MoRoS3D, a multi mobile robot 3D simulator

Eric Colon[1], Hichem Sahli[2], Yvan Baudoin[1]
[1]Royal Military Academy- Department of Mechanics
Avenue de la Renaissance 30 – 1000 Brussels – Belgium
[2]Vrije Universiteit Brussel - Department of Electronics and Informatics (ETRO)
Pleinlaan 2 – 1050 Brussels - Belgium
email: eric.colon@rma.ac.be, hsahli@etro.vub.ac.be, yvan.baudoin@rma.ac.ac.be

Abstract: **This article presents a Java based 3D simulator for mobile robots called MoRoS3D. This application is able to simulate realistic motion of different wheeled mobile robots including dynamic behavior and collision detection. Typical sensors are also available in order to develop intelligent navigation applications. As this simulator provides CORBA interfaces for every active objects, applications can be written in any language supporting this standard. The design and implementation as well as a typical control application are presented in this paper.**

Keywords: **3D simulation, mobile robots, CORBA**

I. Introduction

Having a simulator offers many advantages. First of all it is tremendously cheaper than real robots and sensors, particularly when experimenting with multi robots systems. It allows focusing on intelligence and control and disposing of other, less interesting problems. It makes possible reducing the development time by trying different scenarios and algorithms before experimenting them in a real environment. A simulator also increases safety when developing and testing new control applications. Developing a simulator can be easier or harder than building (or buying) hardware. For instance, simulating a high fidelity stereo vision system would require a lot of work and could cost much more money than buying equivalent hardware.

Different approaches are possible for the modeling step. In some implementations the user has to write equations representing the dynamical behavior of the simulated system or to draw a 3D model including physical properties, this model being also used for the visualization of the results. In the former solution a separate 3D model has to be provided and the visualization is generally handled by a separate animation application. Commercial software's are generic tools and must consequently be versatile and provide easy to use interfaces for model creation and results visualization.

For instance, in the commercial software "Universal Mechanism"[1] (UM) the user draws the model and defines the constraints in a program called "UM Input" while the simulation and the visualization are provided by the "UM simulation" application. UM must be combined with Matlab® if the multi-body simulation has to be embedded in a global control scheme.

In the simulation library EasyDyn [1] the user has to provide the position equations and the applied forces. Accompanying tools automate the creation of the motion equations and generate a C++ program that the user has to complete with additional control equations. Results are saved in files that can be read by third party applications (GNUPlot, EasyAnim, ...).

The Open Design Engine (ODE)[2] is a library that proposes a mixed approach. The user writes a program (in C++ or Python) that describes the simulated system by using objects provided by the library (world, nodes, joints, forces, torques, ...). This library also provides methods for resolving the implicitly generated equations. The visualization part is the responsibility of the developer who has to use third party libraries like Opengl.

Simulators can be divided in off-line and on-line simulators. Off-line simulators compute motion of objects at their own pace and produce data that can be visualized as a movie once the simulation is completed. The aforementioned examples enter in the off-line simulation category. On-line simulators are interactive; the motion of the objects can be modified in real-time by control algorithms or by a user via a GUI or a joystick. The motion of the controlled object are visualized in real-time in 2D or 3D.

The Java based simulator, called MoRoS3D, that has been developed in this work enters in the on-line category. Control commands and environmental conditions can be changed interactively. Furthermore, it runs in real time using any available communication systems and replaces the real hardware in the application control loop in order to test the control components.

II. Simulator Overview.

For the user, the visible output of the simulator is a synthetic image. Actually, it is not only an image but it is also a model that is built with algorithms based on physical laws and using well defined data structures. The simulator provides the following functionalities:

- Real-time simulation of multiple robots concurrently
- 3D real-time visualization of the simulation
- User interaction through a GUI
- Dynamic control of mobile robots
- Detection of and appropriate reaction to collisions between mobile and fixed objects
- Simulation of position and distance sensors
- CORBA interfaces

The simulation process is divided in two main steps: the modeling of 3D scenes and robots by a human and the

[1]Http://www.umlab.ru

[2]Http://www.ode.org

utilization of the modeled objects in the simulator. These two steps are explained in the following sections and illustrated by the diagram of Figure 1.

