Abstract

This work presents the implementation issues arising from the development of a mobile robot simulator by using a game development engine. The developed simulator takes advantage of the graphics engine and of the physics engine available in the video game development platform. We show how the use of these tools enable us to obtain realistic simulations in the execution of robotic tasks including sensing and motion primitives. We will describe the elements composing the mobile robot simulator and how they were verified in some robotic tasks simulated in the application developed. We will also present the main issues arising in the development process.

1. Introduction

The main goal of mobile robotics is the design and implementation of autonomous systems. Autonomy is the capability of an agent, in this case a mobile robot, of executing tasks of increasing complexity. The complexity of a task relates to the input information known by the agent about the task prior to its execution, to the difficulties inherent to the processing needed to elaborate a plan to execute the task and to the problems arising in the implementation of the actual solution. If we take as an example the robot navigation problem, the task of the robot is going from A to B, if there exist a way to navigate between them. To execute this task, the robot needs to have information about the environment, knowledge about its own capabilities to sense the environment and to move and execute the motion task, a path panner module to determine the optimal way taking into account all the constraints and an execution controller charged of implementing the planned solution in the actual environment.

When we are confronted with more complex tasks, we need to have experimentation tools where our solution proposals could be verified before their implementation in a real robot. That is why we need simulation tools that let us to test our theoretical methods using realistic virtual robots in controlled scenarios. In these kind of simulators we need to have access to the different capabilities of the robot both for sensing and actuation in a realistic manner, in such a way that simulation results can be extrapolated to the execution of the same task in real robots. The interest of developing robotic simulation platforms is made patent by works like TeamBots [22], ARIA [18], MissionLab [4], Miro [24], among many others. Every simulation platform uses specific programming languages and interaction mechanisms between the elements that can be used to represent the robots object of the simulation and possibly with even others developed in some other programming languages.

According to Kramer et al. [13], the main elements of a robotic simulation environment are:

**Platform** The target hardware for the application being developed. The main platform elements are: the sensors, the actuators, the computer, the operating system and any other elements like any software relevant for the robot operation or even any other hardware controlled by the platform.

**Components** The elements that operate as part of an agent or system and that function in an independent way.

**Architecture** The structure of the components and the interaction mechanisms provided for its communication.

**Agent** The integration of the software and the hardware in a robot to execute the tasks assigned to it.

**Programming environment** The tools and the infrastructure used for the implementation of programs to execute the robotic tasks.

In our case, the main objectives to develop a robotic simulation environment using a game development engine are:

- To take advantage of the realistic interactions between the robot and all the other elements of the environment where it evolves that the game engine provides. These advantages are: i) The use of the physics engine embedded on the game development environment, specifically in the simulations of sensors and actuation capabilities of the robot. An example of sensors that could be simulated in a more realistic way are the range sensors like sonars and laser range finders; ii) the use of the graphics engine for rendering scenes seen by virtual vision sensors. Most of game engines use advanced graphics rendering systems that could provide virtual images where to test advanced robotic vision algorithms.
• To design robot architectures according to specific application needs. That is, to test different choices for the control architecture of the robot in realistic scenarios. This implies that we can select among different sets of modules, to identify essential modules in a given application, to identify critical elements in each of the modules composing the robot and so on. All these tasks take advantage of a more realistic simulation.

These objectives are similar to those stated by Faust in [7]. Faust also states the set of possibilities that such mobile robotics simulation platform enables. In this paper, we will describe the particular issues that have arisen from the development of another simulation tool and how they are related with the target robot hardware for this simulation. In contrast to many approaches that tend to the generalization of robot models, both commercial and open source, our interest is more oriented towards the design and implementation of the robot simulator as a highly customizable but accurate tool both for scenario modeling and for the robotics platform itself [14].

Rest of this paper is organized as follows: Section 2 presents a review of game development engines, its main features and how can they be used advantageously as the base of a robot simulator. The elements of our robot simulator are presented in Section 3. We present a detailed description of each element of our simulator. Section 4 describes the architecture implemented for the simulated robots in our development. Test used to validate our simulation tool and the results obtained in this tests are presented in Section 5. Finally, Section 6 presents the main conclusions of this work.

2. Game development engines

Game development engines were developed to simplify the process of design and implementation of video games. Typical capabilities for this type of engines include: tools for rapid development, versioning control, collaboration tools for development teams, compatibility to standard graphic file formats (for example png, bmp and jpeg file types), compatibility with 3D model file formats (for example, 3ds, fbx file types), graphics engines, physics engines, among others.

