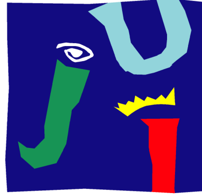

Technical Report ICC 2011-10-01

DESIGN OF HIGH QUALITY, EFFICIENT SIMULATION ENVIRONMENTS FOR USARSIM

Jaime Alemany

October 2011

Department of Computer Science and Engineering

Universitat Jaume I
Campus de Riu Sec, s/n
12071 – Castelló
Spain

http://repositori.uji.es

Abstract

This report describes how USARSim and MATLAB have been combined with some graphic design tools such as 3D Studio Max and Adobe Photoshop producing a personalized simulator which combines a high level of detail with a real time computing speed and a flexible interface for users. What is achieved in this way is that students of robotics of the University Jaume I in Castelló have at their disposal a tool that allows them to carry out virtual experiments with robots, whose satisfaction level is similar to that of a real test at the university.

Keywords:

- 3d mobile robot simulator
- USARSim environment design
- virtual lab

INDEX

1. INTRODUCTION 3
2. OJBECTIVES 7
3. STATE OF THE ART 8
4. METODOLOGY 9
4.1. Design 9
4.2. Dynamics Diversification 11
4.3. Testing 14
4.4. Programming 15
5. TOOLS 18
5.1. Unreal Tournament 2004 and USARSim 18
5.2. 3D Studio Max v 9.0 19
5.3. Adobe Photoshop CS2 20
5.4. Unreal Editor (UnrealEd) 21
5.5. MATLAB R2007b 22
6. DEVELOPMENT 23
6.1. Virtual robot creation 23
6.1.1. Drawings and measures 23
6.1.2. Pictures 23
6.1.3. 3D Robot 24
6.1.4. Unreal Editor 29
6.1.5. Virtual robot programming 30
6.2. Creating a virtual world 38
6.2.1. Drawings and measures 39
6.2.2. Pictures 40
6.2.3. 3D environment 40
6.2.4. Lights 59
6.3. Movable objects 63
6.3.1. Soccer ball 64
6.3.2. Candy 66
6.3.3. Punching bag 68
6.3.4. Toy dispenser 72
6.4. MATLAB functions 73
7. RESULTS 73

8. CONCLUSIONS AND FUTURE WORK 76
9. REFERENCES 77

1. INTRODUCTION

The tests with robots realized in the frame of the investigation involve several familiar problems, derived from both the physical characteristics of the available robots and the environment of the experiment.

On the one hand, the robots leading the tests need to be configured again and again in order that the advances consequence of those essays can be incorporated. In this way, it is common there to be continuous operations of assembly or substitution of sensors, cameras and actuators of all kinds which have to be acquired in the case they are not at disposal, with the consequent cost in time and money.

Similarly, the spaces needed for such experiments are not always available and, when they are, a multitude of retouches are required to get them adjusted to the changeable conditions of the ongoing investigation.

A good solution for these problems consists of moving the whole experiment to a virtual world where everything is always available and with a reduced cost of time and money [1]. To this end several simulators for robots have been developed in either 2D (Stage [2], MobileSim [3]) and 3D (Gazebo, Simbad [4], Webots [5]), but most of them fail to capture the interest of users, mainly due to their lack of realism. In other words, the graphical quality of the virtual environment turns out to be insufficient to allow the test to be easily followed and convincing for the researcher and, in addition, this fact makes it more difficult or even impossible to use the simulator for vision-based software development.

Unreal Tournament 2004 (UT2004) is a commercial first person shooter video game with a powerful physics engine (Epic Games' Unreal Engine 2.5) built for graphical realism and smooth gameplay [6]. USARSim is a high fidelity 3D robot simulator built on top of Unreal Engine 2.5, which provides detailed models with high quality physics of interaction allowing accurate simulations [7]-[11]. Besides, USARSim is an open source software that let users to build their own environments and robots. These reasons have led us to be part of the USARSim's active development community.

Figure 1 shows the ERA robot (ERRATIC Robot developed by Videre Design) moving inside the TI building of the University Jaume I of Castellón during a test with the simulator developed in this work. This picture can be compared with a photograph taken with the same angle corresponding to a real experiment (figure 2).

Fig. 1. ERA robot during a simulation in the TI building.

Fig. 2. ERA robot during a real experiment in the TI building.

Figure 3 and 4 show images of classical simulators with low level of graphics realism. On the one hand, in figure 3 can be observed a screenshot corresponding to the Stage 2D simulator, which distinguishes the path of three robots moving through the classroom building at the Superior School of Technology and Experimental Sciences of the University Jaume I in Castellón. The result is not always easy to interpret. On the other hand, the figure 4 shows an image corresponding to the Gazebo 3D simulator in which a Pioneer P2AT robot can be distinguished while focusing with his camera a spherical form in front of a green background. In view of these images, there can be no doubt that 3D simulators offer a greater degree of clarity when informing a researcher how his experiment is developing. In this respect, Webots provides a step forward, offering high quality 3D graphics, but for the educational community it has, by contrast, the disadvantage of being a closed source commercial simulator.

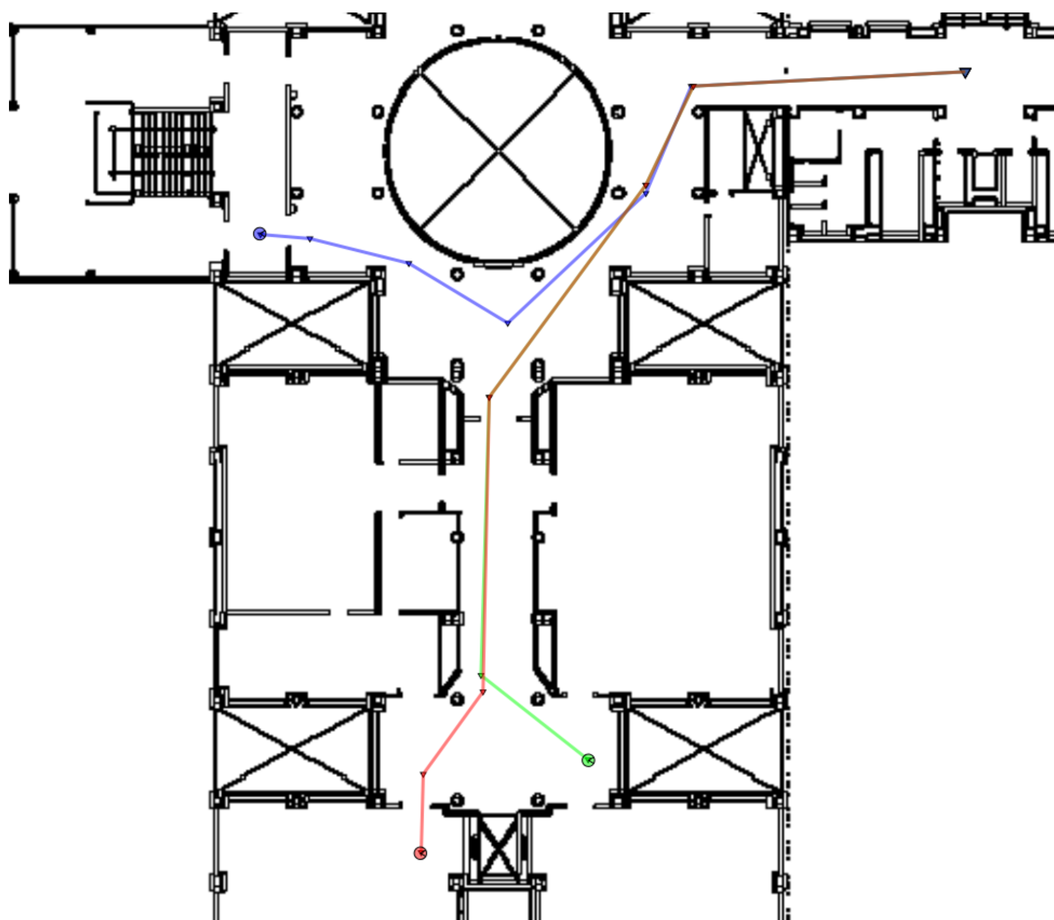

Fig. 3. Player Stage 2D robot simulator.

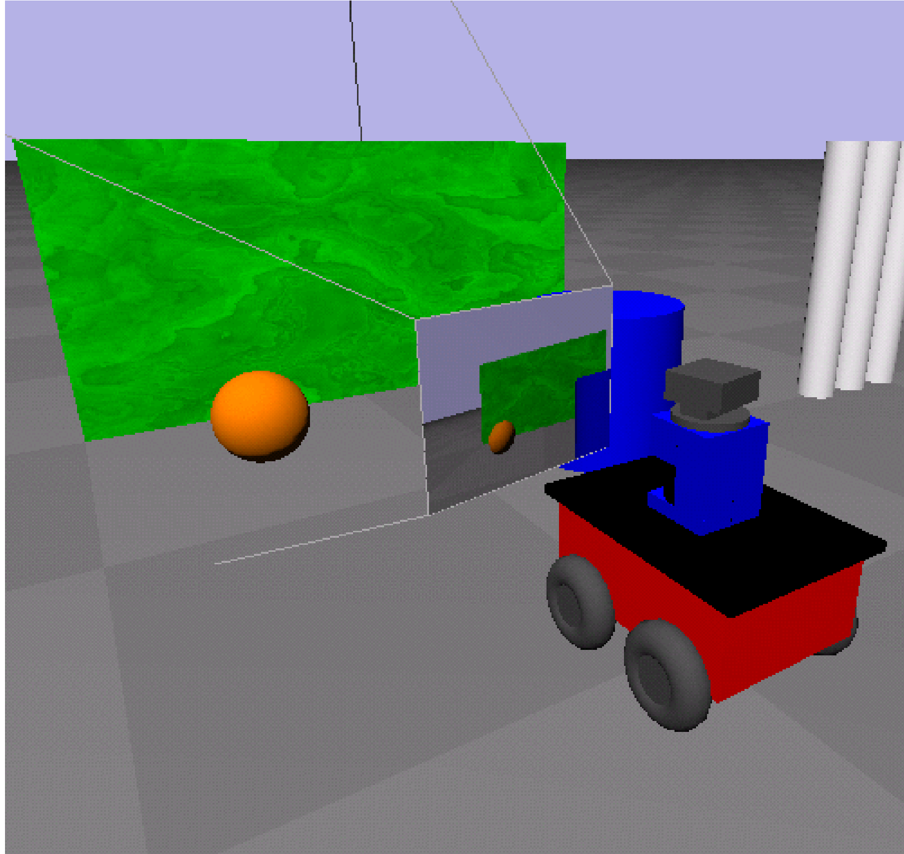

Fig. 4. Gazebo simulator showing a Pioneer2AT.

In conclusion, this work tries to improve the image and sense of satisfaction that most current simulators offer, taking advantage of the enormous graphical and simulation capacity of the current video games like Unreal Tournament 2004, as well as the increasing computing speed of machines, offering a powerful tool for the educational and research community.

2. OJBECTIVES

The objective of this work is to obtain a high fidelity 3D simulator using custom models of environments and robots and combining testable and reliable experiences with real time execution speed.

On the one hand, it must be obtained a 3D model of a robot with the appropriate characteristics in shape, texture and physics.

On the other hand, it is required a 3D environment rich with different scenarios and levels that allow varied experiments.

In addition, for the experiment to be testable, both the robot and its environment must have a match in the material world. This has been possible by modeling a robot available in the Intelligent Robotics lab at the University Jaume I and the whole building containing the latter and its immediate surroundings.

Finally, it is also important that the simulator provides a powerful and flexible interface suitable for researchers and students in robotics. In this sense, it was decided to use MATLAB and its package of features for USARSim.

3. STATE OF THE ART

More than six years ago appeared USARSim as a simulation tool linked to the 3D engine of Unreal Tournament (UT 2003). Since then, researchers around the world have made use of this tool for developing a multitude of works for the educational community with different purposes, but with one common characteristic: all of them have benefited from the huge graphics and simulation capabilities of this software to move their experiments to a virtual world, obtaining a configurable, infinite, economical and always available testing field. So, we can find lines of research in the field of psychology of human-robot interaction [12], learning models in recognition of the environment [13], objects and people [14] with vision systems, sonar or laser, simulators for unmanned aerial vehicles [15], frameworks for the development of robotic games [16]...

This research aims to go one step further in graphical quality of models, trying to gain in realism while maintaining a high computing speed. In this way it will be possible to improve the reliability of vision-based experiments, while increasing researchers' satisfaction.

4. METODOLOGY

The procedure followed to move experiments with robots to a virtual plane can be divided into the phases of design, dynamic diversification, testing and programming, each different stage needing the use of distinct software tools as shown in the flow chart of the figure 5.

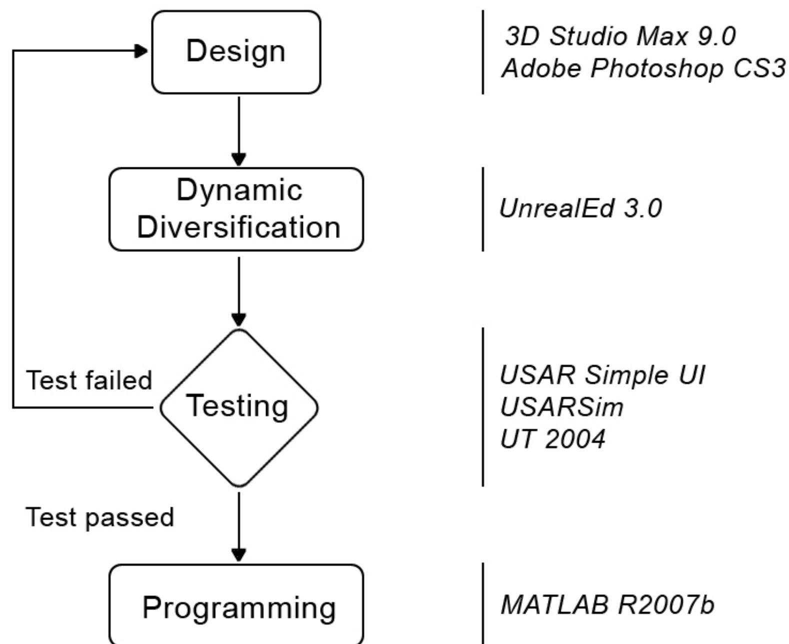

Fig. 5. Flowchart and software used during the simulator development.

4.1. Design

Firstly, the 3D modeling of the TI building, the ERA robot and of some movable objects has been carried out by using 3D Studio Max 9.0 (see the figure 6) [12]. The TI building is important for this simulator because it contains the Department of Engineering and Computer Science and some robotics laboratories. The dimensions and shape of every element have been defined from drawings and measurements, whereas the textures, which give the appearance of color and material, have been obtained by means of photos treated with Adobe Photoshop CS2.

Secondly, some collision hulls have been added to every object in order to define the limits of their contour when coming into contact with other objects or surfaces. These envelopes have been defined from one or more simple geometric shapes, such as a box, a cylinder or a sphere, simplifying the physical behavior of objects and improving the simulator performance.

Thirdly, each object together with its collision hull has been exported from 3D Studio Max 9.0, generating an ASCII file with extension .ase
This is the recommended format for UnrealEd can import 3D objects created by other applications.

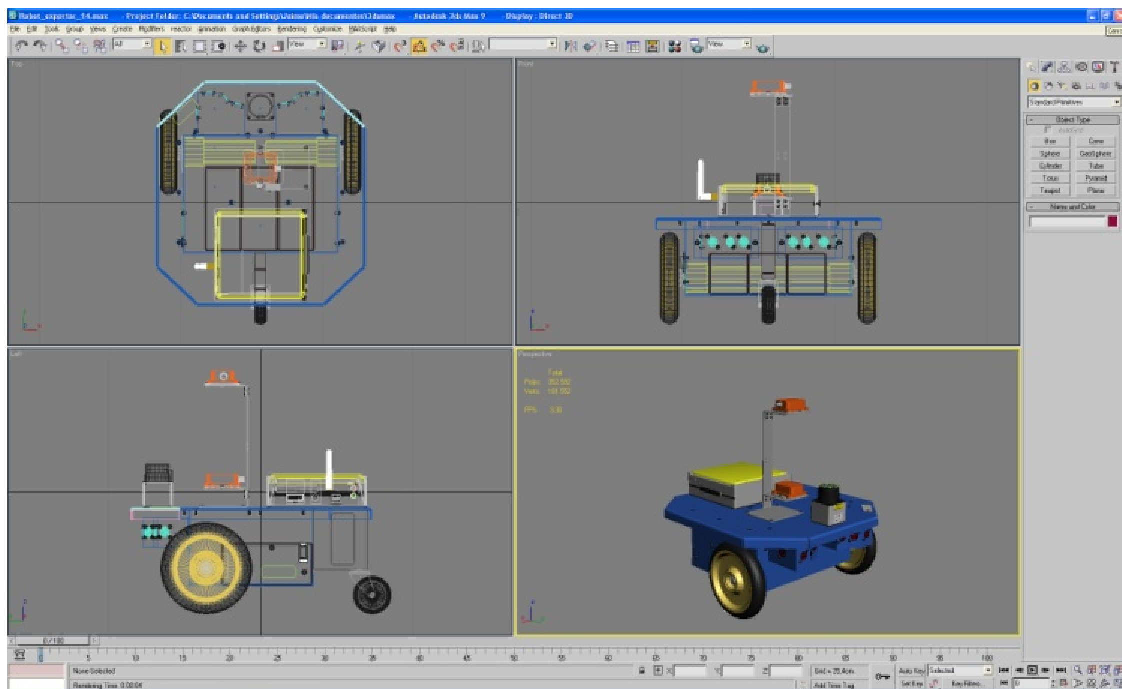

Fig. 6. ERA robot design phase using 3D Studio Max 9.0.

Finally, from UnrealEd 3.0 textures have been imported first and then the objects, generating texture libraries and sets of elements, which have been gathered as they belong to the ERA robot, to movable objects or to the static environment. UnrealEd is the level editor used to create levels for Unreal Tournament.