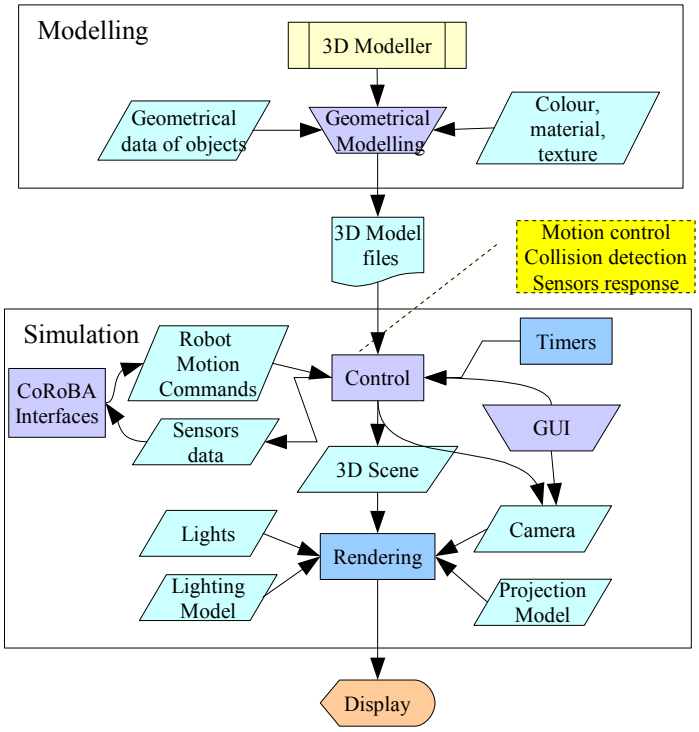

Fig. 1. The simulation process

Starting from a real or an hypothetical robot, the creator uses a 3D drawing program to generate a virtual model. Other information like colours, material and texture can be applied to the objects to improve the realism. Real or imaginary environments (terrain and obstacles) are created separately from the robots. Wings3D uses its own format for internal and external representation but models can also be exported in other popular formats. The exported model is encoded in VRML (Virtual Reality Modelling Language).

The VRML file is read by the application and transformed by the Java3D [2] import library into a Java3D scene graph and inserted in the global 3D scene.

The process flow (control-rendering-display) represented in Figure 1 continuously runs until the application finishes. The control process updates the Scene (section 4), controls the motion of the robots (section 5), performs the collision detection and response (section 6) and finally computes the output of position and distance sensors (section 7). The Control also receives motion commands for the robots and sends sensors' data via the CoRoBA interfaces (section 8). It can also control the camera motion in automatic tracking mode. The GUI (section 3) lets the user chose the camera mode and position and gives the possibility to position the robot in the virtual world.

The execution of the control process is triggered by timer events. As each robot and sensor is represented by separate objects, the events are propagated to all of them. This means that all motion and measurements are synchronized.

Once all transformations of the 3D scene have been performed, the scene is rendered by the Java3D rendering engine. This engine uses different information in order to produce an image that can be displayed on the screen:

- The lights present in the scene.
- The lightning model. Here a Gouraud shading is used for calculating the illumination of the scene.
- The point of view given by the camera position and other viewing information (field of view, near and far clipping distances, ...).
- The projection model, which is a perspective projection in our case.

The rendering engine of Java3D can use the DirectX or OpenGL libraries.

III. Graphical User Interface

MoRoS3D allows to place a robot in a 3D environment and to let it interact with that environment in a manner similar to robots situated in the real world. Although the user visualizes the entire surroundings of the robot, the robot software only ”sees” the information it collects through its sensors, just like a real robot would do.

As can be seen in Figure 2, the main part of the Graphical User Interface (GUI) is off course devoted to the 3D view. On the right of the GUI lie several widgets for managing cameras, robots' position and trajectory plots. The user can choose several viewpoints corresponding to virtual cameras in the 3D scene. There are also two mobile cameras, one on board (button BOARD) and one at the vertical of the robot that points downward (button TRACK). With the NEXT button the user jumps from robot to robot when in tracking or on-board mode. The user can also specify the robots' location and reset one or all robots in a single operation. There is also a button to erase the trajectory plots left behind moving robots. Under the 3D view, the name, position and orientation of the selected robot are displayed.