A game development engine is designed to output applications of high performance. Most of the commercial video games in the market satisfy requirement such as: to exhibit high level of realism, to be a very stable product, to have possibility of executing concurrent actions (maybe in a remote way), to be a very robust product, to be capable of simulating realistic environments and interactions between the scenario elements, to interact with the users in a reliable way, among others. Many of these requirements are also needed for the development of a mobile robot simulator and that is why have motivated our interest in developing such a simulator using a video game development environment.

2.1. A review of game development engines

Game development engines can be classified according to the licensing format in commercial licensing and open source licensing engines.

Some of the more popular commercial engines for video game development are: CryEngine [5], Havok [11], Source [20], Unity [1] y Unreal Engine [10]. These engines have been used in many different games in the market. As a typical requirement of their customers, they provide a wide support to use their environments. Depending on the specific engines, we can expect to pay low costs for using them in non commercial applications. Some of them offer gratis licenses for this kind of applications, even if they are associated to reduced capabilities versions with respect to fully commercial licenses.

In the case of open source game development engines, we can cite for example: JMonkey [12], id Tech [23], Crystal Space [21] and OGRE [15]. For these engines, the support is reduced to a volunteer community and to the learned lessons from the engine developers or some other advanced users of the same environment. This results in difficulties to get the most from the capabilities of this kind of game engines.

Another important issue is the programming languages used for the development of the game development engine and those supported for developing applications using it. There exists a variety of programming languages used for this purpose. However, the dominant languages for these tasks are C++ and script-based languages like Javascript.

As we have mentioned above, one of the main requirements for a video game application is to be able of executing several concurrent processes concurrently. For this reason, one of the most important features to search for in a game development environment is a simple mechanism to associate objects in the game scenario to programmed actions according to different game and user-provoked interactions. This simplicity is an important point to take into account when making a decision about what game engine is the most suitable for the development of a specific application.

2.2. Unity game development engine

We have chosen the Unity game development engine to design and implement the mobile robot simulator application described in this paper. Unity is one of the more recent engines that have appeared in the market. This game development tool have an extensive support for multimedia file formats. Unity is multi-platform targeted, enabling us to develop Windows and Mac OSX applications without changing the application source code. Another interesting feature is the possibility of developing rapidly Web applications. This game engine has been recently used to develop games for most commercial game consoles. Also, the availability of a wider base of support documentation for Unity than for other game engines (e.g. Unreal engine 3) has favored our choice of this game engine as the development platform for this project.

The programming language used for programming games in
the same time of flight measurement principle as the sonar sensor. However, this device uses a laser beam as the active component. In the particular case of the Sick LRF, we can have up to 181 measures through an angular sweep of $[-90, 90]$ with a resolution of $1^\circ$. The use of a laser enables the LRF to have a larger range than the sonar and to exhibit a better accuracy in the measurements. All the measures of the laser are taken at a given plane according to the relative position and orientation of the device on the mobile robot.

3. Mobile robot simulator

The main element of the simulator is the mobile robot. The simulated mobile robot is an autonomous entity with sensing and actuation capabilities. These capabilities are represented in such a way to be realistic enough as those available in the real robot. The robot simulation is executed in an independent way with respect to the other models in the scenario. The robot needs to have mechanisms to keep knowledge about itself and about the scenario through its sensors. In a client-server architecture, the robot is the data server and the clients are the programs that access to the information generated by the robot, its interaction with other objects in the environment and its temporal evolution.

In order to be able to simulate the mobile robot in real time, we need to update the models in the robot scenario using efficient methods both in time and computational power. The development platform used for this system let us to set up the updating time interval for each separate model in the environment. This is crucial to be able to simulate in a realistic way all the motion primitives in the mobile robot.

3.1. Elements of the simulator

The basic elements of the mobile robot simulator reproduce the sensorial and actuation capabilities of the mobile robot and the physical environment where the robot will evolve. Every object needs to be modeled in 3D using any 3D modeling tool, for example Maya or even a simpler tool like SketchUp. In particular, the robot instance that has inspired the functions and capabilities presented here is a Pioneer P3-AT mobile robot. This robot is a differential drive robot (DDR) that moves accordingly to the velocity variation imposed to its wheels. The main elements of this robot are:

Kinematic model This is the model that governs robot motion when its wheels are powered. We use the model proposed in the literature for DDR-type robots (for example, in [19], [6]).

Sonar The sonar is a sensor that measures the distance to the first object struck by an oriented beam emitted by an ultrasonic source. The robot has several sonar sensor located at specific positions of its physical structures and oriented at specific angles in order to be able to take measures of the surrounding environment.

Laser range finder (LRF) The laser range finder device uses

Odometry The robot has also proprioceptive sensors to measure its displacement. For this purpose, we take into account the rotation of its wheels by detecting the change of state of the encoders associated in each wheel.