4.2. Dynamics Diversification

At this stage all the 3D objects have been adjusted according to the nature of their dynamics.

The static environment consists of motionless objects such as walls, floors, sidewalks, windows and some doors. From UnrealEd 3.0 these objects have been placed in their corresponding locations as Static Mesh, generating the TI building and its surroundings. Static Mesh is an UnrealEd term used to refer to objects that cannot be moved. Later, their graphic parameters concerning lighting have been adjusted to achieve a suitable appearance. The doors provided with motion, by contrast, have been placed in the virtual building as Karma Actor [13] or Mover [14] according to their mode of interaction, defining the parameters that govern their physical behavior. Karma Actor and Mover are both UnrealEd terms, the first one is used to refer to objects whose motion is defined by the Karma Physics Engine and the second one is used to call objects that can move between predefined positions and rotations. These doors also need some axes of rotation to restrict their movements.

The movable objects, with which robots can interact, have been treated similarly to those doors provided with motion, but these have been included in a set named Toys. These are objects with simple geometric shapes and varied physics and dynamics which have been created with a double aim: to be used for grasping or vision-based recognition experiments; to serve as the basis for developing any other object required by the educational community. Figure 7 shows some of these objects.

Fig. 7. Movable objects for robot interaction.

With the static elements and the movable objects in their right places, some points of light have been entered so that the whole virtual environment remains properly illuminated and offers a sensation of realism to the user.

3D models in ASCII format corresponding to robot parts have been imported from Unreal Editor and, later, classified in a file called ERARobot, which contains groups of elements according to their function or location within the robot. Then, each one of these elements has been defined as a USARSim recognizable class, placing the resulting files in the folders listed in Table I.

USARBot Classes	USARModels Classes	USARMisPkg Classes
ERA ERARangeSensor ERASonarSensor USAR_ERALTire USAR_ERARTire USAR_ERASmallTire	ERAComputerBody ERALasser ERALTire ERARTire ERASmallTire	ERAComputer ERAComputerBody

Table I. USARSim ERA Robot Classes.

Afterwards, the files USARMisPkg.ini and USARBot.ini have been modified from the System folder at UT2004 directory created when installing the game Unreal Tournament 2004 and then all classes have been compiled. The file USARBot.ini contains information about those robots used by USARSim and,

therefore, it has been necessary to insert a few lines to add the ERA robot to this list. Below are displayed the lines that have been entered in the USARMisPkg.ini file, in a similar way, in order to define the computer on board of the ERA robot (ERAComputer) as an optional accessory consisting of only one element named ERAComputerBody.

```
;------------------------------------------------------------------------------------------------------------------
; Computer mission package used for the ERA
;------------------------------------------------------------------------------------------------------------------
[USARMisPkg.ERAComputer]
Links = (LinkNumber = 1, LinkClass = Class'USARMisPkg.ERAComputerBody', DrawScale3D = (X=1.0, Y=1.0, Z=1.0), ParentLinkNumber = -1, SelfMount = "A")
;------------------------------------------------------------------------------------------------------------------
; Computer Links used for the ERA
;------------------------------------------------------------------------------------------------------------------
[USARMisPkg.ERAComputerBody]
MountPoints = (Name = "A", JointType = "Revolute", Location = (X= 0.0, Y=0.0, Z=0.0), Orientation=(X = 1.5707963267948966192313216916398, Y = 0, Z = 0))
MountPoints=(Name="B",JointType= "Revolute", Location = (X= -0.0, Y = 0.0, Z= -0.0), Orientation=(X=1.5707963267948966192313216916398, Y=0, Z=0))
MaxSpeed=0.1745
MaxTorque=20
MinRange=0.0
MaxRange=3.14159
;------------------------------------------------------------------------------------------------------------------
```

This stage ends up by editing the usar_s.bat file in the mentioned System folder and changing the name of the current map by the one obtained by means of UnrealEd 3.0.

4.3. Testing

The phase of testing has served to adjust and optimize all the work developed during the previous stages.

Regarding the 3D models and the number of polygons that they are made up of, many experiments have been realized with virtual objects in different degrees of approximation to their real shape until obtaining high graphical quality models and real time execution speed. In order to avoid interferences produced by high resources consuming software, these tests have been carried out by using the simple interface under Windows that provides USAR_UI (see figure 8), taking as a reference the computational speed offered by a laptop with the following characteristics:

Processor:	Intel Core Duo T2300 1.66 GHz
RAM Memory:	2 Gb 667 MHz
Video Card:	NVIDIA GeForce Go 7400

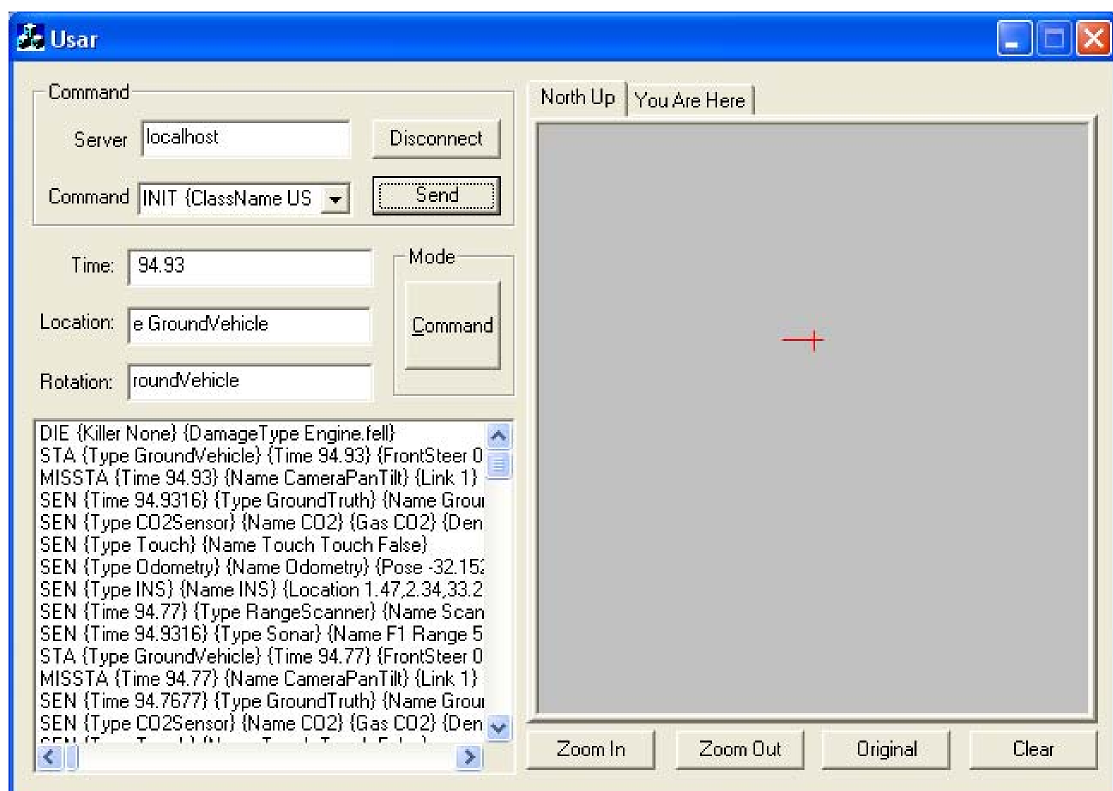

Fig. 8. USAR_UI user interface.

Concerning the elements provided with predefined movements such as some doors, during this phase has been redefined their speed, path and, in some cases, their behavior in the presence of robots or any other event that activates those movements.

Likewise, the physical parameters of the movable objects have been adjusted again and until suitable dynamics were obtained.

Similarly, the ERA robot has been the object of numerous additional tests with various objectives such as verifying the correct positioning of all its elements and sensors, confirming the readings of these ones, examining the images captured by its camera, checking the maximum speed and torque, etc.

Moreover, obtaining a good lighting in the virtual environment has led to many trials with a high cost in time, since it takes UnrealEd 3.0 several minutes to calculate the quantity of light that affects each one of the multiple surfaces that conform the TI building and its surroundings.

4.4. Programming

Once adjusted both environment and robot, the next step was to set customized functions to provide users with the capability of controlling one or multiple robots as well as the information flow perceived by their sensors and cameras.

This part of the work has been developed by means of MATLAB R2007b, turning this tool into a suitable interface between the user and USARSim due to a series of reasons shown bellow.

- MATLAB allows functions for simulating mechanical systems to be created, generating graphical interfaces, achieving mathematical calculations, acquiring information and data from different hardware systems and, farther, managing databases.

- There exists a software package called MATLAB USARSim ToolBox [15], which combines MATLAB's functions (m files) with other functions Java to generate an interface between the user and USARSim.
- The final aim of this work is to serve to scientific community and, therefore, it is very important to choose an interface that is familiar to its members such as MATLAB.

Flow chart in Figure 9 shows the process followed to obtain the desired simulator.

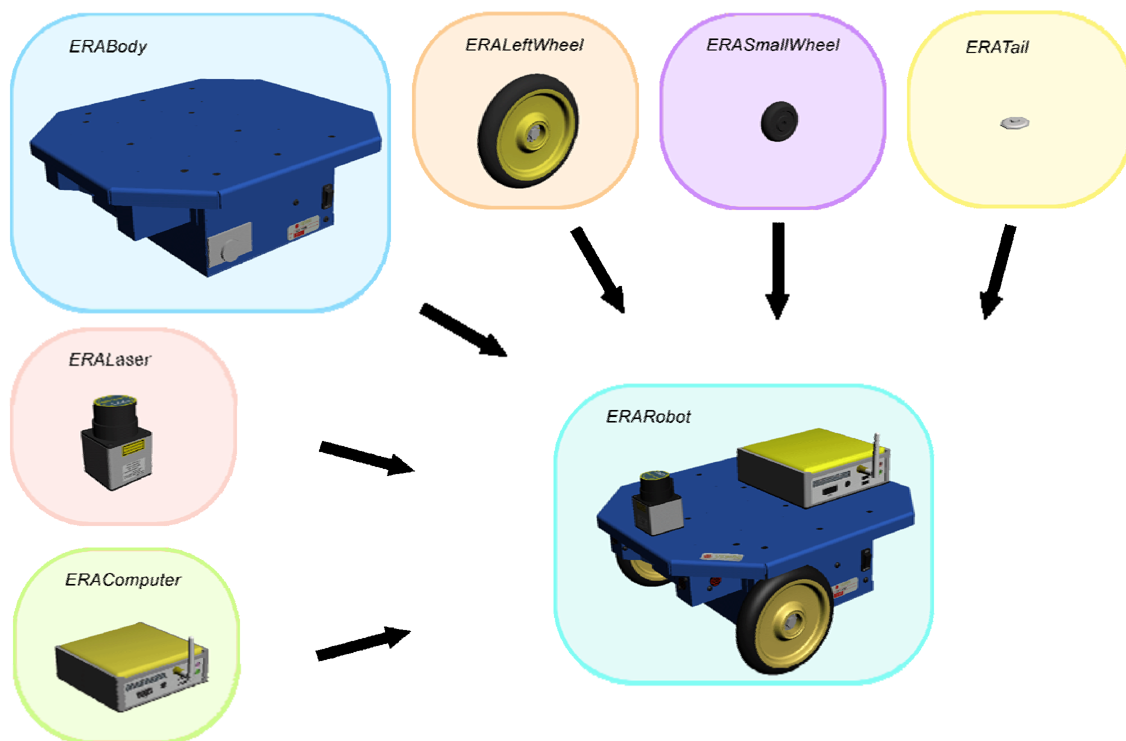

Fig. 9a. Classes flowchart.

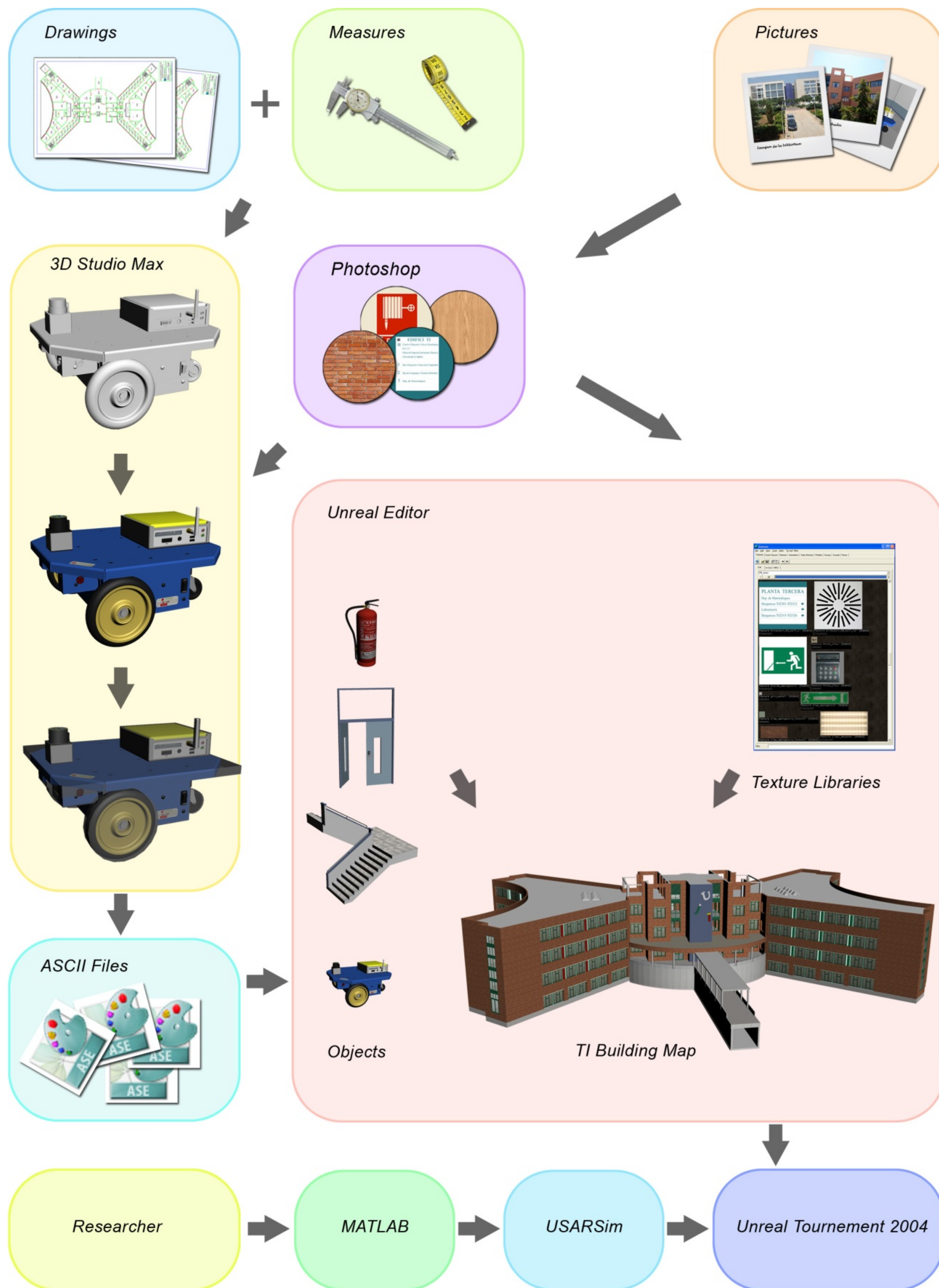

Fig. 9b. Methodology flowchart.

5. TOOLS

During the process of creating, programming and testing of the 3D environment and robot it has been necessary to use different software applications to solve the various problems that this task entails. Here are the tools that have stood out as very useful or essential.

5.1. Unreal Tournament 2004 and USARSim

In the first place, it is important to highlight the importance of the couple of tools Unreal Tournament and USARSim.

Work on USARSim began in late 2002 under an NSF ITR grant to study Robot, Agent, Person (RAP) teams in Urban Search And Rescue (USAR). As work was beginning, Epic Games released Unreal Tournament 2003, a first person shooter (FPS) video game available for multiple platforms and operating systems, whose Karma Physics engine was soon seen as a powerful simulation tool [17].

Currently, the Unreal Tournament 2004 engine (Unreal Engine 2.5) integrates Karma Physis SDK, offering high-quality graphics and high realistic simulation of physical systems, while maintaining a smooth real time data transfer. Unreal Engine 2.5 also incorporates other features that differentiate it from other contemporary engines: it has an object-oriented design, has a scripting language similar to Java and, moreover, Epic Games has developed a scripting language for allowing the introduction Unreal modifications, providing the scientific community make the Unreal Tournament engine a research tool. So, USARSim can be understood as a modification of Unreal tournement 2004 that incorporates a generous amount of prebuilt robot platforms, sensors and levels. Robot, sensor, and map models are inputted and compiled into UT2004. The UT2004 networking is proprietary, however, with the University of Southern California’s GameBots a protocol allows client controllers access to characters

in UT2004 via network sockets. The MATLAB USARSim toolbox connects to UT2004 through this protocol. The Unreal Client is used to generate video from the game. Using the multiview feature, multiple robot cameras can be captured and displayed on the screen. An image server can grab the video from the Unreal Client through a DirectX hook.

5.2. 3D Studio Max v 9.0

The environment where virtual robots will move around is the TI Building at University Jaume I of Castellón, which has an X-shaped section and five floors. The complexity of this environment suggests to use specialized software applications in order to carry out an accurate modeling. In this sense, it has been chosen Autodesk 3D Studio Max for this job. This is a widely used tool for modeling objects and recreating all kind of 3D scenes such as characters, objects or vehicles for video games, animation films, simulators of all kinds, computer graphics for architectural projects, etc … This feature has allowed 3D Studio Max to have very good ability when opening, importing, exporting or, in general, processing data from many applications necessary to complete the modeling work. Here are the main reasons why 3D Studio Max is ideal for the purposes of this paper.