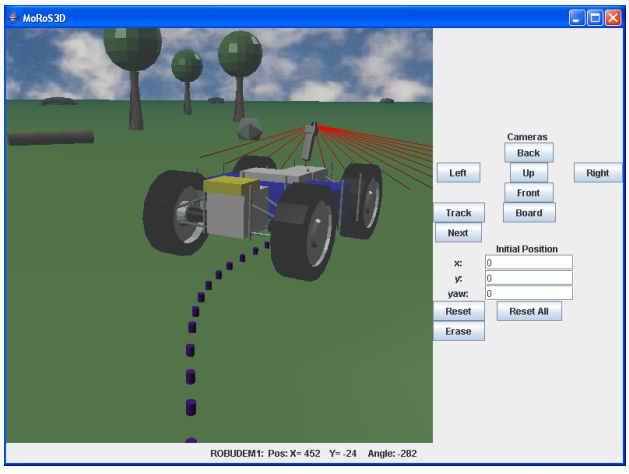

Fig. 2. GUI of MoRoS3D

IV. Scene Graph

Many free and open-source toolkits are available for building 3D applications[3]. However, most of them focus on

[3]More than 230 engines are recorded in the database of the site http://www.devmaster.net

visual aspects and few offer high level facilities for managing scenes. This is one reason justifying the use of Java3D for the development of the simulator. Java3D is a full-featured API for interactive 3D graphics. It is based on a high-level scene graph programming model that describes the scene, Java3D managing the display of it. Scene graphs are treelike data structures used to store, organize and render 3D scene information. They are made up of objects called nodes, which represent objects to be displayed, aspects of the virtual world or group of nodes.

Nodes and *NodeComponents* are the basic elements of the scene graphs. Nodes can be divided into the following basic categories:

- Shape nodes, which represent 3D objects in the world.
- Environment nodes, which represent characteristics of the world such as light, fog, sounds, etc.
- Group nodes, which organise the scene graph.
- The *ViewPlatform*, which is a place where a viewer can look at the world.

Group is the base class for a number of classes that position, orient and control scene graph objects in the virtual universe. The two subclasses used in MoRoS3D are *BranchGroup* and *TransformGroup*. *BranchGroup* holds sub-graphs that can be added and removed while the scene is being displayed. *TransformGroup* changes the transformation of its children, giving them a different position, orientation and size.

By default, each object in a Java3D scene is initially stationary and remains at its starting location unless code specifies otherwise. A *TransformGroup* is associated with a *Transform3D* structure that corresponds to a 4x4 transformation matrix. A single *Transform3D* object can represent a translation, a rotation, a scaling or a combination of the three. A transformation turns the X,Y and Z coordinates of a point into a new set of coordinates:

This relations can be expressed with 4x4 matrices, where $[x\ y\ z\ 1]^t$ are the original and $[x'\ y'\ z'\ 1]^t$ the transformed coordinates:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (1)$$

There are many methods to create and modify *Transform3D* objects. These include methods to make a *Transform3D* have a translation, scale or rotation. When a *TransformGroup* is the child of another *TransformGroup*, the effects of their Transform3D objects are multiplied so that all the children of the child *TransformGroup* are affected by both sets of transforms.

NodeComponents are nodes that hold properties or data. *Shape* nodes are *NodeComponents* that consist of two properties: the geometry, which specifies the 3D coordinates and the appearance, which specifies the colour and other properties of the shape.

SimpleUniverse is a Java3D utility class that manages low level functionality as for instance 3D to 2D mapping. The *SimpleUniverse* renders the image in a *3DCanvas*, which is a drawing widget added to the *View3DPanel*. The *ViewPlatform* is a member of the *SimpleUniverse* used to transform the viewpoint with the mouse via predefined behaviours (*MouseZoom*, *MouseRotate*, *MouseTranslate*).

The *worldBGroup* (*BranchGroup*) contains environmental node such as lights, fog and background and the *sceneBgroup* (*BranchGroup*).

The objects of the 3D world have been divided in three groups: the terrain, the obstacles and the robots. This separation provides flexibility in the composition of the scene. The *sceneBGroup* therefore contains the *terrainBGroup*, the *obstacleBGroup* and the *robotBGroup*.