Vision-based sensor This sensor provide us the richest amount of information about the robot environment. The sensing is performed by a camera equipped with vision sensor, either CCD or CMOS-type. The sensor reading is converted to a digital format and transferred to computer. There are two ways to perform this operation, if the camera offers an analog output, a frame-grabber is used; if the output is already in digital format, the information is sent using a communication protocol like IEEE-1394 or USB. The result of this process is a digital image. To perform this in the simulator, the game engine takes a snapshot from the camera position in the robot using its graphic rendering capabilities.

For each of the measurement devices, the simulator provides mechanisms to perturb the ideal measures in an individual way. We have added a Gaussian additive noise to each measurements. We can then configure the system even to a no noise condition in order to verify algorithms performance. We will provide more details about each element in the following.

3.2. Kinematics model

The current position of a mobile robot can be represented as point $P$ in a global reference frame $\{X_I, Y_I\}$ and its orientation, represented by $\theta$, as the angle with respect to such global reference frame. The combined robot pose is represented as $x_I$. The subindex $I$ is used to indicate when a pose is given with respect to the global reference frame.

$$x_I = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

Figure 1 shows an example of a robot pose in the global frame. The point $P$ is located in the center of the robot.
In order to determine the mathematical expressions of the robot kinematics, we need to take into account the contribution of each wheel movement to the variation of the robot position. The differential robot exhibit a pure translational motion if its two wheels rotate in the same direction and at the same angular velocity. If one of the wheels rotate at a different velocity or direction, this will provoke a rotation of the robot around the point \( P \).

If we assume that each of the robot wheels has a given radius \( r \) and that they are separated by a distance \( l \) from the robot reference point \( P \), and if they exhibit angular velocities \( \dot{\phi}_1 \) and \( \dot{\phi}_2 \), for the right and left wheels respectively; the robot motion with respect to the global reference frame can be evaluated as:

\[
\dot{\xi}_I = R(\theta)^{-1} \dot{\xi}_R
\]

where \( R(\theta) \) is defined as:

\[
R(\theta) = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 \\
\sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 1
\end{bmatrix}.
\]

The game engine takes into account the physical properties of the object for computing the distance that will be returned to a ray casting request. It uses the physics engine embedded on it. In the case of the laser range finder we cast 181 rays sweeping the orientation of the ray cast and using for all rays the position where the LRF is located on the robot.

In Figure 2, we show the laser model. We use \( \theta \) as the aperture angle, \( h \) is the height and \( l \) is to define the maximum depth of the ray used. In the simulator, we need to take into account the physical properties of the object struck by the ray. In our implementation, that is performed by the physics engine embedded on the game development platform. The rays used for the LRF simulation are casted at a horizontal direction of the laser orientation in the physical position of the laser device defined in the robot model.

The sonar sensor uses a given aperture angle and the measurement returned by the virtual sensor is taken as an average of the measurements returned for the ray casts over that angular interval. In Figure 3, the sonar model is shown. \( \alpha \) and \( \beta \) are used to define the aperture angle in the vertical and horizontal axis and \( l \) defines the maximum depth of the ray.
In both cases, the measurement parameters (quantity of ray casts, aperture width, etc.) associated to the range sensors can be configured to be as similar as possible to the real sensors available in the real robot.

3.4. Odometry

In order to implement the odometry sensor in our system, we use numerical integration. The numerical integration procedure uses the angular and translational velocity information. Angular and translational are derived from the equations describing robot kinematics as:

\[
\omega = \frac{(\omega_1 + \omega_2)R}{2l} \quad (4)
\]

\[
v = \frac{(\omega_1 + \omega_2)R}{2} \quad (5)
\]

Current configuration (pose) of the robot \(\{x \ y \ \theta\}^T\) is computed by using the trapezoidal rule for numerical integration. That is,

\[
\Delta x = (\Delta t) \frac{v(t) \cos(\theta(t)) + v(t-1) \cos(\theta(t-1))}{2}
\]

\[
\Delta y = (\Delta t) \frac{v(t) \sin(\theta(t)) + v(t-1) \sin(\theta(t-1))}{2}
\]

\[
\Delta \omega = (\Delta t) \frac{\omega(t) + \omega(t-1)}{2}
\]

\[
x = \int \Delta x \, dt \approx \sum_{k=0}^{n} \Delta x(kT)
\]

\[
y = \int \Delta y \, dt \approx \sum_{k=0}^{n} \Delta y(kT)
\]

\[
\theta = \int \Delta \theta \, dt \approx \sum_{k=0}^{n} \Delta \theta(kT)
\]

4. System architecture

As we have stated before, the main element of the simulation is the real robot. The 3D model of the Pioneer P3AT robot was built in a 3D modeling tool. A view of this model is shown in Figure 4. The model is a replica of the physical characteristics of the real robot. This is an important fact, because we will need to consider the robot scale to represent another objects in the robot scenario.