First, we used the floor plans and elevation of this building as the foundation for its development. These drawings have been created using Autodesk's AutoCAD and their source files in dwg format are available on the OTOP website. 3D Studio Max v 9.0 offers full compatibility for data import in dwg format and can treat them like any other object created with 3D Studio Max, retaining all its original properties as measures, dimensions, etc...

Secondly, it is known that 3D models require textures that help to define the material they are made. 3D Studio Max lets you apply textures to objects in many formats and, more specifically, the Targa. tga 24-bit required by UnrealEd (level creator for Unreal Tournament). This will be essential during the development stage.

Third, this tool allows exporting objects in ASCII format with extension. ase. This type of file it is necessary for UnrealEd to be able to import 3D models while retaining all its properties of form, texture and collision hull.

Finally, it is important to note the tremendous ability of 3D Studio Max when creating any type of character, object or environment in three dimensions, as well as its ease of use.

5.3. Adobe Photoshop CS2

As it can be learned from the previous section, this work has required modeling a wide variety of objects made of different materials and in different colors. Once built the 3D model of any such objects, the next natural step is to apply a distribution of textures that faithfully simulate those colors or materials.

Textures can be obtained in several ways: searching through computer files or on the web, scanning images or taking photographs. For the development of this work have been used a lot of digital photographs and only two textures from images obtained from the web. In any case, it is not usual to found textures that do not require certain changes before being applied to a 3D model, which leads to the need of using photo editing software for this task. In addition, the Unreal level editor (UnrealEd) only supports textures in 24-bit Targa format file (. tga) and its pixel dimensions must be powers of 2, features very restrictive if you're looking for textures ready to use.

To prepare all necessary textures for the virtual environment of the TI building as well as robots and other objects, we have chosen the Adobe Photoshop CS2 due to a number of reasons listed below.

Adobe Photoshop is the most commonly used graphic design tool, allowing image files exchange with many graphics applications, and therefore, with a huge variety of formats, among which is the Targa.

Finally, it is important to note that more than 100 textures have been created to complete this work and no one of them has required a transformation that could not be done by this application.

5.4. Unreal Editor (UnrealEd)

USARSim uses the Unreal Tournament 2004 graphics engine to carry out the simulation of robots moving through virtual environments. Thus, it is absolutely necessary to use the map editor of Unreal Tournament (Unreal Editor 3.0) to build a virtual environment compatible with this engine.

Unreal Editor 3.0 allows creating high quality virtual objects and maps, offering a very complete working environment, similar to many software applications for 3D design. Although Unreal Editor cannot possibly create objects with so much detail as other applications like 3D Studio Max, what makes it different from other 3D design software is its huge capacity to simulate the behavior of mechanical systems in virtual environments, requiring the setting of a reduced set of parameters.

To facilitate the incorporation of models of objects created by other applications to their maps, Unreal Editor can import them easily admitting very common formats like. obj and. ase, being Studio 3D Max the recommended source for this purpose. This capability ensures the quality of the models used by Unreal and has been widely exploited in this work to the point that almost all of its components are modeled in 3D Studio Max.

5.5. MATLAB R2007b

With all the elements necessary for virtual testing completed, we decided to use MATLAB as user–USARSim interface, due to a series of reasons detailed below.

On the one hand, it is known throughout the scientific community the enormous power of this tool to perform mathematical calculations, simulating mechanical systems, generating and managing databases, creating custom functions and interfaces for each user and acquiring data from various systems and hardware.

On the other hand, MATLAB has been for many years a widely used tool for students and researchers from around the world. This is expected to increase the number of potential users of this virtual environment as well as and their ability to customize it according to their purposes.

In addition, as discussed in paragraph 3.1 of this work, there is a set of functions grouped in the package MATLAB USARSim ToolBox optimized for simulations of robots with USARSim. These functions allow commanding one or more robots and acquiring all data captured by their sensors, including images from their cameras.

6. DEVELOPMENT

The procedure for moving the robot tests the virtual plane consists of several phases which, in turn, depend on which item you try to virtualize: a robot, an inanimate object or environment.

6.1. Virtual robot creation

The Department of Engineering and Computer Science at the University Jaume I has 6 units ERA robot. Since its acquisition, these robots are doing many experiments to provide information to different lines of research. To expedite such testing and, in turn, make them more accessible we have chosen this type of robot.

6.1.1. Drawings and measures

To obtain a virtual model of the ERA robot it has been necessary to have all the dimensions that determine the size of all its elements. In this sense, we have proceeded to collect all documents containing such information. However, in the documents consulted we have not found all the dimensions that define the geometry of the robot, requiring some measurements on one of the available robots to be done.

6.1.2. Pictures

Once collected all the dimensions that give form to the 3D model of the robot, we must focus on the color and textures that will give real look at those materials that compose it. With this objective we have taken photographs of

different parts of the real robot, until obtaining a representative sample from each of the materials in it.

The photographs have been treated with Adobe Photoshop CS2 to get the desired textures, which must meet two additional requirements to be used by UnrealEd: UnrealEd 3.0 only allows textures to be imported in 24 bit TARGA format (.tga file) and, moreover, their dimensions in pixels must be powers of 2.

6.1.3. 3D Robot

With the dimensions and textures of the robot it begins the development phase of the virtual model using 3D Studio Max. Figure 10 shows the 3D model of the ERA robot during its design in 3D Studio Max 9.

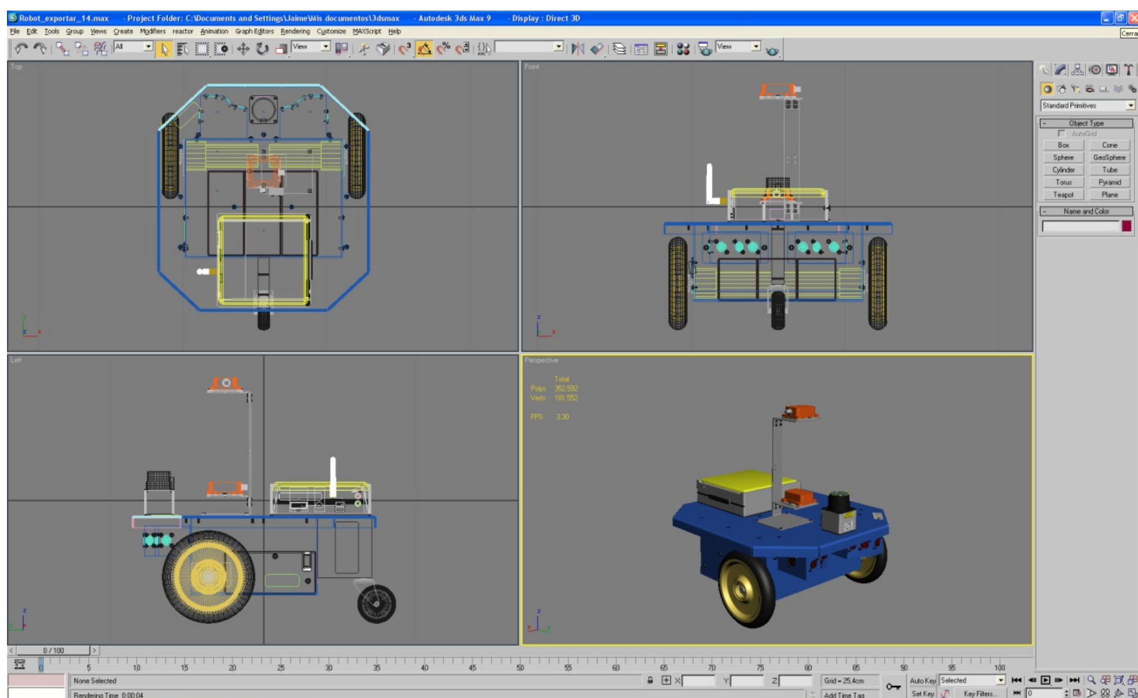

Fig. 10. Robot model in 3D Studio Max

6.1.3.1 LowPoli approximation

First we built the 3D model of the robot from its size, claiming that it be as close as possible to the real one without losing computing speed during the simulations. This was achieved by approximating the shapes of the robot to sets of geometric primitives, which shall be composed of a number of polygons or faces that is not too high and therefore consume a large amount of computation time. At this point, the 3D robot consists of several sets of elements to be separated or grouped to form independent entities according to the type of object they constitute. For example, sonar, laser, wheel, chassis... are separate entities to be treated differentially. In general, any element that has an individual job or can describe a relative motion respect the others must be regarded as independent.

It is important to note that the map editor UnrealEd for Unreal Tournament 2004 requires that imported objects consist of a single mesh with all its textures distributed according to a UVW map layout. In addition, these textures should be properly grouped and stored in a file with .utx extension. Figure 11 shows one of these .utx files containing the textures of the ERA robot laser.

Second, each one of these separate elements has become a unique mesh. This operation removes any prior geometric information and converts the 3D object into a set of polygons defined by triangular faces. As stated, the greater the number of triangles, the greater the resemblance of the model with the real thing, but also the greater computational load. It is therefore necessary to reach a degree of reality simplification to solve the compromise between graphic quality and speed of simulation.

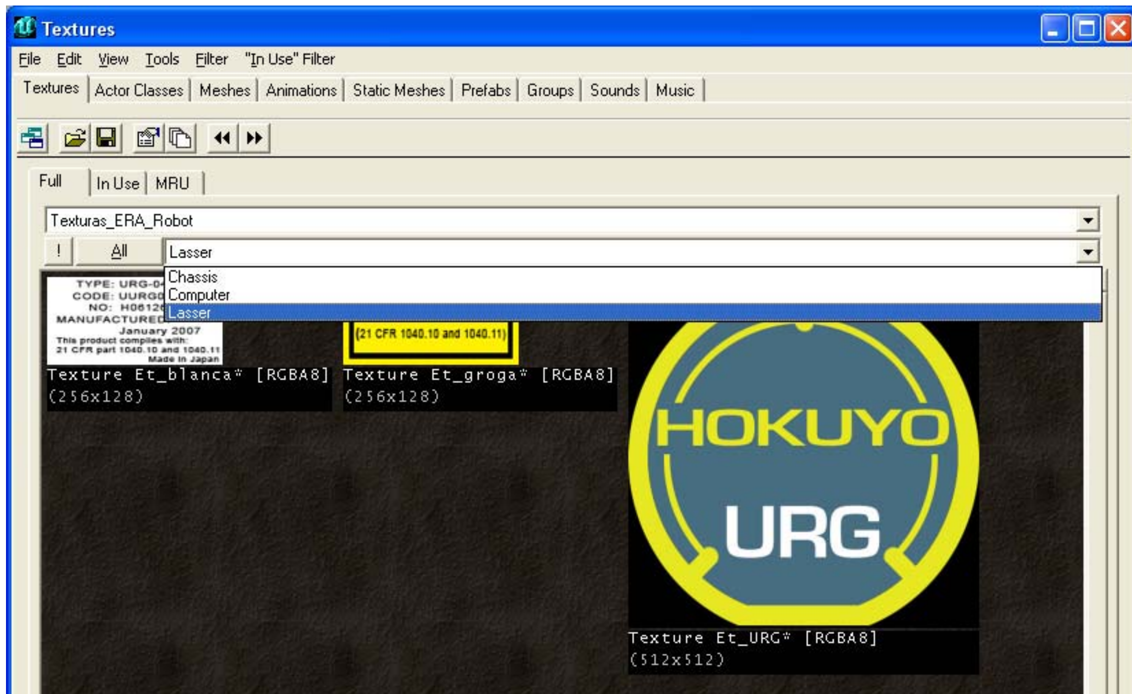

Fig. 11. UnrealEd texture browser showing some ERA laser textures.

6.1.3.2. Material Multi-Subobject

Then, with the small robot parts converted to mesh, we have proceeded to provide them with the corresponding textures. Most of the component parts of the robot have more than one color or texture and, in turn, can only be a single mesh. This requires the use of multi-subobject materials when working with 3D Studio Max, which can consist of as many textures as desired.

Thus, it has been created a suitable material for each independent element (mesh) of the robot and its distribution has been defined on it by a UVW map, as required for proper textures import by UnrealEd. Figure 12 shows a screenshot of 3D Studio Max taken when applying its corresponding multi-subobject material to the ERA robot laser mesh.

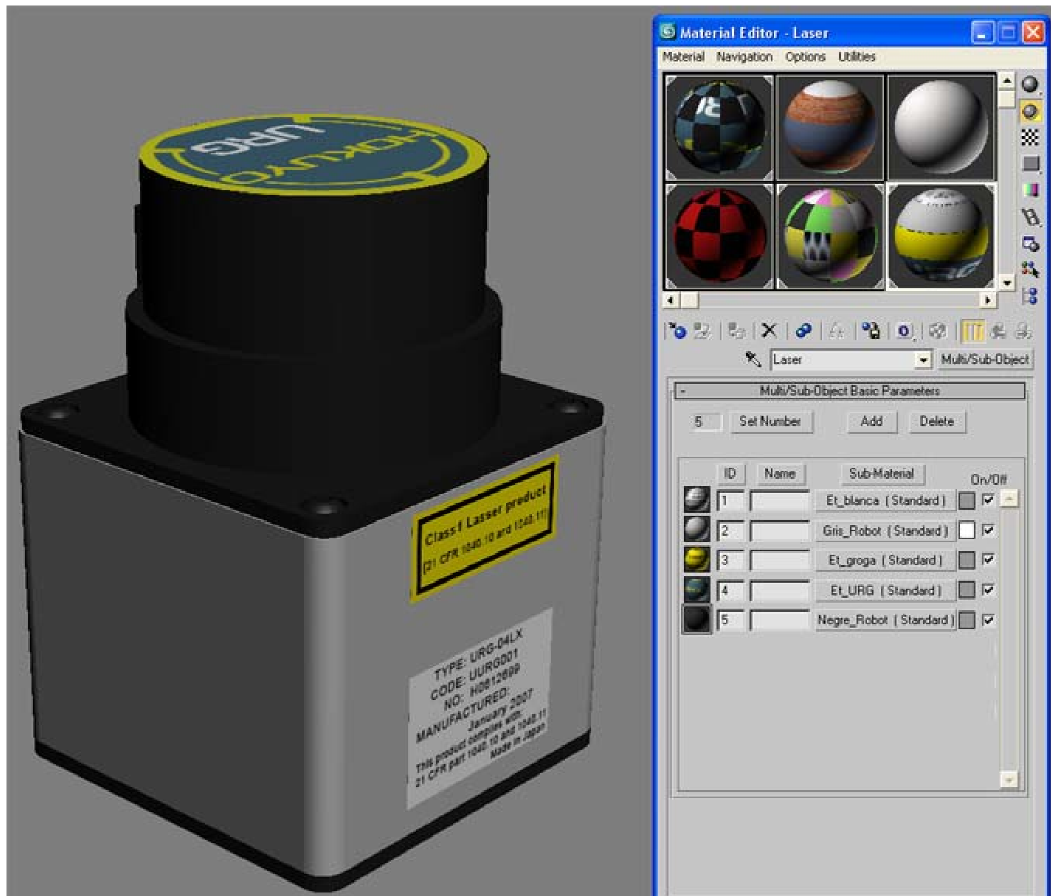

Fig. 12. ERA robot laser and its multi-subobject material in 3D Studio Max.

6.1.3.3. Collisions

Once finished the physical appearance of the elements that compose the robot, they must be provided with an envelope that defines their collision outline. Otherwise, the robot will pass through all surfaces in its path, falling for ever because it cannot even collide with the floor.

UnrealEd allows assigning these collision envelopes not only to objects created using this program, but also to those imported in. ase format. However, there is a problem when using UnrealEd to define the collision hulls of imported objects, because those envelopes must whether match the mesh of the object or have very simple geometries (cubes, cylinders, spheres...). In the first case, the

computation time is usually very high, because the resulting envelope will contain a large number of triangles or faces. In the second case, we get a high computing performance, but it is not easy to set one simple geometric shape as a collision envelope. In the figure below you can see how a simple object like the ERA robot's computer takes several primitives to define acceptably its collision outline.

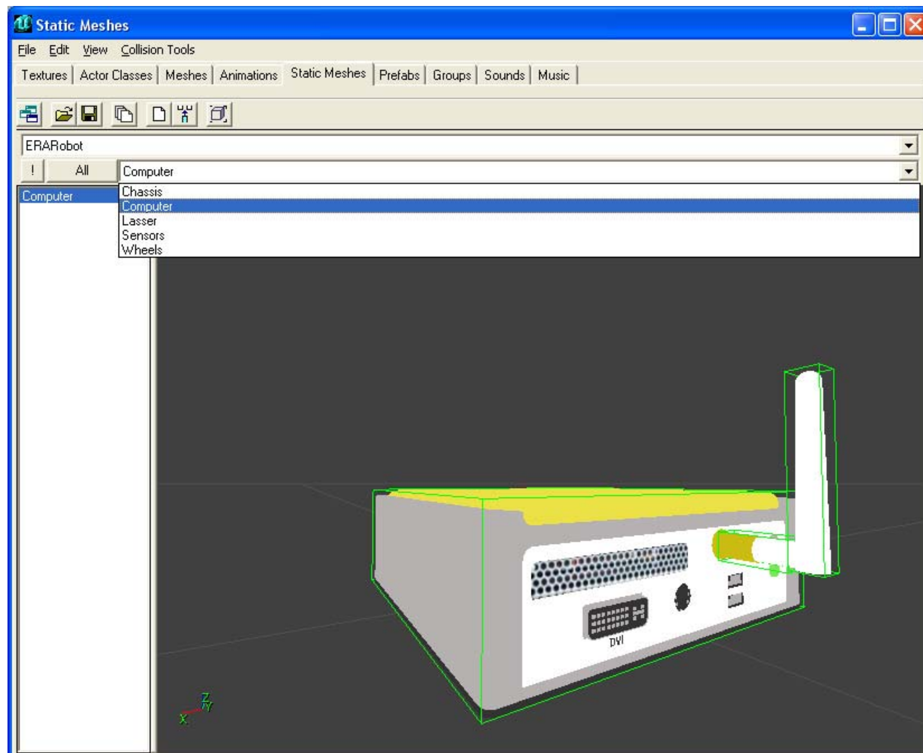

Fig. 13. ERA robot’s computer collision hulls.