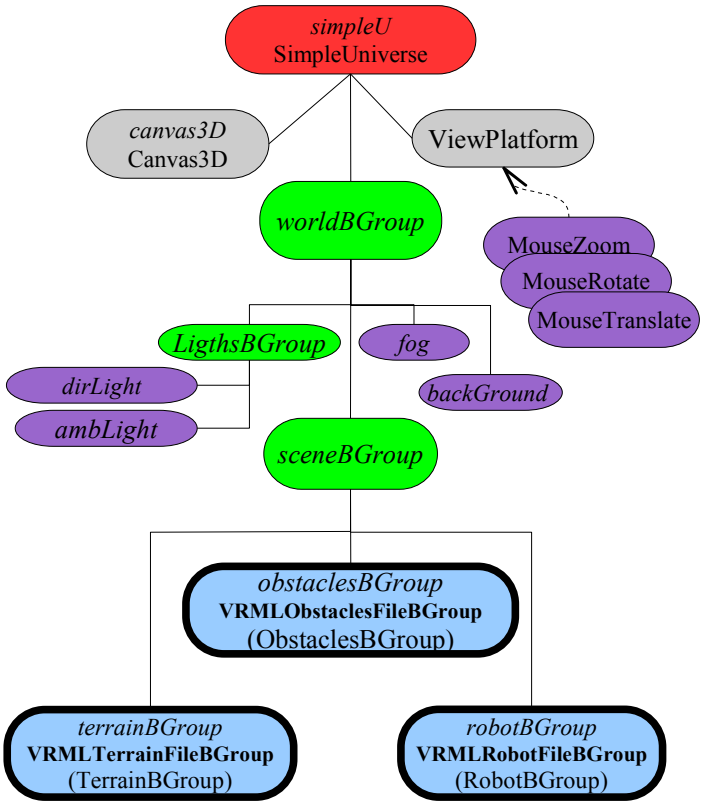

Fig. 3. Scene graph.

V. Robot Models

The geometry of robots is determined by their shape and dimensions. The 3D models have been drawn with a 3D modeling application (Wings3D) and exported in the VRML format.

A. Nomad

The Nomad (Figure 4) has a simple geometry and only visible parts have been modeled.

The Nomad is actuated by a synchronous mechanism, each wheel is capable of being driven and steered. The three steered wheels are arranged as vertices's of an equilateral triangle and all the wheels turn and drive in unison. Actually the real Nomad has a third degree of freedom, the turret can turn independently of the base but this mechanism has not been implemented in the model.

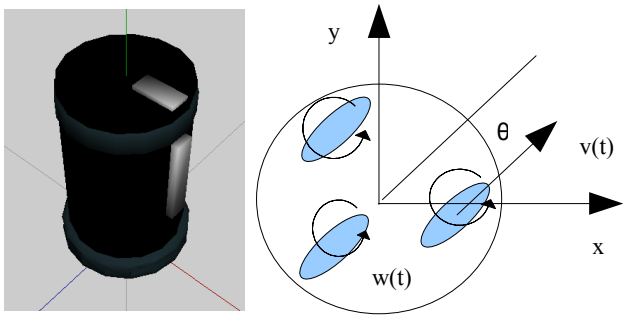

Fig. 4. 3D and kinematic models of the Nomad

The motion equation of the Nomad are:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \begin{bmatrix} v_k \cos\theta_k \\ v_k \sin\theta_k \\ w_k \end{bmatrix} \cdot h \qquad (2)$$

The Euler algorithm [JAME85] is used as integration method to obtain the position from the velocity. The standard Euler integration method requires a single forcing function evaluation, and produces a first order accurate solution. The algorithm for a single variable is simple:

$$x(t+h) = x(t) + h\, v_k \cos\Delta_k \qquad (3)$$

The algorithm is applied repetitively to compute a solution for the state at equally spaced intervals of time. The Euler method is known for accumulating errors at each integration steps. We neglect these errors here as we are more interested by global behaviors and environment interaction than by exact trajectories.

Determining the real dynamic behavior of such a robot is not a trivial task, but as the motion of the real robot is controlled by a PID controller, we assume that it can be reasonably modeled by a first order differential equation. The steer speed and the translation speeds are consequently updated according to the following equations:

$$\begin{aligned} v_k &= v_{k-1} + (v_c - v_{k-1}) h/\tau_t \\ w_k &= w_{k-1} + (w_c - w_{k-1}) h/\tau_s \end{aligned} \qquad (4)$$

Where τ_s and τ_t are the estimated time constants of the system and v_c and w_c are the command speeds. These constants have been adjusted for the typical dynamic behavior of the Nomad taking into account the standard value for accelerations (These values can be changed by calling the appropriate function of the Nomad API).

B. Robudem

The geometry of the Robudem is more complicated than the one of the Nomad. The Robudem has four wheels that are individually actuated by electrical motors. The two axles are steerable and are actuated by two linear electrical motors via an Ackerman mechanism. The following figures show the real robot and its 3D model.

