fig. 17. Hollow parallelepiped and bar to be inserted.

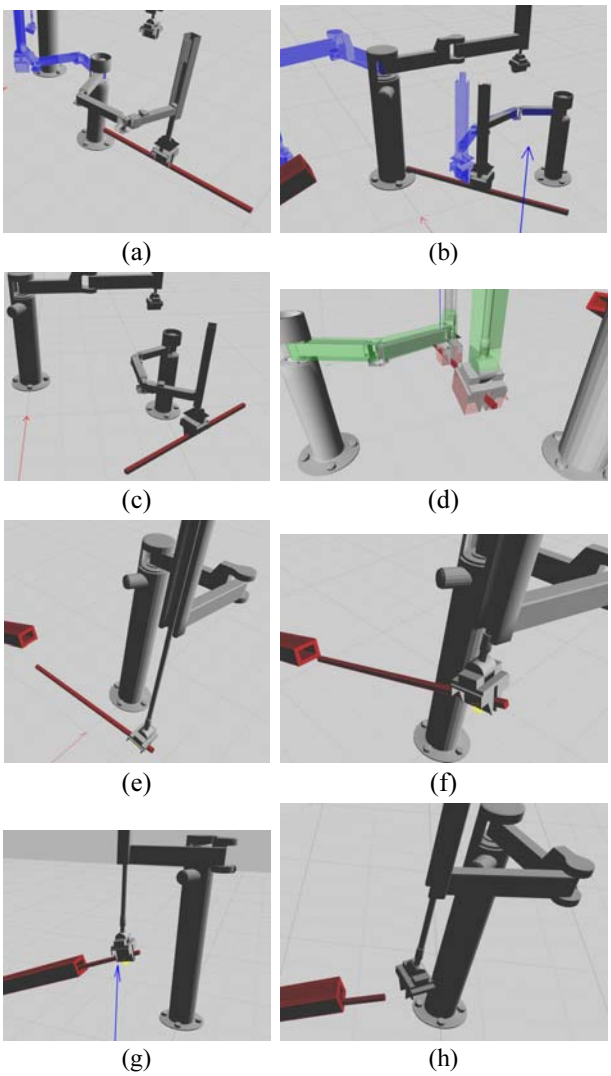


Fig. 18. (a) SCARA robot # 2 lifting a bar. (b) SCARA robot # 2 moving a bar to SCARA # 1. (c) SCARA robot # 2 moving a bar avoiding collisions. (d) Both robots holding a bar. (e) SCARA robot # 1 preparing to orient a bar. (f) SCARA robot # 1 orienting a bar to insert it in the hollow parallelepiped. (g) SCARA robot # 1 inserting the object. (h) SCARA robot # 1 returning to starting position while the object slides in.

11.6. Sequence of images describing the simulation

Fig. 18 shows a sequence of images that depict what happened when the application described in the previous section was implemented. The robots' desired positions are displayed in transparent blue, the bar to be handled is shown in red, and at the moment both robots make contact with the bar, the manipulators, except their bases, are shown in green.

12. CONCLUSIONS

The present paper shows the integration of multiple systems and platforms into a single final result. The cooperative simulation performed has a high performance,

with up to 400 images per second in a medium-grade PC, allowing the size of Runge-Kutta steps to be small enough to allow the simulation to be precise when run in real time, and the execution of the task can be observed in full detail.

Every joint of each robot has different associated loads, according to the structural specifications of each kind of robot, so in the case of using PID controllers, for example, each joint must be configured having in mind its individual features.

The visualization of the simulations is done in real time, allowing to observe and quantify some aspects of the simulation that would not be seen only through graphics, like the existence of any kind of collisions, for example, especially those that cannot be known by the evaluation of simple mathematical operations.

This work shows high flexibility, offering great educational potential for the study of physical aspects in robotic systems and mechanical systems in general, besides easing the implementation of complex simulations that are not limited only to a couple of equations, but are subjected to more complex restrictions.

Through the integration of multiple specialized computer tools it is possible to obtain interesting applications that could hardly be solved using a single software or tool. On the other hand, the detection of collisions implies a great enhancement in the simulation of mechanical systems, a feature that can be employed in combination with the dynamic representation of an almost infinite variety of systems, for both simulation or reality, or for testing and implementing algorithms for the design and tuning of controllers, including trajectories that take into account three-dimensional geometric features for their operation.

REFERENCES

[1] J. Craig, *Introducción a la Robótica*, 3rd ed., Pearson Education. México, 2006.

[2] H. Ghariblu and A. Javanmard, "Maximum allowable load of two cooperative manipulators," *Proc. of the Second International Conference on Computer Engineering and Applications*, pp. 560-570 2010.

[3] A. Irawan and K. Nonami, "Force threshold-based omni-directional movement for hexapod robot walking on uneven terrain," *Proc. of the IEEE Fourth International Conference on Computational Intelligence, Modelling and Simulation*, Kuantan, Malaysia, pp. 127-132, 2012.

[4] A. Mohammad, Khedher A. and A. Mahdi S, "SCARA robot control using neural networks," *Proc. of the IEEE Fourth International Conference on Intelligent and Advanced Systems*, Kuala Lumpur, Malaysia, pp. 126-130, 2012.

[5] B. K. Rout and R. K. Mittal, "Optimal design of manipulator parameter using evolutionary optimization techniques," *Robotica*, vol. 28, no. 03, pp. 381-395, 2010.

[6] D. P. Garg, "Adaptive control of nonlinear dynamic SCARA type of manipulators," *Robotica,* vol. 9, no. 03, pp. 319-326, 1991.

[7] J. Lee, "Velocity workspace analysis for multiple arm robot systems," *Robotica*, vol. 19, pp. 581-591, 2001.

[8] P. Majewski and R. Beniak, "Robust control of scara manipulator," *Proc. of the third International Students Conference on Electrodynamics and Mechatronics (SCE III)*, pp. 47-51, 2011.

[9] W. Khalil and E. Dombre, *Modeling, Identification and Control of Robots*, Butterworth-Heinemann, Elsevier, France, 2004.

[10] Z. Zhao, K. Hei, and R. Du, "The simulation of scara robot based on OpenGL and STL," *Proc. of IEEE Second International Conference on Mechanic Automation and Control Engineering*, Hohhot, Mongolia, pp. 5429-5432, 2011.

[11] M. Kemal Ciliz and M. Ömer Tuncay, "Comparative experiments with a multiple model based adaptive controller for a SCARA type direct drive manipulator," *Robotica*, vol. 23, no. 06, pp 721-729, 2005.

[12] M. Sami and R. Patton, "Active fault tolerant control for nonlinear systems with simultaneous actuator and sensor faults," *International Journal of Control, Automation, and Systems*, 11, pp. 1149-1161, 2013.

Claudio Urrea was born in Santiago, Chile. He received his M.Sc. Eng. and his Dr. degrees from Universidad de Santiago de Chile, Santiago, Chile, in 1999, and 2003, respectively; and a Ph.D. degree from Institut National Polytechnique de Grenoble, France in 2003. Ph.D. Urrea is currently a Professor at the Electrical Engineering Department, Universidad de Santiago de Chile, from 1998. He has developed and implemented a Robotics Laboratory, where intelligent robotic systems are development and investigated. He is currently the Director of the Doctorate in Engineering Sciences, Major in Automation, at the Universidad de Santiago de Chile.

Jean Paul Coltters was born in Santiago, Chile. He received his M.Sc Eng. from Universidad de Santiago de Chile, Santiago, Chile in 2012. Currently he is researching 3D simulators and control systems for robotics and other physic related systems.