This being the case, it was decided to use 3D Studio Max to define the collision envelopes of the items created with this program. Working with 3D Studio Max, the assignment of this type of outlines is very easy due to several reasons:

First, and since the object is being created and with 3D Studio Max, there is no need to switch applications and envelopes can be created like any other shape. We can even make use of those primitives (simple shapes) that were used to create the object itself, which already have the appropriate shape and location.

Secondly, 3D Studio Max can export the collision envelope along with the modeled object. You only need to go to the properties menu of the primitives that define such envelopes and name them correctly (see table II). Once that is done, select the object's mesh and those of the envelope and export them as an ASCII file . ASE with the name you want.

Collision Hull Name	***Primitive***	***Example***
MCDBX_Name	*Box*	*MCDBX_ComputerChassis*
MCDSP_Name	*Esphere*	*MCDSP_ERAWheel*
MCDCY_Name	*Cylinder*	*MCDCY_Columna*
MCDCX_Name	*Convex Mesh*	*MCDCX_Extintor*

Table II. Collision hull names.

6.1.4. Unreal Editor

With all the robot parts completely modeled in 3D Studio Max and exported to ASE format, it is time to create a file that lets you store them in order to be accessed and understood by Unreal Tournament 2004 engine. To achieve this goal, we followed the following steps:

1. From UnrealEd Static Mesh Browser, we imported the .ASE file corresponding to the part of the robot we wanted to save, by defining a suitable group for it (body, camera, wheels ...).

2. Once imported and distributed in groups all the elements that make up the robot, we saved a static mesh file in .USX format.

Following this procedure we obtained the file ERARobot.usx, which contains all the information of the appearance and collision hull of the robot, required to perform simulations with USARSim. In the figure below you can see the ERA

robot's chassis, which was stored in the group *chassis* of the above mentioned file.

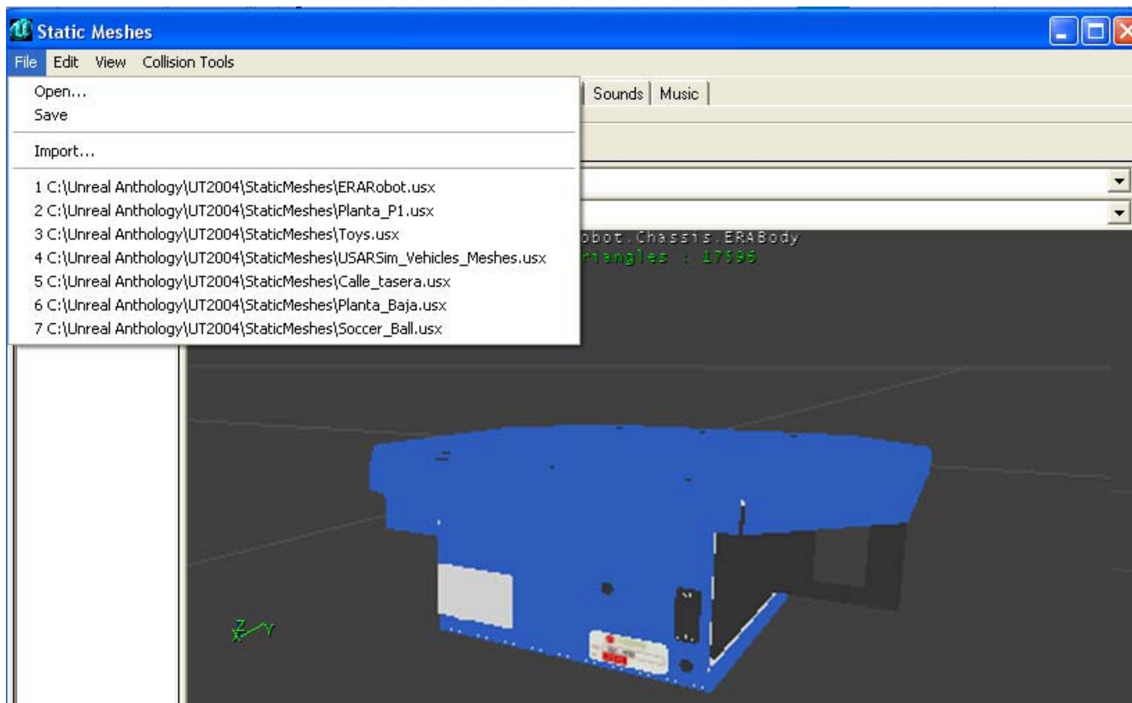

Fig. 14. UnrealEd Static Mesh Browser showing ERA robot's body.

6.1.5. Virtual robot programming

6.1.5.1. Classes

Once the 3D model of the robot is obtained and stored it in a .USX file, the next step is to generate a series of files .UC to define classes for each part of the robot. To define the ERA robot, it has been necessary to have the following class hierarchy:

USARBot Classes	*ERA.uc*
	ERARangeSensor.uc
	ERASonarSensor.uc
	sensor.uc (mirar si es una modificación)
	USAR_ERALTire.uc
	USAR_ERARTire.uc
	USAR_ERASmallTire.uc
USARModels Classes	*ERAComputerBody.uc*
	ERALasser.uc
	ERALTire.uc
	ERARTire.uc
	ERASmallTire.uc
USARMisPkg Classes	*ERAComputer.uc*
	ERAComputerBody.uc

6.1.5.1.1. USARBot Classes

ERA.uc defines the ERA robot as belonging to the class *SkidSteeredRobot* used by USARSim to simulate robots the rotation of which is produced by a change in the speed of rotation of the drive wheels. Skid Steered robots do not have steering wheels to change its orientation. Such is the case of P2DX and P2AT robots, both implemented in USARSim. ERA.uc class was obtained from the class P2DX.uc by modifying some lines to fit the physical parameters of ERA robot.

USAR_ERALTire.uc defines the left driving wheel of the ERA robot as an extension of the class BulldogTire.uc, which is used by USARSim to simulate robots as P2DX or P2AT. In this file are set the parameters that define the physical behavior of the wheel, which is associated with the mesh called *ERARobot.Wheels.ERALeftWheel*, that is the mesh *ERALeftWheel* contained in the group *Wheels* at the static mesh file *ERARobot.utx*.

USAR_ERARTire.uc is similar to the previous class, but in this case the wheel is associated with the static mesh *ERARobot.Wheels.ERAReftWheel*.

USAR_ERASmallTire.uc defines the ERA robot's rear idler (castwheel) as an extension of the class *BulldogTire.uc*, used by USARSim to simulate robots as P2DX. In this file we configured the physical parameters of that wheel and associated it with the static mesh called *ERARobot.Wheels.ERASmallWheel*.

Fig. 15. P2DX and ERA robot wheels.

ERARangeSensor.uc is equal to the USARSim class *RangeSensor.uc*, but in this case, we have changed the name to allow modifications that only affect the ERA robot's sensors. *RangeSensor.uc* is an extension of the class *Sensor* used as a basis for programming all types of USARSim robots sensors. *RangeSensor*, in turn, is used as a basis for sonar or laser sensors.

ERASonarSensor.uc defines the operation and scale of the ERA robot sonar.

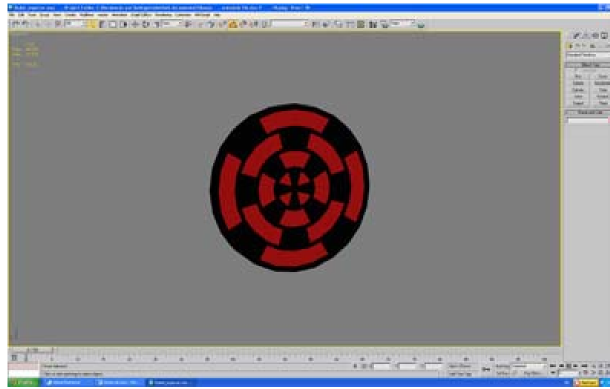

Fig. 16. 3D Studio Max screenshot showing an ERA robot's sonar.

6.1.5.1.2. USARModels Classes

ERAComputerBody.uc defines the body of the ERA robot's computer as an extension of the class KDPart, used by USARSim as the basis for all parts of the non-wheeled robots. In this case, we configured the physical parameters of the computer's body and associated it with the static mesh called *ERARobot.Computer.Computer*.

ERALasser.uc defines the ERA robot's computer as an extension of the class *RangeScanner* used by USARSim as the basis to simulate laser sensors. In this file the static mesh *ERARobot.Lasser.Lasser* is assigned to the ERA robot laser and is scaled to have the right size.

ERALTire.uc defines the robot's left drive wheel as an extension of *USAR_ERALTire* class and sets all its physical parameters of simulation.

ERARTire.uc defines the robot's right drive wheel as an extension of *USAR_ERARTire* class and sets all its physical parameters of simulation.

ERASmallTire.uc defines ERA robot's cast wheel as an extension of the class *USAR_ERASmallTire* and adjusts all its physical parameters of simulation.

6.1.5.1.3. USARMisPkg Clases

ERAComputerBody.uc defines the body of the ERA robot's computer as an extension of the class *MisPkgLinkInfo* used by USARSim to simulate a wide range of accessories (or parts of them) for robots. In this case, the class *ERAComputerBody* is assigned to the computer’s body, which is detailed by an USARModels class with the same name.

ERAComputer.uc defines the ERA robot's computer as an accessory. Later in the file *USARMisPkg.ini* it is specified that *ERAComputer* is an accessory consisting of a single element called *ERAComputerBody*.

Once all these classes have been defined and stored in their corresponding folders, we have compiled them by running the file *make.bat* in each of those folders. Thus, all the ERA robot parts are defined physically and cinematically. But, so that the robot can be simulated, it is necessary to tell USARSim how are these parts interconnected. With this objective, we proceeded to modify the files *USARMisPkg.ini* and *USARBot*.ini in the Unreal Tournament 2004 *System* folder.

6.1.5.2. USARMiskPkg

Experiments with robots often require the use of a variety of devices, mostly optional, which must be mounted on those robots. In other words, each test requires the robot to mount a series of accessories such as a camera, an arm, a clamp... according to the experiment purposes. All these optional devices are included within a USARSim group called *USARMisPkg*. As discussed earlier in this document, the folder *USARMisPkg* contains a directory called *classes* where are defined all those classes corresponding to robot accessories.

On the other hand, the *System* folder contains the file *USARMisPkg.ini*, which describes how many parts make up every accessory and how are they interconnected. In order that the ERA robot's computer can be removed from the robot when needed, we have added the following lines in the file *USARMisPkg.ini*:

```
;-----------------------------------------------------------------------------------------------------------
; Computer mission package used for the ERA
;-----------------------------------------------------------------------------------------------------------
[USARMisPkg.ERAComputer]
Links = (LinkNumber = 1, LinkClass = Class'USARMisPkg.ERAComputerBody', DrawScale3D = (X=1.0, Y=1.0, Z=1.0), ParentLinkNumber = -1, SelfMount = "A")

;-----------------------------------------------------------------------------------------------------------
; Computer Links used for the ERA
;-----------------------------------------------------------------------------------------------------------
[USARMisPkg.ERAComputerBody]
MountPoints = (Name = "A", JointType = "Revolute", Location = (X= 0.0, Y=0.0, Z=0.0), Orientation=(X = 1.5707963267948966192313216916398, Y = 0, Z = 0))
MountPoints=(Name="B",JointType= "Revolute", Location = (X= -0.0, Y = 0.0, Z= -0.0), Orientation=(X=1.5707963267948966192313216916398,Y=0,Z=0))
MaxSpeed=0.1745
MaxTorque=20
MinRange=0.0
MaxRange=3.14159
```

In this way, the ERA robot's computer (ERAComputer) is defined as an optional part of the robot, consisting of a single body (ERAComputerBody), which can rotate around its z axis.

6.1.5.3. USARBot

Once declared all parts of the ERA robot, next step is to add the ERA robot to the list of those available to be simulated with USARSim. This objective requires the file *USARBot.ini* (located in the System folder) to be modified by inserting a block of lines containing at least the following information:

1. *Robot’s weight.*
2. *Robot’s maximum load.*
3. Parts of the robot, including accessories.
4. Location of the binding sites between adjacent parts.
5. Type of articulation that restricts the relative movement between adjacent elements

With this objective we have introduced the following lines in the *USARBot.ini* file:

```
[USARBot.ERA]
bDebug=false
Weight=9
Payload=20
ChassisMass=2.000000
bMountByUU=False
JointParts=(PartName="RightFWheel",PartClass=class'USARModels.ERARTire',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),bSteeringLocked=True,bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',ParentPos=(Y=0.1606467,X=0.077586700,Z=0.09909333),ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),SelfPos=(Z=-0.0),SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0))
JointParts=(PartName="LeftFWheel",PartClass=class'USARModels.ERALTire',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),bSteeringLocked=True,bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',ParentPos=(Y=-0.1606467, X=0.077586700, Z=0.09909333), ParentAxis=(Z=1.0), ParentAxis2= (Y=1.0), SelfPos = (Z=-0.0), SelfAxis=(Z=1.0), SelfAxis2=(Y=1.0))
JointParts = (PartName = "RearWheel", PartClass = class'USARModels.ERASmallTire', DrawScale3D = (X = 1.0,  Y = 1.0, Z = 1.0), bSteeringLocked = False, bSuspensionLocked = true, Parent="",JointClass = class'KCarWheelJoint', ParentPo s = (Y = 0.0, X = -0.19948333, Z = 0.1438400), ParentAxis = (Z = 1.0), ParentAxis2 = (Y = 1.0), SelfPos = (Z=-0.0), SelfAxis = (Z = 1.0), SelfAxis2=(Y = 1.0))
MisPkgs=(PkgName="CameraPanTilt",Location=(Y=0.10,X=0.08,Z=-0.01),PkgClass=Class'USARMisPkg.CameraPanTilt')
MisPkgs=(PkgName="ERAComputer",Location=(Y=0.00,X=-0.09,Z=-0.01),PkgClass=Class'USARMisPkg.ERAComputer')
Cameras=(ItemClass=class'USARBot.RobotCamera',ItemName="Camera",Parent="CameraPanTilt_Link2",Position=(Y=0.08,X=0.06,Z=-0.0188),Direction=(Y=0.0,Z=0.0,X=0.0))
```

HeadLight=(ItemClass=class'USARBot.USARHeadLight',ItemName="HeadLight",Parent="Cam
eraTilt_Link2",Position=(Y=0.08,X=0.079999916,Z=0.07399993),Direction=(Y=-
0.2876214,Z=0.0,X=0.0))
Sensors=(ItemClass=class'USARBot.ERASonarSensor',ItemName="F1",Position=(X=0.147216
67,Y=-0.11129667,Z=0.03873333),Direction=(Y=0.0,Z=-1.5707964,X=0.0))
Sensors=(ItemClass=class'USARBot.ERASonarSensor',ItemName="F2",Position=(X=0.182916
67,Y=-0.09289333,Z=0.03873333),Direction=(Y=0.0,Z=0.0,X=0.0))
Sensors=(ItemClass=class'USARBot.ERASonarSensor',ItemName="F3",Position=(X=0.174180
00,Y=-0.06439000,Z=0.03873333),Direction=(Y=0.0,Z=0.747001,X=0.0))
Sensors=(ItemClass=class'USARBot.ERASonarSensor',ItemName="F4",Position=(X=0.162360
00,Y=-0.04142000,Z=0.03873333),Direction=(Y=0.0,Z=0.204029,X=0.0))
Sensors=(ItemClass=class'USARBot.ERASonarSensor',ItemName="F5",Position=(X=0.162360
00,Y=0.04142000,Z=0.03873333),Direction=(Y=0.0,Z=-0.204029,X=0.0))
Sensors=(ItemClass=class'USARBot.ERASonarSensor',ItemName="F6",Position=(X=0.174180
00,Y=0.06439000,Z=0.03873333),Direction=(Y=0.0,Z=-0.747001,X=0.0))
Sensors=(ItemClass=class'USARBot.ERASonarSensor',ItemName="F7",Position=(X=0.182916
67,Y=0.09289333,Z=0.03873333),Direction=(Y=0.0,Z=0.0,X=0.0))
Sensors=(ItemClass=class'USARBot.ERASonarSensor',ItemName="F8",Position=(X=0.147216
67,Y=0.11129667,Z=0.03873333),Direction=(Y=0.0,Z=1.5707964,X=0.0))
Items=(ItemClass=class'USARBot.Item',ItemName="cosa",Position=(X=0.114999875,Y=0.1299
9986,Z=-0.01),Direction=(Y=0.0,Z=1.5707964,X=0.0))
Sensors=(ItemClass=class'USARModels.ERALasser',ItemName="Scanner1",Position=(X=0.13
83900,Y=0.0,Z=-0.0039600),Direction=(Y=0.0,Z=0.0,X=0.0))
Sensors=(ItemClass=class'USARBot.INSSensor',ItemName="INS",Position=(X=0.0,Y=0.0,Z=-
0.0),Direction=(Y=0.0,Z=0.0,X=0.0))
Sensors=(ItemClass=class'USARBot.OdometrySensor',ItemName="Odometry",Position=(X=0.0
,Y=0.0,Z=-0.0),Direction=(Y=0.0,Z=0.0,X=0.0))
Sensors=(ItemClass=class'USARBot.EncoderSensor',ItemName="ECLeft",Parent="LeftFWheel
"Position=(X=0.0,Y=0.0,Z=-0.0),Direction=(Y=0.0,Z=1.5707964,X=0.0))
Sensors=(ItemClass=class'USARBot.EncoderSensor',ItemName="ECRight",Parent="RightFWh
eel",Position=(X=0.0,Y=0.0,Z=-0.0),Direction=(Y=0.0,Z=1.5707964,X=0.0))
Sensors=(ItemClass=class'USARBot.EncoderSensor',ItemName="ECTilt",Parent="CameraPan
Tilt_Link2",Position=(X=0.0,Y=0.0,Z=-0.0),Direction=(Y=0.0,Z=1.5707964,X=0.0))
Sensors=(ItemClass=class'USARBot.EncoderSensor',ItemName="ECPan",Parent="CameraPa
nTilt_Link1",Position=(X=0.0,Y=0.0,Z=-0.0),Direction=(Y=1.5707964,Z=0.0,X=0.0))
Sensors=(ItemClass=class'USARBot.RFIDSensor',ItemName="RFID",Position=(X=0.0,Y=0.0,Z
=-0.0),Direction=(Y=0.0,Z=0.0,X=0.0))
Sensors=(ItemClass=class'USARBot.VictSensor',ItemName="VictSensor",Parent="CameraPan
Tilt_Link2",Position=(Y=0.0,X=0.06,Z=-0.0088),Direction=(Y=0,Z=0,X=0))

Sensors=(ItemClass=class'USARBot.GroundTruth',ItemName="GroundTruth",Position=(X=0.0, Y=0.0,Z=-0.0),Direction=(Y=0.0,Z=0.0,X=0.0))
Effecters=(ItemClass=class'USARBot.RFIDReleaser',ItemName="Gun",Parent="",Position=(Y= 0.0,X=-0.1523808,Z=0.142857),Direction=(Y=0.0,Z=3.1415927,X=0.0))

At this point the ERA robot is ready for virtual experiments using USARSim. Now we have to create a virtual world.