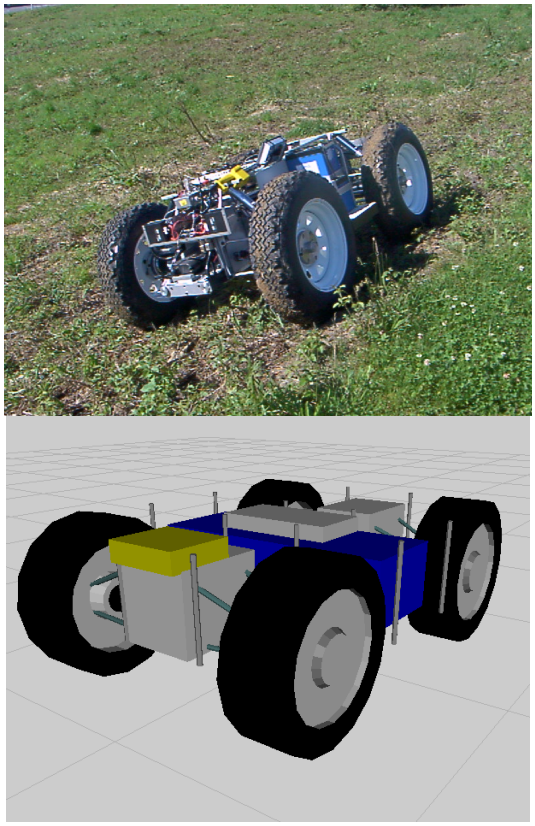

Fig 5. Picture and 3D model of the Robudem

The trajectory control of the real Robudem is based on two parameters: the instant desired speed v and the instant desired steering lock α. Indeed, at each time, the vehicle trajectory can be expressed with those two values that are given by the user through a joystick interface or by a control program.

Supposing a perfect Ackermann steering mechanism for the front axle results in the instantaneous center of rotation lying on the axis of the rear axle. In this case we can use a bicycle model for representing the kinematics of the vehicle (Figure 6): the four wheels are replaced by two wheels located in the middle of the vehicle.

Let the angular velocity vector along the body z axis be $\dot{\theta}$. Using the bicycle model approximation, the radius of curvature R and the steer angle α are related by the wheelbase L. By definition of the curvature:

$$\frac{d\theta}{ds} = \frac{1}{R} = \frac{\tan\alpha}{L} \qquad (5)$$

$$\dot{\theta} = \frac{d\theta}{ds}\frac{ds}{dt} = \frac{1}{R}\; v = v\frac{\tan\alpha}{L} \qquad (6)$$

$$\Delta\theta_k = v_k h \frac{\tan\alpha_k}{L} \qquad (7)$$

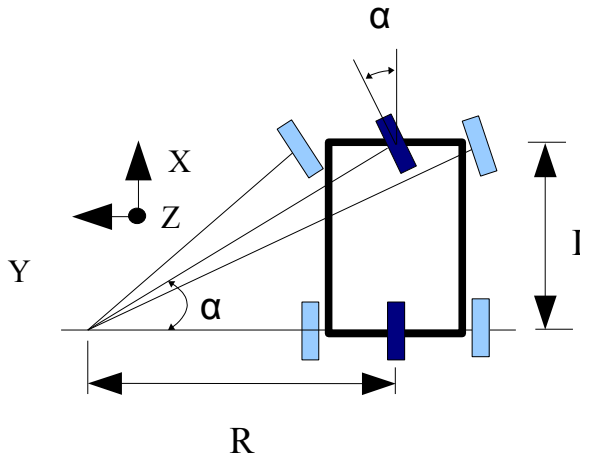

Fig. 6. Kinematic model of the Robudem

Once the incremental angle has been obtained, The model of the equations 2 and 3 can be used. Simulating the dynamic behavior of Robudem is based on the same model as for the Nomad (Equation 4). Off course the time constants have been adapted to reflect the dynamics of this robot. Another particularity of the real controller that has been taken into account is the following: when the user suddenly puts the joystick in neutral position, the controller immediately stops the robot while when he pulls it gently, the speed is reduced by applying a linear profile.

VI. Handling Collisions

The previous section has presented the motion control of robots in open environment, that is without any obstacles. Off course in any realistic application robots have to cope with static and dynamic obstacles. In the developed simulator, dynamics obstacles are other mobile robots while the environment is static. It this then necessary to be able to detect and to react to collisions. Moving autonomously implies detecting and avoiding obstacles. One of the basic requirement of the simulator is consequently to provide collision detection to detect when the control algorithm fails and the robot collides with the environment or with other robots and to provide adapted response.