The robot is composed of a kinematics module that is responsible of positioning the robot according to the velocity imposed to its wheels and to the interactions with the other objects in the scenario. The other modules in Figure 5 provide the sensing capabilities (proprioceptive and exteroceptive) of the robot. Different instances of robot could have different sensing modules, enabling us to have the simulation of heterogeneous robots in multi-robot problems [17].

All these modules are located at the physical layer of the robot. To access all the services provided by them, we use an abstraction layer. There are two purposes to use this layer: i) to provide the robot with methods to access sensors and actuators and ii) to emulate the actual implementation of the physical robot. The execution layer is in charge of the execution of tasks such as the execution of motion primitives or the reading of sensor measurements. In the decision and control layer, the robot makes the decisions needed to ensure that tasks are completed successfully.
Each one of the modules of the robots is executed in a separate thread, letting us to simulate that each robot capability is executed in an independent and concurrent way as the robot task being executed. As shown by the diagram 6, the robot by itself is a component of the entire scenario where it evolves simultaneously with all the other entities of which the environment is composed of. The simulator itself is in charge of the updates of all the entities including the interactions requiring physics engine methods, graphics rendering capabilities and the handling of the state of any message passing mechanism implemented for communication between the entities in the scenario.

5. Results

We have performed several tests to verify the sensor and actuation capabilities of the robot in some given environments. We display numerical and graphical outputs of the simulation process. As an example, see Figure 7 where a graphical snapshot of the robot using its laser range finder is presented. As we can observe there, the range measure is affected by all the objects in the field of view of the LRF.

In order to verify the motion execution characteristics of the simulated robot, we have developed a translational motion primitive. This primitive uses as argument the distance to be advanced in millimeters ($mm$). The use of this unit was chosen because it is the same that uses the API of the ARIA development platform, supplied by the Pioneer robot manufacturer as the native interface.

We have tested the motion primitive described above by advancing 1000 $mm$. The advance primitive executes a triangular velocity profile. In a first execution, we have commanded a greater acceleration limit than the deceleration limit. In a second execution, we have inverted such behavior. The expected result is that we get the same displacement for both executions. In Figure 8 we show the actual velocity profiles executed by the robot. In the horizontal axis is shown the time $t$ whilst the vertical axis represents the instantaneous velocity $v$ of the robot. In the first case, the actual displacement was 1126.40 $mm$ and for the second case 1016.96 $mm$.

In the graph, we can observe the presence of errors in the execution of the motion primitive. These errors are associated to the discretization time used in the evaluation of the output of the kinematic model. These errors will also be present if there is an overload of the host computer for the simulation process. The elimination of these errors will be possible if we can execute the simulation process in a computer system than ensures a lower discretization time. In such a system, the resulting simulation will be more realistic compared to the execution of the task in the real robot.

To test the odometry system, a motion routine was implemented. The robot performs a square path command with side length equal to 1000 $mm$ for several times. As we have mentioned, there exists some error in the execution of motion primitives. This test intends to show the errors in the odometry system. We can observe the results for this test in Figure 9. In this graph, we can observe how the execution of the square path is affected gradually by the errors in the motion primitives (mainly at the points where a rotation is needed).

Odometry error is computed as the difference between the real position of the robot (only known by the simulator) $\xi_{\text{abs}}$ and the position computed by the robot for itself $\xi_{\text{calc}}$ [8]. We compute then the average error of the odometry system. For this specific routine, and for illustrative purposes, the error is $\bar{\varepsilon}_\xi = \{5.9728 \ \text{mm} \ 5.5326 \ \text{mm} \ 0.0124 \ \text{rad}\}^T$. From this data, we conclude that there is a good approximation to the real position the odometry using numerical integration.
6. Conclusions

In this article, we have discussed the components of a mobile robot simulator when implemented using a game development engine. We have presented the advantages of using this kind of development environments for this task. We have also presented several available platforms, both commercial and open-source and we have briefly analyzed their pros and cons for mobile robot simulation tasks.

We have described the elements of the developed mobile robot simulator and we have presented its implementation details, including the kinematics model of the robot.

We have verified the correct operation of the implemented components, including the graphical environment needed to see a graphical output of the simulation. We have performed motion tasks based on a set of motion primitives. We have observed also the effects of the model discretization, These problems are inherent to each mobile robot simulation.

We will continue to improve all the simulated robot subsystems. We will also work towards developing multi-robot task to validate this functionality in our simulator.

Acknowledgements

Uriel H. Hernandez-Belmonte thanks the funding of its work by Consejo Nacional de Ciencia y Tecnología (CONACYT) through the Grant No. 329356/229784.

References