6.2. Creating a virtual world

The virtual world needed for virtual testing should include the research labs of the Engineering and Computer Science Department at the University Jaume I and some of their immediate environment. This is consistent with much of the ground floor and first floor of the TI building at the university, so it was decided to model the whole building and its immediate surroundings. Figures 17 and 18 show exterior views of the virtual TI building.

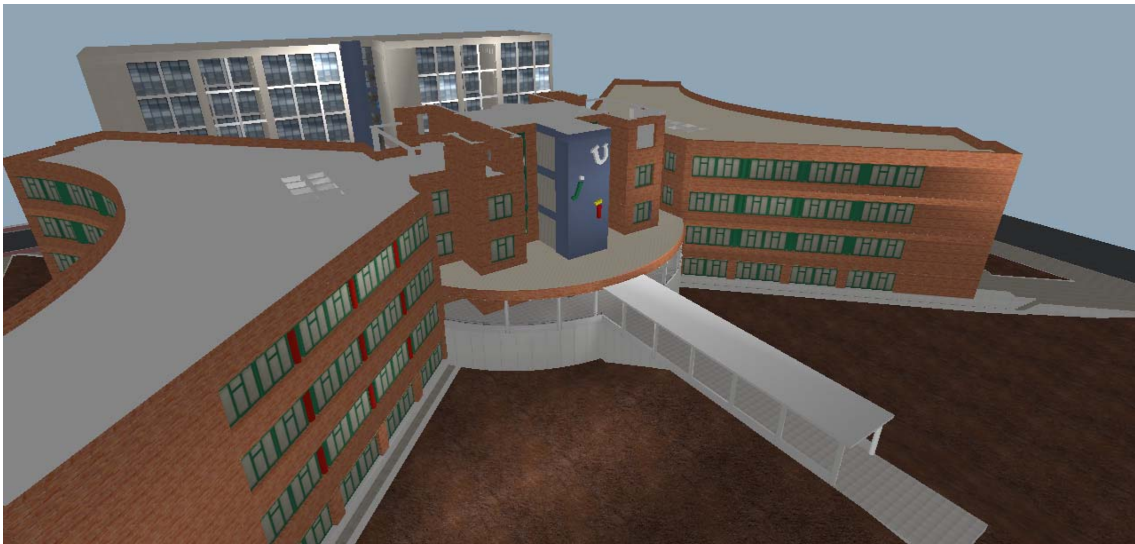

Fig. 17. TI building exterior view.

Fig. 18. TI building exterior view.

In order to create the virtual TI building we have followed a similar procedure to that used to obtain the virtual ERA robot, but there are, by contrast, some differences that it is important to note.

6.2.1. Drawings and measures

To model the TI building we have used its drawings in AutoCAD .dwg format (see figure 19), what have done things easier. However, although those files contain much useful information, it has been necessary to do many measurements, especially with regard to height of windows and some items not appearing on the drawings. Most of these elements are part of the furniture inside the building such as: bulletin boards, bins, vending machines, benches, information boards, maps of evacuation...; others, however, could be included as part of the building installations: switches of many types (light, elevator, fire alarm), sensors (smoke detectors or presence detectors) lamps, fire hydrants, fire extinguishers...

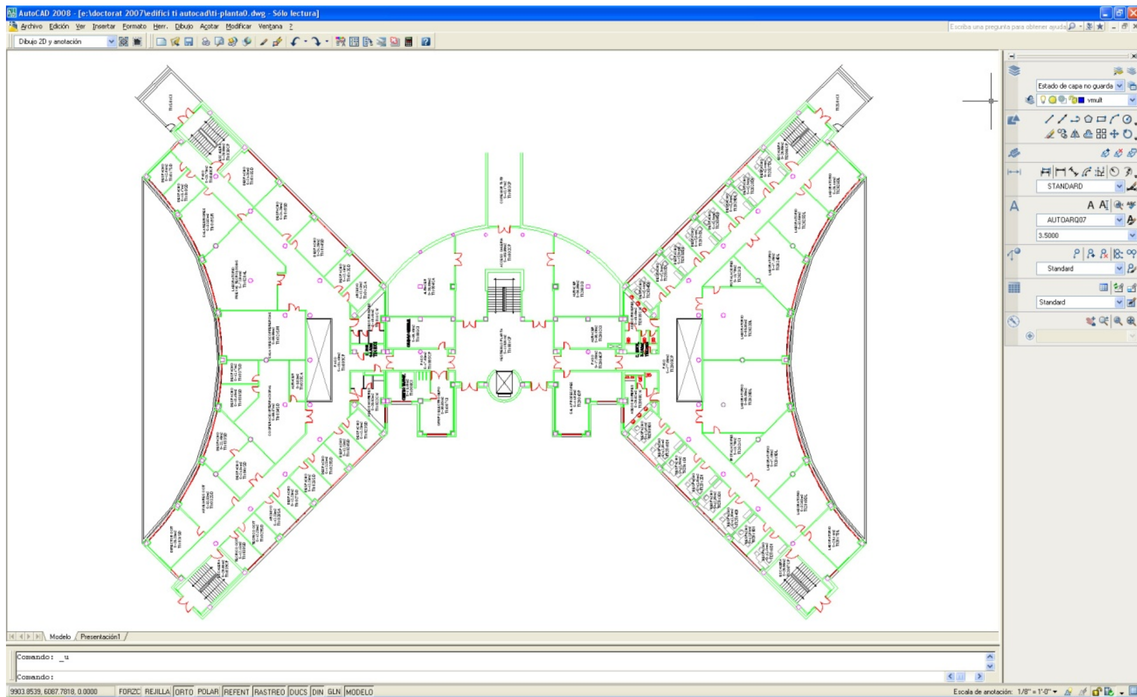

Fig. 19. TI building first floor AutoCAD drawing

6.2.2. Pictures

The wide diversity of materials and colors that make up the elements that conform the TI building and its surroundings has forced to make many photographs. Thus, we have taken pictures of the walls, ceilings, floors, doors, sidewalks, cement, asphalt, signs, lamps, railings, columns, vents, sensors, etc… Those pictures have been treated with Adobe Photoshop CS3 to get the desired textures, similarly to the process for the ERA robot.

6.2.3. 3D environment

With the dimensions and textures of the building and its immediate surroundings, begin the development phase of the virtual environment model using 3D Studio Max.

6.2.3.1 LowPoli aproximation

The construction of the virtual TI building has taken many hours due to the large number of elements that conform it such as walls, ceilings floors, columns, windows, doors...

Walls

On the one hand, there are the walls, slab, floor and ceiling of each floor. These elements have been created from AutoCAD drawings of the building. Thus, the drawings of each floor have been imported from 3D Studio Max, then they have been superimposed on the distance between floors and each wall has been created so that its shape is as simple as possible. That is, we have tried to create as many walls as possible from extruded rectangles, generating parallelepipeds. This approach brings many important advantages and some minor inconveniences that is interesting to note.

It is easier to apply texture maps and collision envelopes when dealing with hexahedral walls (boxes). Texture maps use to accommodate to certain simple geometric shapes such as hexahedrons, cylinders, spheres or planes. When importing 3D models from UnrealEd, those which offer better simulation performance are those whose collision hulls are based on combinations of geometric figures as above. Moreover, if working with 3D Studio Max we export a single hexahedron as a wall, there is an additional advantage because when we import this wall from UnrealEd we will be able to associate a box as collision hull. Although this latter option does not always work as desired depending on the orientation of the wall (see figure 21).

In contrast, dividing all walls into hexahedral shapes involves multiplying the number of objects to export, which is already very high. You just have to take a

look at the drawing of a floor to realize that the number of walls is higher than 200 (figure 21 shows that in the first floor there are more than 122 walls).

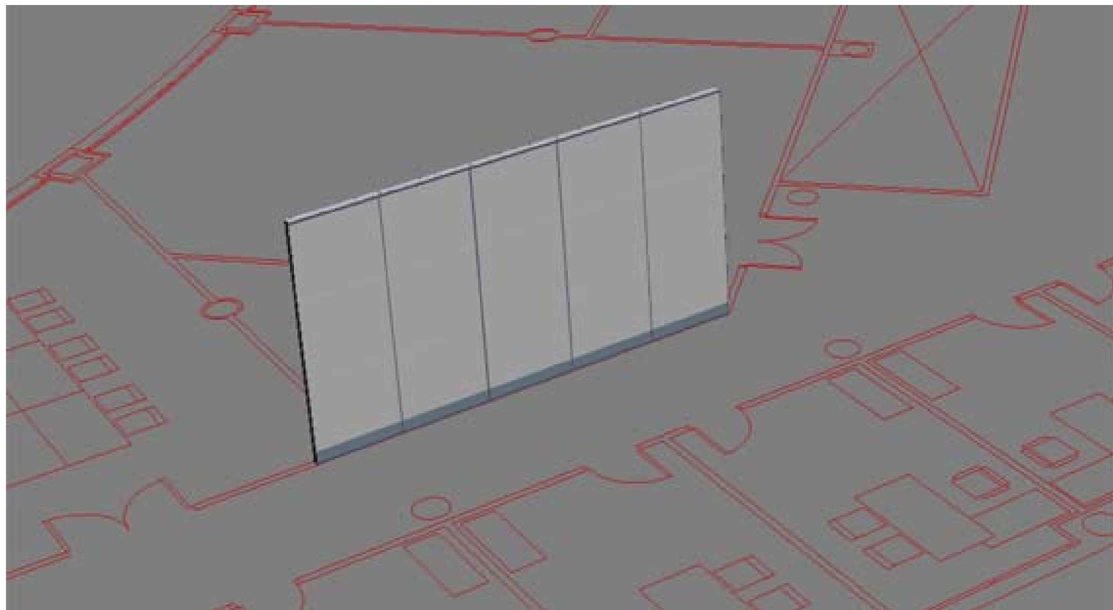

Fig. 20. TI building wall being created with 3D Studio Max

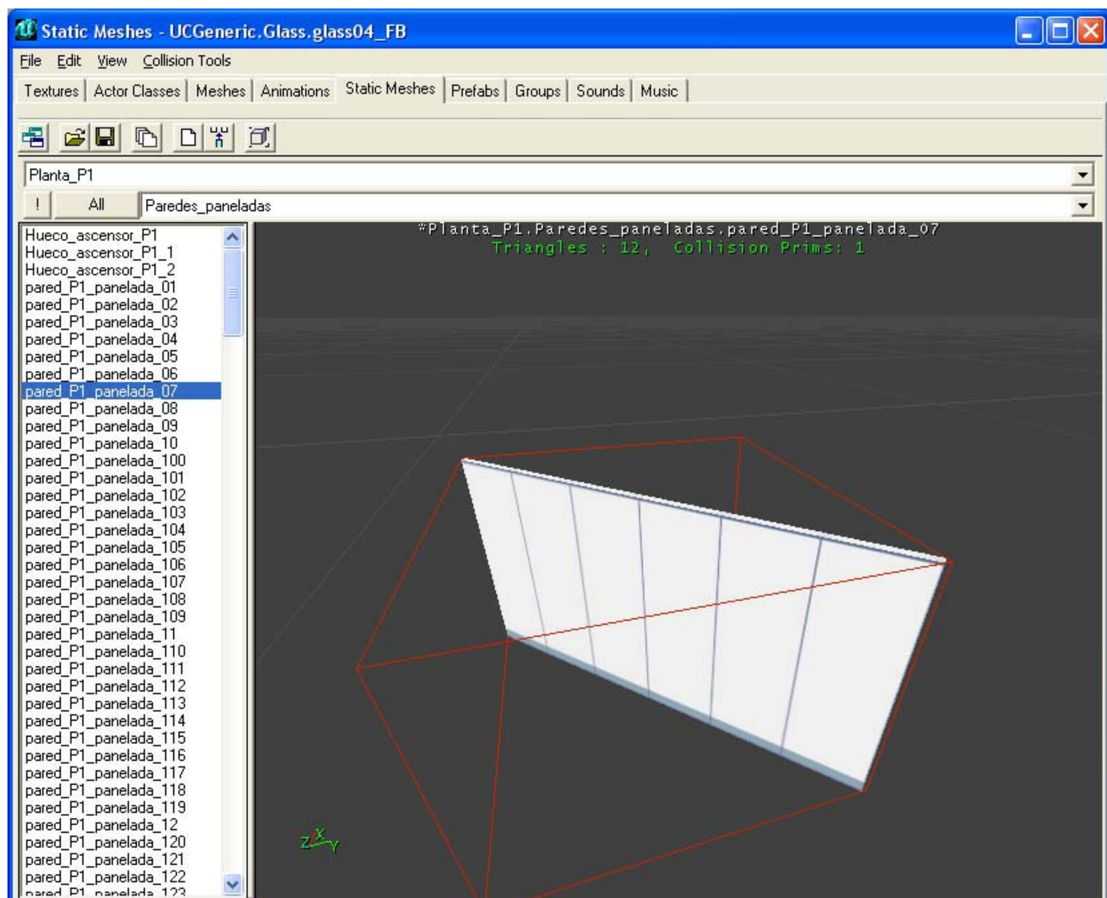

Fig. 21. Static Mesh Browser showing a wall with an UnrealEd box collision hull

Floor slabs and ceilings

From the floor drawings of each building we have created both floor slabs and ceilings. In general, this task has been easier than wall creation, as the number of elements is reduced.

Basically, to obtain the 3D shape of these elements, we have redrawn the outer edge of each floor and those lines defining the gaps between different floors. The resulting line has been extruded forming a floor slab (see figure 22) in some cases and, in others, a ceramic coating or a ceiling.

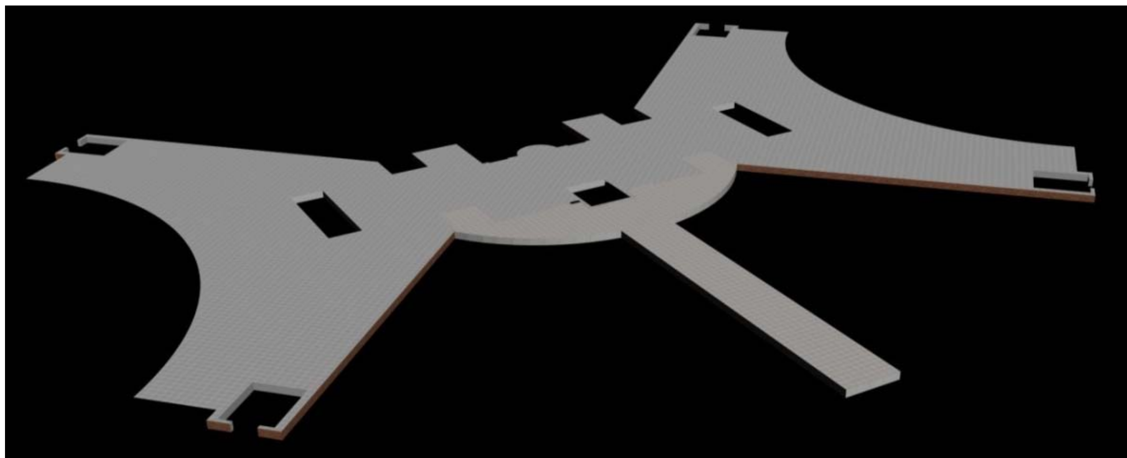

Fig. 22. Floor slab in 3D Studio Max

Concerning to ceilings, they are modeled from photographs, respecting their paneled structure, fluorescent lamps and air conditioning diffusers (see figure 23).

Fig. 23. Paneled office ceiling with fluorescent lamps and air conditioning diffuser

Columns

The cylindrical columns that are part of the building structure have been modeled according to the drawings of each floor. In this case, and for obvious reasons, they have come into cylinders with a sufficient number of faces to look appropriate.

Windows

In the TI building there are various types of windows that have been shaped differently to facilitate their export to UnrealEd. In general, the windows are located in places where no direct clashes will occur with the robot or at least, if these contacts are realized, there will be simultaneous collisions with the wall that holds the window. This represents a considerable advantage in dealing with such objects as it does not require any collision envelope for windows.

With the outside windows of laboratories, workshops and offices we have followed a different procedure to that used to obtain the internal windows. This is due to the presence of blinds, which are modeled as a plane with a texture that mimics these mechanisms (see figure 24). This procedure requires export to ASCII format one by one all those windows, because of difficulties with UVW maps that determine the placement of textures on objects. Explanation is simple: all objects whose texture is a more or less uniform color (like green textured aluminum frames of a window) can be exported as a whole, so that when imported from UnrealEd they continue to retain its texture apparently intact, whereas this does not happen the same way when the texture is not uniform, despite being the same for all exported objects.