Java3D provides classes for detecting collisions between objects. However, this API works asynchronously and does not offer any guarantee when the detected collision will be reported, what happens generally after the object has entered into another one. This is not an appropriate mechanism and therefore a collision detection algorithm exploiting Java3D Behaviors has been implemented. Collision detection is more a geometric problem than a physical one. To make sure that any area of space cannot be occupied by more than one object, collision detection based on the geometry of the objects is required [3].

For any realistic environment and even if simplified shapes are used, the collision detection needs a lot of mathematical operations. each object is composed of hundreds of triangles and the collision detection would required too much time if it had to be performed for any triangle. During the simulation we need to check for collisions at every frame therefore it is important that collision detection be very efficient. We therefore need to apply a method to speed up the computation. Hopefully, there exist different optimization methods for reducing the amount of operations. For instance, bounding volumes can be used to reject non intersecting objects. Axis Aligned Bounding Box (Figure 7) and spheres are the most used bounding volumes.

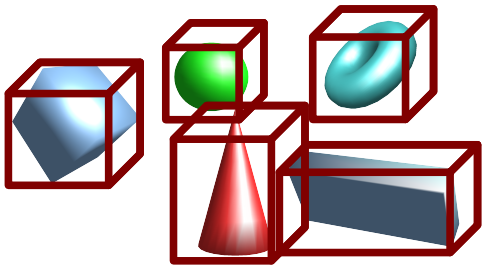

Fig. 7. Bounding boxes

If a single volume around the object does not give accurate enough collision detection for the shape then it is possible to use multiple boxes in a hierarchical way to more accurately match the shape of an irregular object.

Using bounding volumes reduce the computing cost by eliminating objects that do not collide but it may not be good enough to rely on the bounding box or sphere alone especially if the objects are complex shapes. However they can at least filter out those objects that do not overlap. Another reason that we cant rely on bounding rectangle or sphere alone is that in order to go on to the next stage of working out the collision response we also need to know the points of impact.

If we want to test for collision of meshes, made up from triangles, and we want to check for collisions accurately, using all the information from the geometry, we may need to test each triangle on object 'A' with each triangle on object 'B' for intersection. Currently the most efficient test is the algorithm of Moller [4] that is explained below.

We first determine the equation of the planes containing the triangles and work out the intersection line for the two planes (Figure 8). Intervals Ia and Ib are are computed. If Ia ∩ Ib ≠ ϕ than the two triangles intersect.

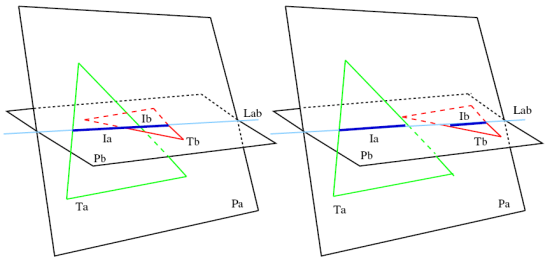

Fig. 8. Testing a possible intersection between two triangles

Java3D offers methods for calculating intersection between bounding boxes of objects. However automatically computed bounding boxes have always their axes parallel to the global reference frame. This gives an unrealistic representation for any real object.

Therefore the method used in the simulator is a compromise between the two approaches presented above. It consists in replacing mobile robots by a good approximation and checking for collision with the real geometry of other objects. The Nomad is for instance simply replaced by a cylinder. The Robudem has basically a box shape and it is more suited to define its contour manually by specifying a bounding box defined by six orthogonal plans what in the Java3D jargon is called a "Polytope".

In order to render realistic collision occurrences we must be able to predict these events before they effectively happen in order to avoid that a robot enters into an object. Knowing the actual speeds we compute for each frame the posture. With this prediction we check if a collision occurs with any fixed or mobile obstacles.

Once we have detected a collision between two objects, we can compute the reaction by using physical laws and by considering for each object the velocity, the mass, the centre of mass, the inertia, ... As in targeted applications real robots are moving slowly we do not need complex collision response because most of the time robots are simply blocked when they move into an obstacle. So in case of collision we stop the robot by disregarding the last transformation.

VII. Sensors

Two kinds of sensors are necessary for developing intelligent control applications in mobile robotics: position and environment perception sensors.

Global position sensors can be easily implemented within the simulator because we perfectly know the position and orientation of the robot and of all its components. Relative position sensors and low level encoder signals can also be derived from this global position knowledge.

A mobile robot can only act intelligently if it perceives its environment. Distance sensors are mandatory for seeing what stands around the robot. Three models of such sensors have been implemented in the simulator, namely laser, infra-red and ultrasonic sensors. To implement the measurement process we have used Java3D's picking routines. The idea is to cast a ray into the space around the robot. This ray has a length equal to the maximum distance the sensor can measure.

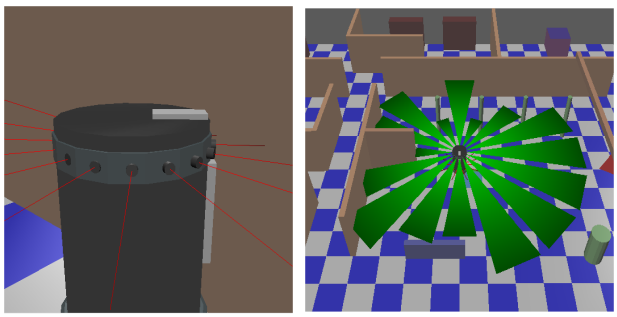

Fig. 9. Simulated Laser and Ultrasonic sensors

VIII. Developing Applications

A. Principle

MoRoS3D integrates seamlessly with the control framework that has been developed at the Royal Military Academy. CoRoBA is a solution package for developing distribution applications that uses components with standardized interfaces and communication mechanisms. Components are divided in Actuators, Processors and Sensors.

The utilization philosophy is to develop and tune control algorithms in simulation and to simply replace simulated by real components once satisfying results have been reached, no further modification of the Processor components being required. In Figure 10, the concept of integrating MoRoS3D in the CoRoBA framework is shown. Sensor and Actuator components developed with CoRoBA can be seen as interface components that have to be specific for the simulator or the hardware they are linked to.

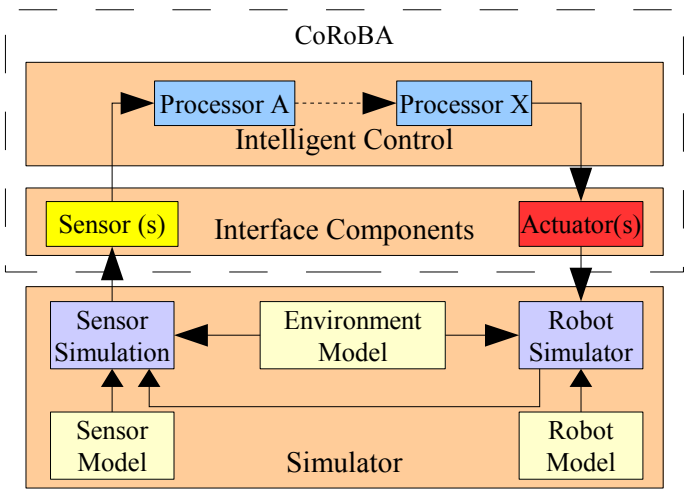

Fig. 10. Simulator and CoRoBA integration

The block named "Intelligent Control" on top of Figure contains Processors. This part does not care if real or simulated hardware is used. The Processor components are the key-stone of the control architecture and exhibit the largest potential of reuse between applications involving different robots while Sensors and Actuators, that serve as interfaces or translators between the software and external modules, are specific to these devices. The more abstract a Processor is, the greater the possibility of reusing it without any modification.

The middle block corresponds to interface components that make the link between the Processors and the simulated world. Sensor and Actuator components implement the same interfaces as those implemented by components linked to physical systems, allowing to instantaneously switch between simulation and reality.

The last block represents the simulator. It is constituted by different elements that are described hereafter. First of all it contains models of the physical elements. The robot model deals with the geometric, kinematic and dynamic aspects of the robot. The sensor model encodes information about the sensors like the radiation model, the minimum and maximum distances, the precision, etc. The environment model contains the 3D geometrical representation of the environment. The robot simulator is responsible for the realistic motion of the

robot and takes care of the collision with fixed and moving obstacles like other robots. It receives motion commands from Actuator components.

The simulated sensors produce measurement data that are injected in the application control loop by the Sensor components . The data is forwarded to Processor components where they are exploited to finally produce motion commands that are sent to the Actuator Components. These ActuatorComponents adapt and send this information to the robot objects. The sensors affect the vehicles motion through Intelligent Control and vehicles motion affect sensors through the Simulator taking into account the model of the environment.