Fig. 24. Outside window with blinds

As described in previous paragraph, those windows equipped with only frame, hinges, handle and glass are easier to export to ASCII format because, in many cases, we can include several of these windows in one only mesh, what reduces the number of objects and work. Figure 25 shows an exterior window with the above characteristics.

Fig. 25. Window with only frame, hinges, handle and glass

As regards the export of windows it is important to explain the benefits of the particular way in which a robot may collide or not with them. However, it is important to note that the objects exported with no collision hull from 3D Studio Max to UnrealEd have certain peculiarities during a simulation: on the one hand, they have no capability to hit the robots, so they are traversed by robots with no interaction; on the other hand, they are detected by the robot's sensors and are able to collide with any other mobile entity. This permits saving collision envelopes for windows and other objects that are not reachable by the robots and, in many cases, to export these objects in groups.

Doors

Despite its apparent simplicity, the doors are difficult to treat elements in a virtual world. That is, their shape is usually simple (like a hexahedron) as well as their textures, but their position and movement possibilities result in the need to take certain decisions prior to their placement in a virtual world.

At the TI building we can distinguish three types of doors: those static, those that can be pushed by the robot and those that open automatically when the robot is approaching. The former are treated as if they were walls. The second and third, however, must be considered as moving objects with the difficulties that this entails.

The doors of the laboratories and seminars of the TI building have a glass in the middle of each door sheet. If we consider one of these doors as fixed (static), then we can use the following simplification: modeling the door without its glass, handle, lock and hinges, as they are separate items that can be exported into homogeneous groups to UnrealEd. Thus, each one of these independent elements has a texture and a simple collision envelope and, moreover, we can create the glass from UnrealEd as an independent and stationary object. In contrast, if a door with glass is required to be mobile, then we will have to model the whole set as a unique mesh with all their textures properly distributed and their collision hull; also UnrealEd can import only objects with image-based textures in tga format (this is an important handicap that will be explained later). Figure 26 shows a door with glass.

Fig. 26. Hall door with glass porthole

So, there are still some questions that need to be explained: What happens when we have glass or opaque materials? Can we export them from 3D Studio Max or other 3D design application?

To export successfully doors with parts of glass we have followed this procedure: first, we have taken the name of a glass texture of those used by UnrealEd and then from 3D Studio Max, we have assigned that name to any

texture not used; second, we have applied that texture to the transparent parts of the door in construction and, when ready, we have exported that door in ASCII format; third, with the UnrealEd texture browser opened we have selected the texture whose name we have been using and then we have imported the door using the UnrealEd static mesh browser; what have forced UnrealEd to whether searching the door texture by its name or, if not found, applying the default texture (the selected texture in the texture browser) to the transparent surface. In this way, we can get moving objects with transparent or semi-opaque parts. Figure 27 shows some of the UnrealEd texture possibilities to simulate glass.

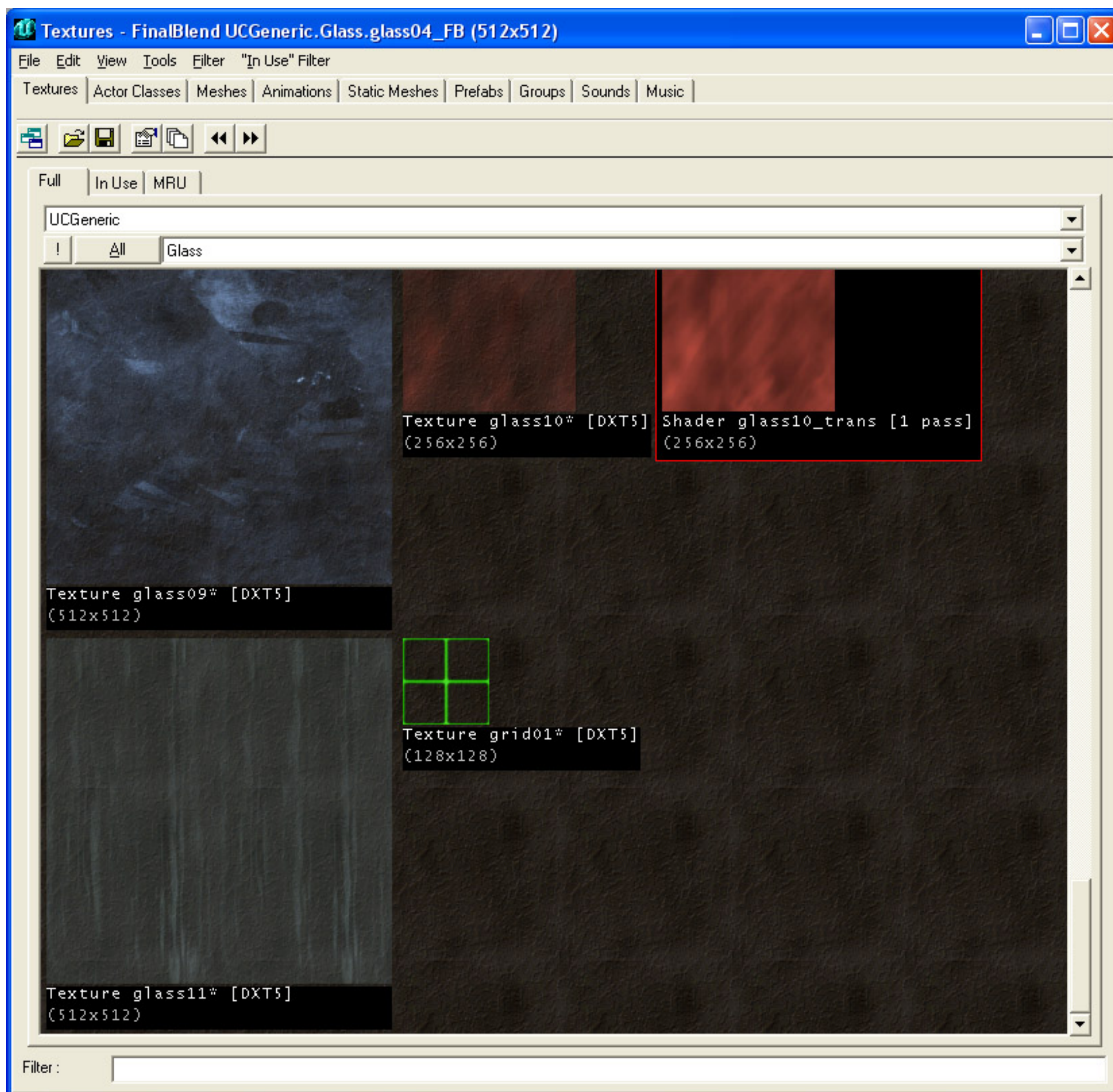

Fig. 27. UnrealEd texture browser showing glass textures

Once imported to UnrealEd, the swinging doors should be treated very differently to those static. Since there are two types of swinging doors and they are very different, we will describe them separately.

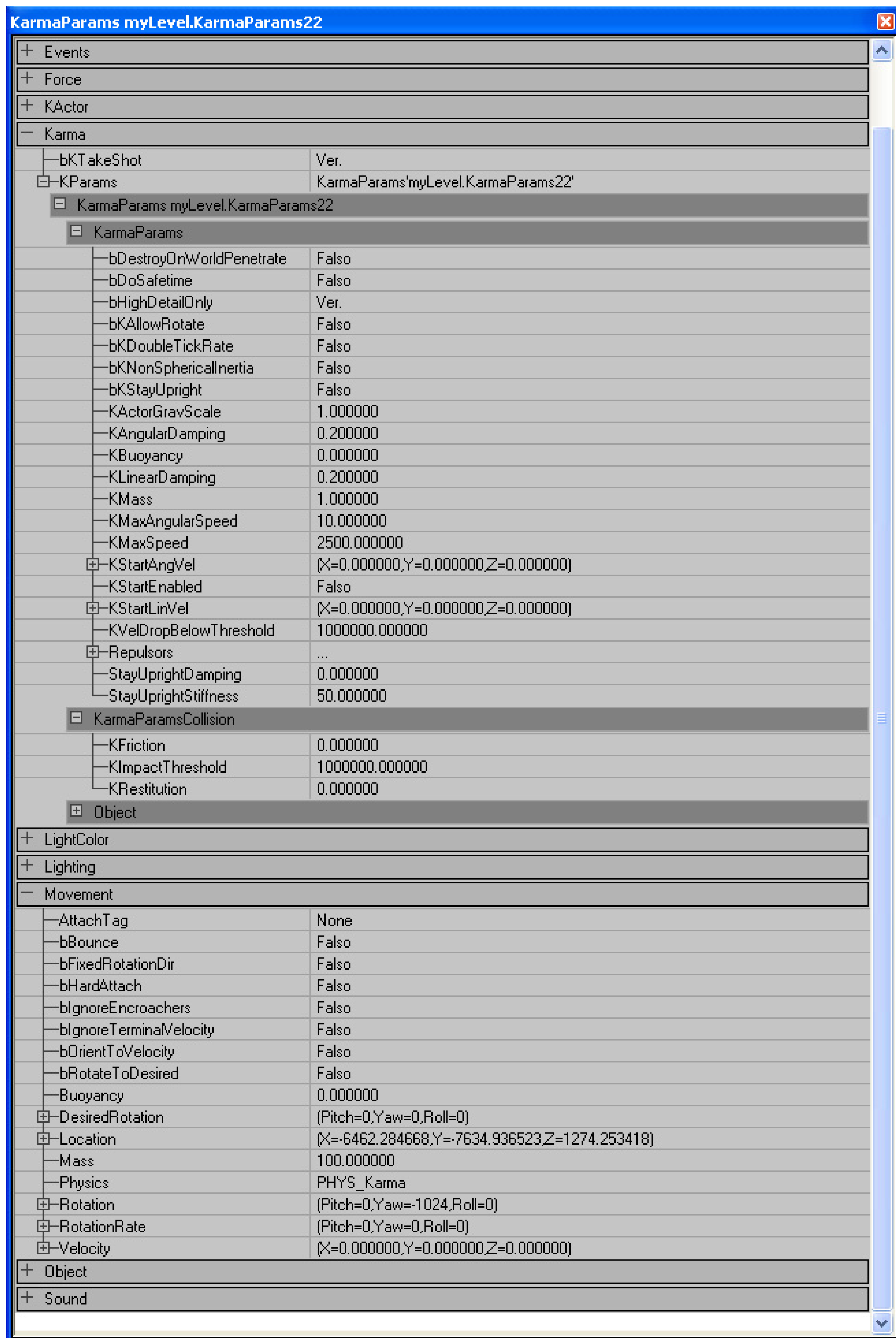

Fig. 28. Pushable door properties (KActor properties)

Pushable doors

Doors that only move when they are pushed by robots must be introduced to the map as an *Actor* rather than *StaticMesh* (such as walls, and other non-moving objects). This done, the next step is to adjust the parameters describing the kinematics of each door of this type present on the virtual level. We can do that by clicking the right mouse button over the door in process and then clicking the *Kactor Properties* shown in the figure above (figure 28). The most important variables are:

KarmaParams

bHighDetailOnly: Some objects introduced as *Karma Actor* (*KActor*) will only use Karma Physics if this parameter is set to True. In addition, to avoid slow machines change it to False, trying to increase their performance in the simulation, you must change the following lines in the file default.ini

[Engine.LevelInfo]

PhysicsDetailLevel=PDL_Medium

Instead of

Engine.LevelInfo]

PhysicsDetailLevel=PDL_High

KActorGravScale: Parameter that describes how much an object is affected by gravity during a motion.

KAngularDamping: Parameter that describes how much air friction attenuates the rotational movement of an object.

KBuoyancy: Applies in water volumes. This parameter set to 1.2 will allow the object to float to the surface when within a water volume. 0 = no buoyancy.

KLinearDamping: Linear velocity damping (drag).

KMass: Mass used for Karma physics.

KarmaParamsCollision

KFriction: Parameter that defines the friction coefficient between the actor and the ground or other surfaces.

KRestitution: Factor describes how much an object is able to absorb a collision. The value of 0.0 makes the object behave like a block of stone

Additionally, it is important to know that if an object with no defined collision envelope must hit against others, then it is necessary to set to True the parameter *UseSimpleKarmaCollision* from the StaticMesh Browser, otherwise, UT2004 won't find any envelope and collisions will not occur. When an envelope is defined in this way, it is possible to adjust to simple geometric shapes like a box or a line by setting the parameters *UseSimplelinCollision UseSimpleBoxCollision* to true or false.

Next, it is necessary to define the restrictions on the movement of the doors (constraints). In the TI building there are no sliding doors, all of them rotate about an axis. Therefore, for each one of these doors we have defined a rotation axis as shown in the figure below.

Fig. 29. Laboratory door rotation axis in orange

When working with UnrealEd the axes of rotation as described are called *KHinge* and we have added them to our map by selecting from the Actor Classes Browser the *KHinge* option in the *Kconstraint* group at the *KActor* directory (see figure 30). Later, we have clicked the right mouse button on the desired entry point, and then we have selected the choice "Add KHinge here". Finally we have taken the KHinge from the its insertion point and we have located it into its right place (door rotation axis).

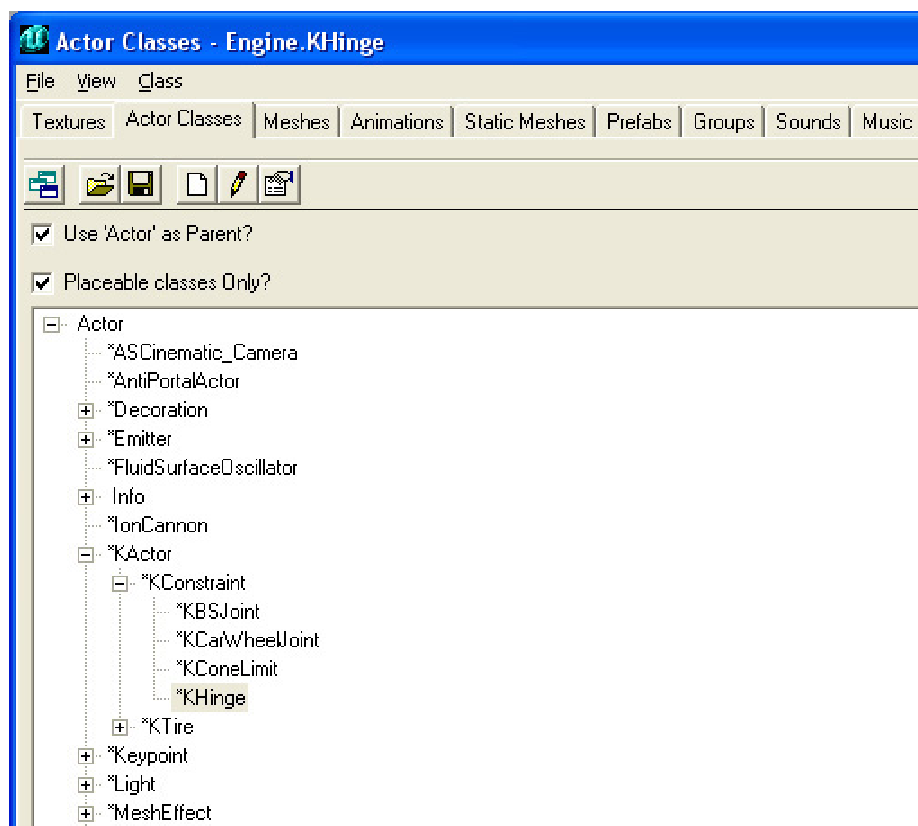

Fig. 30. Actor Classes Browser showing KConstraint tree

With the axis in place, we have proceeded to assign the sheet of a door as an object whose motion likely will be constrained by that. So, we have accessed to the axis properties (*Khinge* Properties) and opened the group *KarmaConstraint*. The parameter *KConstraintActor1* must poi nt to the door sheet in process, with this objective we have clicked on this parameter and chosen the aforementioned sheet as shown in Figure 31.

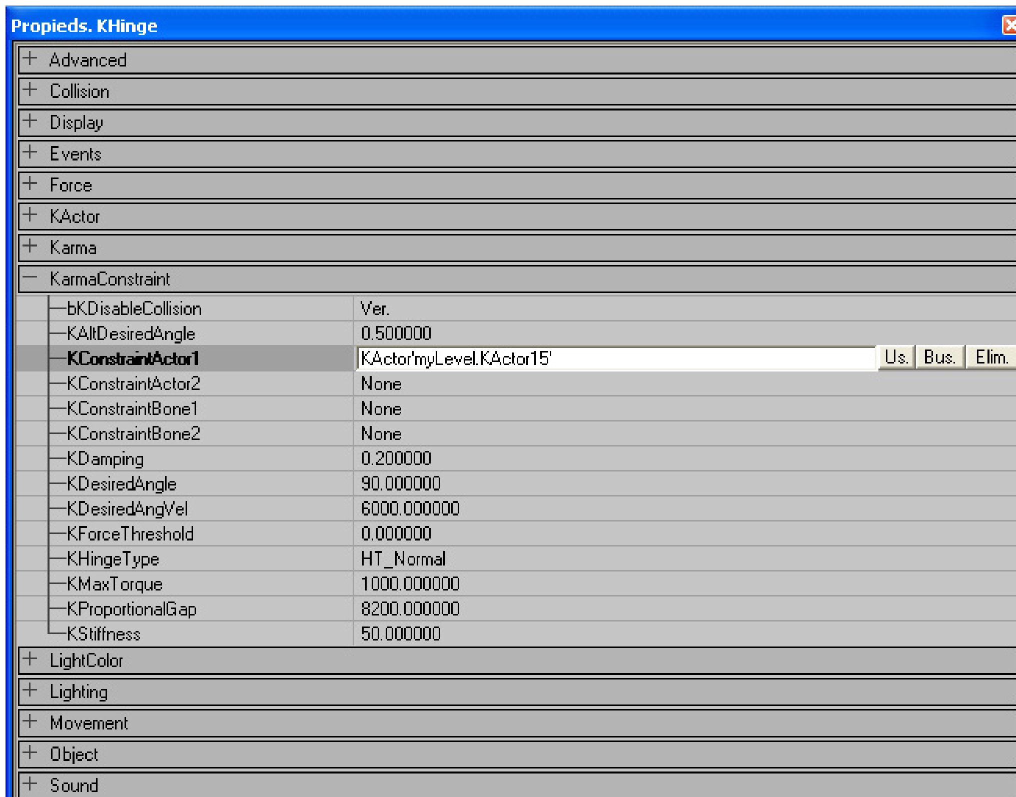

Fig. 31. KHinge parameters

Automatic Doors

To get doors that open automatically when approaching a robot we have followed an entirely different procedure than above. First, we have introduced a *Mover* on the TI building map and then we have assigned it to the mesh of a moving door sheet. This last step can be done from the *Mover's* properties selecting the option *StaticMesh* from *Display* group as shown in figure 32.

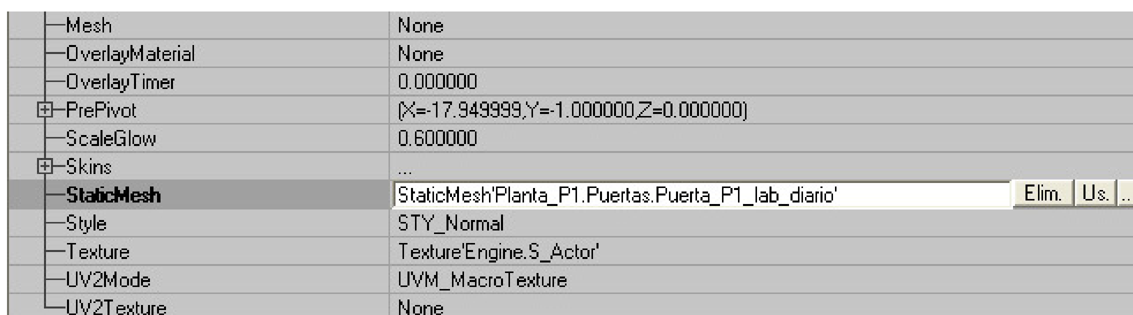

Fig. 32. Assigning a mesh to a *Mover*

It is then necessary to define what kind of movement makes the *Mover*, linear (sliding door) or circular (swinging door), from the parameter *KeyNum* in the group *Mover* of *Mover* properties (see figure 33). Thus, starting from the door in closed position (KeyNum 0), we have selected *KeyNum* 1 and placed the sheet of the door in fully open position, so that with only two positions (NumKeys 2) its trajectory is determined.

Mover	
AntiPortalTag	None
bDamageTriggered	Falso
bDynamicLightMover	Falso
bOscillatingLoop	Falso
BrushRaytraceKey	0
bSlave	Falso
bToggleDirection	Ver.
bTriggerOnceOnly	Falso
BumpEvent	None
BumpType	BT_PlayerBump
bUseShortestRotation	Falso
bUseTriggered	Falso
DamageThreshold	0.000000
DelayTime	0.000000
EncroachDamage	0
KeyNum	1
MoverEncroachType	ME_IgnoreWhenEncroach
MoverGlideType	MV_GlideByTime
MoveTime	1.000000
NumKeys	2
OtherTime	0.000000
PlayerBumpEvent	None
StayOpenTime	4.000000
WorldRaytraceKey	0

Figure 33. Mover group of parameters

To determine the behavior of automatic doors is also necessary to adjust various parameters of the *Mover* properties, the most important of which are described below sorted by categories:

Mover

bTriggerOnceOnly: Boolean variable indicating in this case if the door will open once and remain open.

EncroachDamage: Variable that determines the amount of damage caused by the *Mover* by colliding with an agent who has crossed his path.

MoverEncroachTipe: Describes the behavior of the Mover when an agent (eg a robot) crosses their path. Normally *ME_IgnoreWhenEncroach* option is chosen for this parameter and the value 0 for *EncroachDamage*. So agents can pass through the doors that automatically open to them.

MoveTime: Time required by a *Mover* to complete a trajectory. In our case the time in seconds it takes for the door to switch from its open to closed position or vice versa.

StayOpenTime: Time the door remains opened when the triggering agent is gone.

Movement (see figure 34)

bHardAttach: Boolean variable that determines whether the *Mover* moves attached to other objects.

Mass: Mover Mass.

Events (see figure 35)

Tag: Variable used to activate the *Mover* motion by an event such as pressing a remote pushbutton. This parameter must contain a character string that identifies a *Mover* unequivocally.

MoverSounds (see figure 36)

ClosedSound: Mover Sound when reaching closed position.
ClosingSound: Mover Sound when closing.
OpenedSound: Mover Sound when reaching opened position.
OpeningSound: Mover Sound when opening.

Object (see figure 36)

InitialState: Parameter that defines the initial state of a *Mover*, and therefore its activation event. We have chosen the option *BumpOpenTimed* for the doors activated by the presence of a character and *TriggerOpenTimed* for those doors operated by a remote button.

Movement	
AttachTag	None
bBounce	False
bFixedRotationDir	False
bHardAttach	False
bIgnoreEncroachers	False
bIgnoreTerminalVelocity	False
bOrientToVelocity	False
bRotateToDesired	False
Buoyancy	0.000000
DesiredRotation	(Pitch=0,Yaw=0,Roll=0)
Location	(X=-8487.000000,Y=-10192.000000,Z=1275.000000)
Mass	100.000000
Physics	PHYS_MovingBrush
Rotation	(Pitch=0,Yaw=-21504,Roll=0)
RotationRate	(Pitch=0,Yaw=0,Roll=0)
Velocity	(X=0.000000,Y=0.000000,Z=0.000000)

Fig. 34. *Mover's Movement* parameters group

Events	
Event	None
ExcludeTag	...
Tag	Puerta_diario

Fig. 35. *Mover's* Events parameters group

MoverSounds	
ClosedSound	None
ClosingSound	None
LoopSound	None
MoveAmbientSound	None
OpenedSound	None
OpeningSound	None
Object	
Group	None
InitialState	TriggerOpenTimed
Name	Mover1

Fig. 36. *Mover's* MoverSounds and Object parameters groups

Additionally, some doors have been programmed so that robots can open them by pressing a pushbutton. Those buttons have been added to the map from the UnrealEd Actor Browser by opening the parameters group *Triggers* and then selecting the option *Trigger*. Then, we have accessed to their properties and within the *Events* group we have given the parameter *Event* the name on the *Tag* parameter of the door we want to automate. Figure 37 shows an example of a remote controlled door.

Fig. 37. Laboratory door controlled by a push-button on the bottom left corner of the image

Stairs

The stairs were obtained using a method similar to that followed when creating walls, but in this case, we only took into account the collision hull for the first steps leading to an upper floor. This simplification is justified by the fact that the robots available in the Department of Engineering and Computer Science at the University Jaume I can not go up or down stairs. If these conditions changed with the need to simulate a robot capable of moving up or down the stairs, it would be easy to correct by adding a BlockingVolume for every step from UnrealEd or by adding a suitable collision hull from 3D Studio Max and re-

exporting the new stairs. The latter option has the advantage that the new steps being re-imported from UnrealEd would automatically appear in the correct place and with all properties identical to those defined for old stairs, except that in this case the new stairs would incorporate different collision envelopes.

It is important to note the functionality of those BlockingVolume for the current work, as they have been used to provide collision envelopes to various objects and parts of the environment with complex or large contours, such as soil or building slabs. This has prevented in some cases the robot fall indefinitely due to lack of collision with the ground.

Outer environment

It was decided to include the IT building close environment in order to give a likeable look to simulations, so that the building does not seem to be floating in outer space. For this purpose, we have created an approximate model of the landscape around the building making sure that what is observed through the windows and glass doors match reality.

Thus, we have constructed a realistic environment composed basically of land, streets, the library building of the University Jaume I and a blue sky. Because all of them are immobile elements, the procedure to create them was similar to that followed for walls, ceilings and floors, but in this case have not defined collision envelopes to simplify work and improve simulation speed. So, instead of collision envelopes we have added some *BlockingVolumes*.

Miscellaneous objects

Inside the TI building there is a large diversity of elements that have been necessary to include in the virtual environment to increase the realism degree of simulations. This is the case of railings, visible parts of the fire facility, posters of various types, vending machines, lamps, etc ... The static nature of these elements suggests giving them the same treatment as the walls. For obvious

reasons, we have omitted collision hulls for those objects not reachable by robots. Figure 38 shows some miscellaneous items such as posters and vending machines.

Fig. 38. Vending machines on the ground floor hall

6.2.4. Lights

The appearance of a virtual environment depends largely on the type of lighting that was used in its creation. Thus, the absence of light is unacceptable, because all surfaces would become black regardless of their original color.

To understand how it has solved the problem of lighting in this virtual environment it is necessary to know that the light reflected by the surface of an object comes from two sources: the object's own light auto lighting (if any) and the external light reaching the object.

When a new object is introduced in an UnrealEd virtual environment, in the absence of light sources, we can see that all its surfaces become black without any other distinguishing color or texture. To fix this, at first, we introduced external lights to illuminate all surfaces within their reach trying to get a natural look. The result, however, was not as expected. Surfaces near light sources

were almost white, that is burned, while distant objects to those lights were barely distinguishable in the dark.

The next step was to multiply the number of external lights, dividing their power with the hope that this would cause overexposed and underexposed areas to disappear. The result of this new lighting attempt was not significantly different from the previous, because, although to a lesser extent, there were still many over illuminated and infra illuminated surfaces. As a conclusion, it was drawn that only if the number of external lights tended to infinity it would disappear that problem, while a new one would emerge. The new problem was the growing shadow attenuation as light sources increased and, therefore, the number of angles from which they reach an object. If the shadows disappear, so does the realistic look of a virtual environment.

Parallel to the testing with external light sources we have done many experiments with self-illuminated objects from which we concluded that this type of lighting affects only on self-illuminated object colors. So we could see them as darker or lighter homogeneous colors, but without gloss or shadows on any of their faces and, therefore, without three-dimensional appearance. This is because for obtaining surfaces that degrade from light to dark colors or shadows cast by opaque objects, we must define at least a point from which lights up our virtual world.

Fig. 39. UnrealEd point of light

From all the above we can conclude that each object must have its own light degree in order to ensure a minimum illumination on all its sides and to minimize the amount of required external lights to give it a natural look . Figure 39 shows an UnrealEd light (external light) from which we can obtain shadows on nearby surfaces.

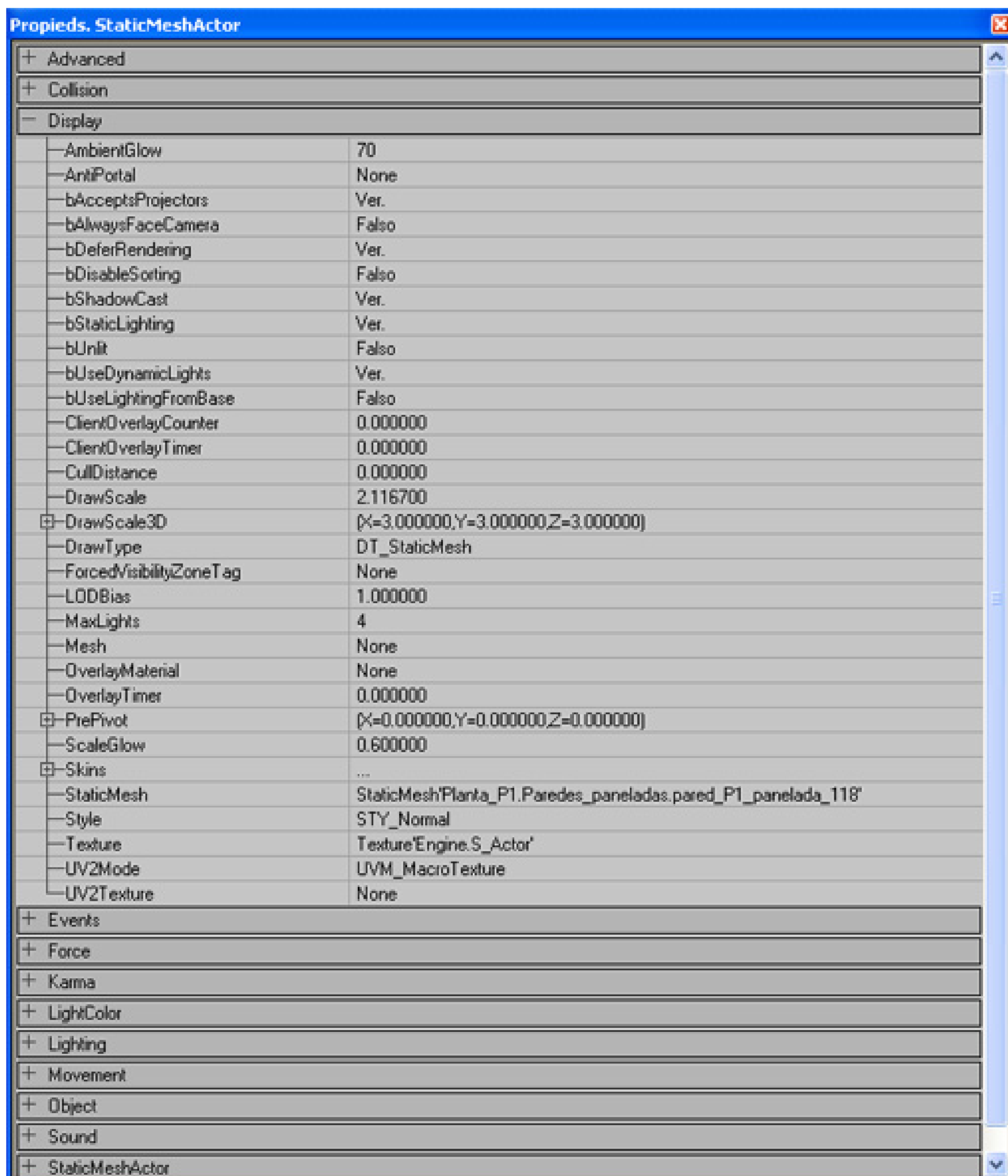

Fig. 40. Graphic properties of an UnrealEd object

Figure above shows the graphical properties of an object when working with UnrealEd. AmbientGlow and ScaleGlow parameters are those most important when it comes to light in UnrealEd. The first relates to the amount of light falling on all sides of the object in question, while the second expresses how the object is affected by being hit by a light source nearby. AmbientGlow parameter is set to value 70, because this will get the minimum amount of self-illumination as explained above. ScaleGlow parameter depends very much on the available external light and, in general, values between 0.4 and 0.7 have been successful for both TI building's interior and its external environment.

Below are the most important parameters that define UnrealEd external lights:

LightColor

LightBrightness: Light brightness.

LightHue: Light tone. With this parameter we can get different colored lights.

LightSaturation: This factor acts on the base color of light making it a brighter color or a more desaturate color. When we desaturate a picture, we obtain a grayscale one.

Lighting

LightCone: This parameter determines the angles in which the light is emitted from its insertion point.

LightRadius: Range of light.

LightType: Light type. You can define several types of light such as static or intermittent.

The following figure shows the average values that have been assigned to the above parameters, after doing many tests in the virtual environment and studying their results. However, depending on the location, number of lights and the size of the area to be lit, these values have changed, especially LightBrightness and LightRadius because of they are prone to produce overexposure in areas close to the point of light emission.

LightColor	
LightBrightness	33.867199
LightHue	0
LightSaturation	255
Lighting	
bActorShadows	Falso
bAttenByLife	Falso
bCorona	Falso
bDirectionalCorona	Falso
bDynamicLight	Falso
bLightingVisibility	Ver.
bSpecialLit	Falso
LightCone	255
LightEffect	LE_None
LightPeriod	32
LightPhase	0
LightRadius	67.734398
LightType	LT_Steady

Fig. 41. UnrealEd lights parameters

6.3. Movable objects

In many experiments, a robot must recognize an object, chase, grab or manipulate it with a predefined purpose. To meet these needs, we have modeled some inanimate objects, but movable. They are mainly toys with simple shapes and kinematics that can serve as a basis for developing more complex objects, depending on the needs of each experiment. Thus, we have introduced in the virtual environment a soccer ball, a candy, a punching bag and a dispenser of objects operated by robots

6.3.1. Soccer ball

To start with something simple, we have introduced a soccer ball, so that the robots of the simulations have an object that can be pushed from side to side. With this goal we have taken the diameter of a real soccer ball and its color scheme, so that it looks as real as possible. Then, from 3D Studio Max we have given shape and color to it (see figure 42) and, later, we have associated a spherical collision envelope to the ball. Done this, we have placed the geometric center of the ball and its envelope at the point of coordinates X = 0 Y = 0 Z = 0 and then we have exported both ball and envelope in ASCII format, generating the soccer_ball.ase file.