B. Example: Goal Navigation with the Robudem

The purpose of this application is to let the Robudem move autonomously from a given position to succession of goals in an obstacle free environment. The components involved in this application and the transferred data are shown in Figure 11. The Goal Controller uses a Fuzzy Inference System.

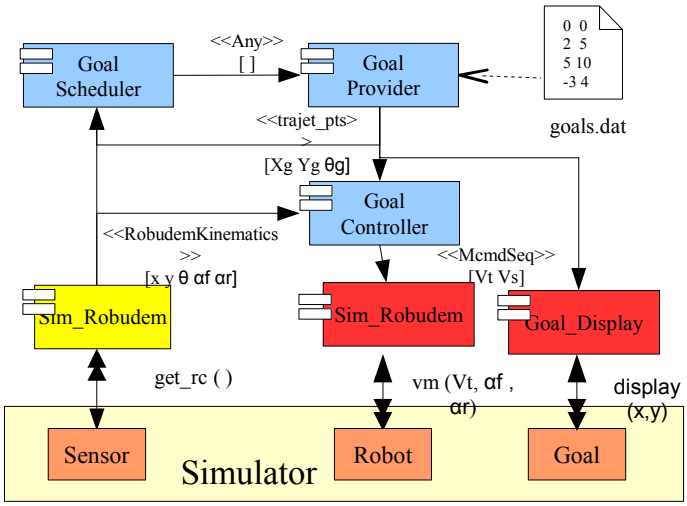

Fig. 11. Components' network

During the execution of the application, the following operations are executed:

- At initialization, the *Goal_Provider* reads a list of goals from a file (goals.dat).
- When the components are started, the first goal position [Xg Yg θg] is sent to the *Goal_Controller* and to the *Goal_Scheduler*.
- These components also receive the global position of the robot [x y θ αf αr] from the *Sim_Robudem* Sensor component).
- The *Goal_Controller* uses this information to produce steering and driving commands [Vt Vs] in order to reach the goal .
- These commands are received by the *Sim_Robudem* Actuator that adapt them to the controlled robot.
- The *Goal_Scheduler* compares the goal position received from the *Goal_Provider* with the instantaneous position of the robot. When the robot is sufficiently close to the goal, the *Goal_Scheduler* sends an event to the *Goal_Provider* to inform it that it has to provide the next goal .
- A new goal is sent to the *Goal_Controller* and *Goal_Scheduler* .
- These operations are repeated until the last goal is reached.

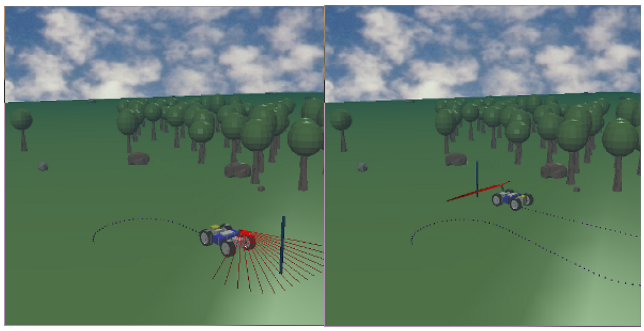

Fig. 12. Two steps of a navigation sequence

IX. Conclusions

This chapter presented MoRoS3D, a multi-robot and sensor simulation application that simulates 3D environment for developing mobile robots applications. This simulator is versatile enough to simulate different types of robots.

The 3D scene modeling and rendering is based on Java3D and is therefore platform independent. It is extensible and users can easily change the environment (terrain and obstacles) as they are passed as command line parameters, they do not need to recompile the application.

Finally, it integrates seamlessly into the CoRoBA framework thanks to the CORBA middleware.

Despite this tight integration, the simulator can also be used independently of the framework and control applications can be written in any language supporting CORBA interfaces.

Keeping the control algorithms out of the simulator has the advantage that an application developer does not need to deal with Java3D programming.

References

[1] Verlinden O., Kouroussis G.,Conti C., EasyDyn: A framework based on free symbolic and numerical tools for teaching multibody systems, Proceedings of the Multibody Dynamics 2005, ECCOMAS.

[2] Walsh A. & Gehringer D, Java3D API Jump-Start, Prentice Hall, ISBN 0-13-034076-6.

[3] Ericson C., Real-Time Collision Detection, The Morgan Kaufmann Series in Interactive 3-D Technology, 2004, ISBN: 1558607323.

[4] Moller, T., A fast triangle-triangle intersection test. Journal of Graphics Tools. (1997).