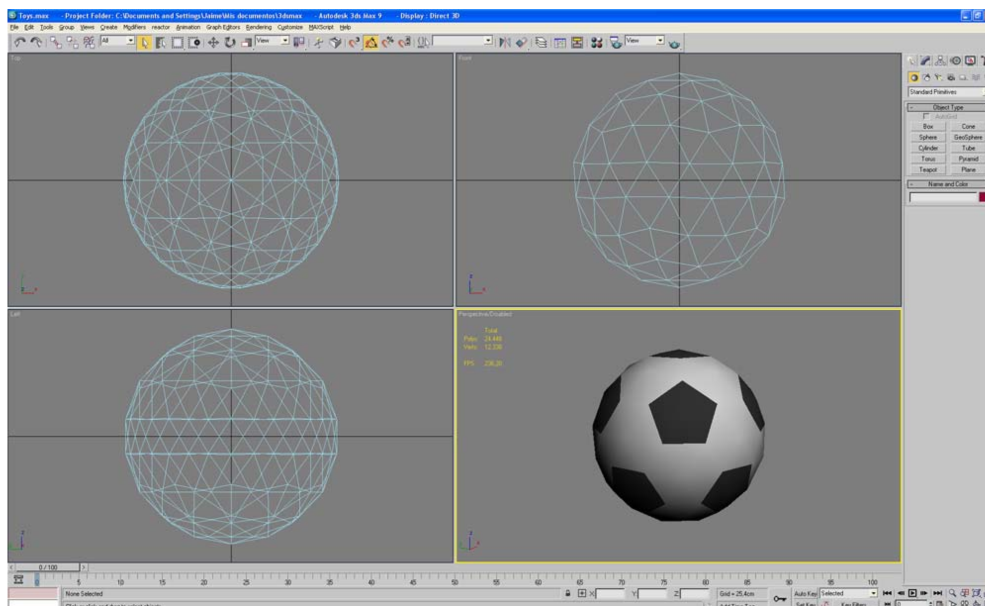

Fig. 42. 3D Studio Max screenshot showing a soccer ball at (0, 0, 0) point

At this point, it is important to note that the location of the geometric center of an object at the origin of coordinates is advantageous when it is expected to undergo operations of rotation or translation from UnrealEd. Thus, during the introduction of this object in the virtual world, UnrealEd will take the center of the object as both insertion point and axis of rotation. This avoids the following problem: when rotating some objects in UnrealEd, they have whether gone out

of the 3D view or seemed to be moving linearly instead of rotating, all due to rotation axes far away from those objects.

To continue with the process of generation of mobile inanimate objects, we have created from UnrealEd a file (in this case Toys.utx) to keep them all. Then from the *StaticMesh Browser*, the ball has been imported (soccer_ball.ase) as *ball* within the group *Soccer_Ball* of Toys.utx file as shown in figure 43.

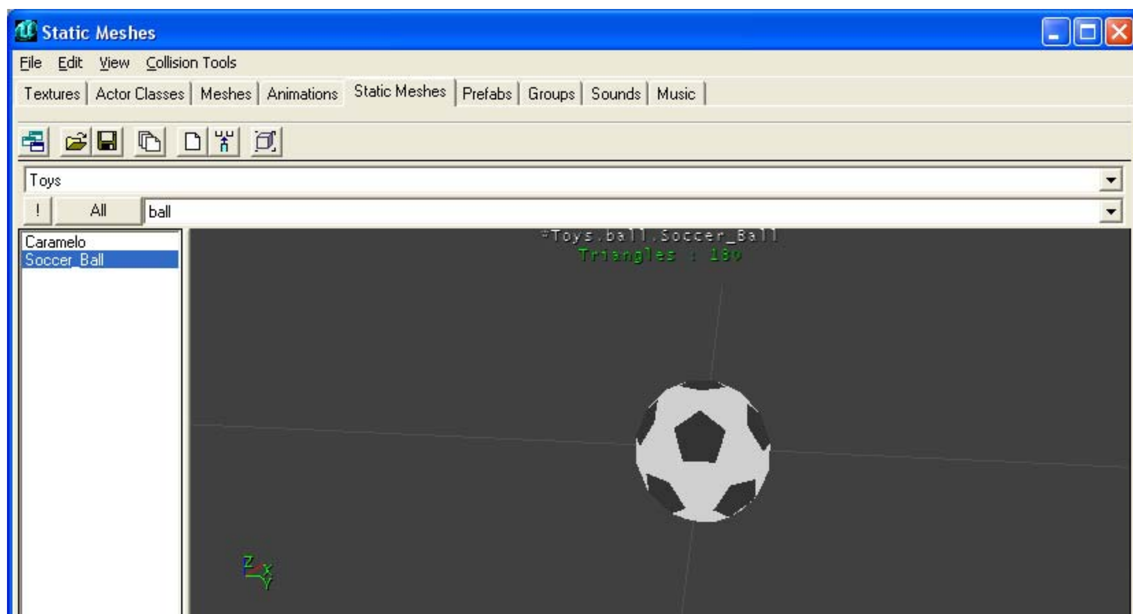

Fig. 43. UnrealEd Static Mesh Browser showing the soccer ball

Finally, the ball was inserted in the virtual lab as an *Actor*, because it is not a static object. Next, we had to make the ball not only looks good physically, but also reacts appropriately to the forces acting on it. This was achieved by adjusting the KarmaParams within the properties of the *Actor* in question as follows:

KarmaParams

bHighDetailOnly: *False*

KActorGravScale: *1.0*

KAngularDamping: *0.7*

KBuoyancy: *0.2*

KLinearDamping: *0.7*

KMass: *0.2*

KarmaParamsCollision

KFriction: *0.7*

KRestitution: *1.0*

The explanation of *KarmaParams* can be found in paragraph relative to the pushable doors.

6.3.2. Candy

The next object in order of simplicity is a "candy", which consists of a long cylindrical body ending in two circular caps slightly larger in diameter. In this case, the dimensions were chosen so that a robot equipped with a gripper can grab the entire length of the object as if it were a can of beer.

As in the case of the ball, we have elaborated by means of photoshop a texture that permits to determinate whether the object is rotating around one of its axes. The texture of red and white stripes spiral perfectly fulfills this role and has served to give the name "candy".

From 3D Studio Max we have given shape and texture to the candy, and also a cylindrical collision envelope. Then, we have placed the geometric centers of the candy and its collision envelope at the point of coordinates X = 0 Y = 0 Z = 0 (see figure 44) and then we have exported them in ASCII format, generating the *caramelo.ase* file.

Similar to the case of the soccer ball, we have opened the UnrealEd *StaticMesh Brower* and then the file called *Toys.utx* in order to import the candy (*caramelo*.ase) into the group named *caramelo*.

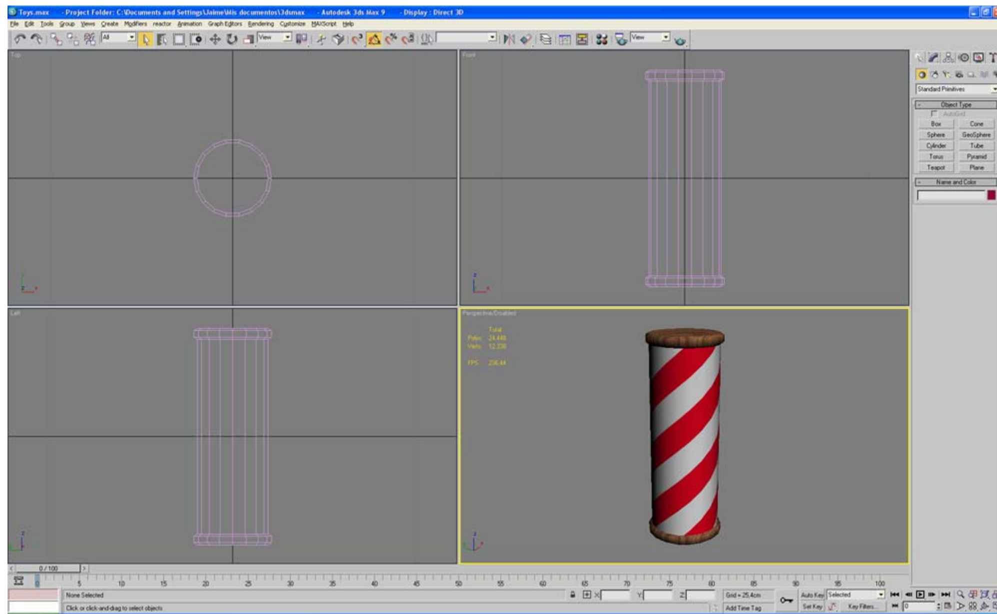

Fig. 44. Candy and its collision hull at coordinates (0, 0, 0)

Finally, we have inserted the candy in the virtual lab as an Actor and we have adjusted its KarmaParams from UnrealEd Actor properties as shown bellow. It should be noted that these parameters have been defined so that the candy has little weight, slides easily on the floor and has some ability to bounce off any surface.

KarmaParams

bHighDetailOnly: *False*
KActorGravScale: *0.6*
KAngularDamping: *0.1*
KBuoyancy: *0.8*
KLinearDamping: *0.2*
KMass: *0.2*

KarmaParamsCollision

KFriction: *0.6*
KRestitution: *1.0*

6.3.3. Punching bag

In order to exploit the simulation capabilities of USARSim, it was decided to create a punching bag (see figure 45). This way we can simulate the motion of a pendulum to be beaten by a robot, which introduces restrictions on the movement possibilities of movable objects, opening a wide range of possibilities for future experiments In this case, although the actual dimensions have been taken for both the bag and for the support, is likely to have to make changes in them to bring this subject to potential needs.

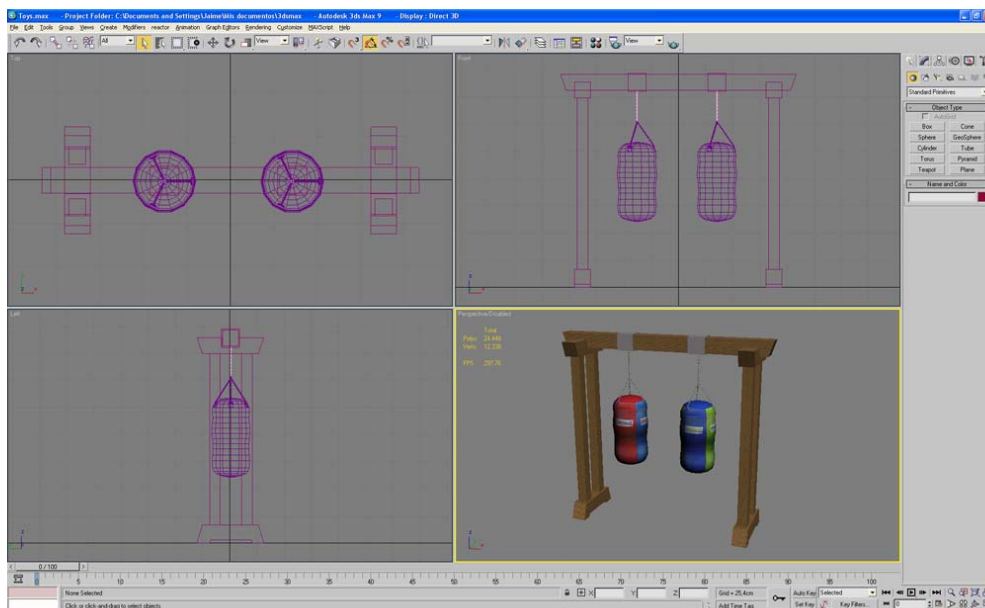

Fig.45. 3D Studio Max screenshot showing the punching bag

Unlike the above objects, the punching bag is not composed of a single element, but three: the bag, the chains and the structure. However, his process of creation doesn't differ much of that of the previous objects. Thus, textures of sailcloth, wood and metal have been created in Photoshop for the bag, the structure and the chains, respectively. Next, we have modeled these three elements in a single file of 3D Studio Max, choosing appropriate collision envelopes (see figure 46) and then we have exported each element separately generating the files *Estructura_Punching.ase*, *Chain.ase*, *Punching_Bag.ase* and *Punching_Bag2.ase*. As can be seen, two bags were created to take

advantage of a comprehensive structure which allows a robot to easily maneuver (see figure 45).

The following figure shows the collision envelopes used for each of the parts of the punching bag. Notice that those parts not expected to be hit don't have collision envelope (upper bolster of the wooden structure).

Fig 46. Punching bag elements with their collision hulls

Once the components of the punching bag have been exported to ASCII format, from UnrealEd we have opened the file *Toys.utx* and imported them into the group *Punching_Bag* with the names: *Chain*, *Estructura_Punching*, *Punching_Bag* and *Punching_Bag2*.

Next, these objects have been inserted into the virtual lab. Whereas the strings and the bag have been introduced as *Actor*, the structure has been introduced as *StaticMesh*, since it supposed to be static. Therefore, there has only been to adjust the *KarmaParams* of initial length of the chain (element called *Chain*) and the group called *Punching_Bag*, which are detailed below.

Chain

KarmaParams

bHighDetailOnly:	*False*
KActorGravScale:	*1.0*
KAngularDamping:	*0.2*
KBuoyancy:	*0.0*
KLinearDamping:	*0.2*
KMass:	*0.1*

KarmaParamsCollision

KFriction:	*0.0*
KRestitution:	*0.0*

Punching_Bag

KarmaParams

bHighDetailOnly:	*False*
KActorGravScale:	*1.0*
KAngularDamping:	*0.8*
KBuoyancy:	*0.0*
KLinearDamping:	*1.0*
KMass:	*2.0*

KarmaParamsCollision

KFriction:	*0.0*
KRestitution:	*0.0*

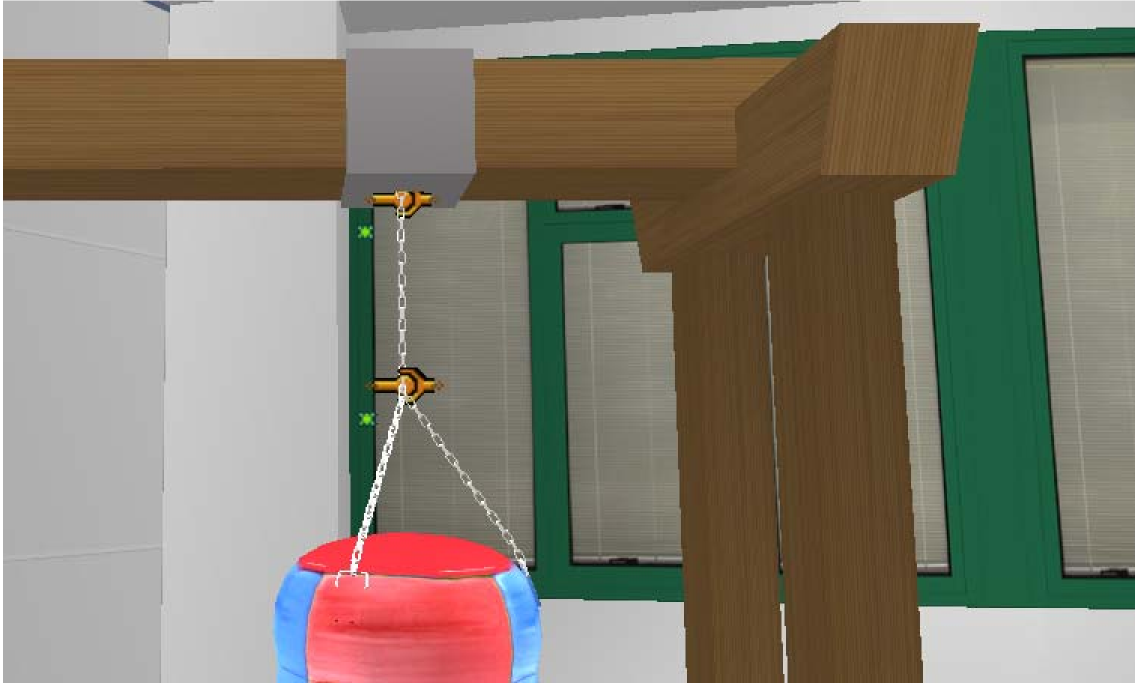

Fig. 47. Punching bag movement constraints

Later, we added the movement constraints for the punching bag moving parts such as the first length of chain (*Chain*) and the bag with the second length of chains (*Punching_Bag*) (see figure 47). These restrictions have been materialized by means of two conical joints, which correspond to the class *KBSJoint* that is located at the same subgroup of the *Actor Classes Browser* as *KHinge*. This latter constraint has been previously used for some doors.

Finally, for each constraint (joint) we have set the parameters that determine the actors whose movement is being restricted as described below.

Conical joint 1 (Estructura_Punching-Chain joint)

KarmaConstraint

KConstraintActor1: KActor'myLevel.KActor8'

Conical joint 2 (Chain-Punching_Bag joint)

KarmaConstraint

KConstraintActor1: KActor'myLevel.KActor8'

KConstraintActor2: KActor'myLevel.KActor9'

Conical joint 1 only restricts the movement of the first length of chain (*Chain*). By default this length of chain has its parameter *Name* set to '*KActor8*'.

Conical joint 2, by contrast, restricts the movement of two elements (*Chain* and *Punching_Bag*) which have their *Name* parameters set to *KActor8* and *KActor9* respectively.

6.3.4. Toy dispenser

In order to allow users to choose which object they want to interact with, we designed a dispenser of toys. It consists of a ramp that ends in a horizontal surface where there are two objects (toys), which can be forced to fall down the slope by means of two handles located on the sides of the ramp (see figure 48).

We have given to the static part of the mechanism a look of wood and a collision envelope composed of several boxes (Box). Moving the handles have a metal look with rubber coated ends and simple collision envelopes (two boxes). We have also added two axes to restrict the possibilities of movement of the handles. The creation process for this mechanism is very similar to that followed to obtain some doors.

Fig. 48. Toy dispenser

6.4. MATLAB functions

With all the elements necessary for virtual testing finished and in place, the next step was to use the MATLAB USARSim Toolbox package to establish a flexible interface between the user and USARSim. Thus, we have implemented navigation and databases management functions to check both the robot's mobility and the flow of information collected by its sensors, being at the discretion of each user to implement specific functions for each particular experiment.

7. RESULTS

The simulator result of this work meets the goals set at the beginning of this document and is currently available to the researchers at the University Jaume I in Castellón.

Table III shows the sensitive detail difference (measured in triangles) between the obtained model of the ERA robot and the P2DX available in the platform USARSim.

Table III. Robot detail measured in triangles

Robot Part	P2DX	ERA
Body	449	17569
Computer	0	4690
Laser	224	2968
Left wheel	200	11524
Right wheel	200	11524
Rear Wheel	200	4124
TOTAL	1273	52399

Figure 49 compares the performance of the obtained simulator (measured in frames per second) using two different computers. The first one is the laptot

mentioned before (Nvidia 7400 Go), whereas the second is a desktop computer provided with a Nvidia GeForce 7950 GTX video card, 4 Gb of RAM memory and a processor Intel Core2 Quad Q9300 2500GHz. As it can be observed, the computing speed is around 35 fps in the worst case scenario.

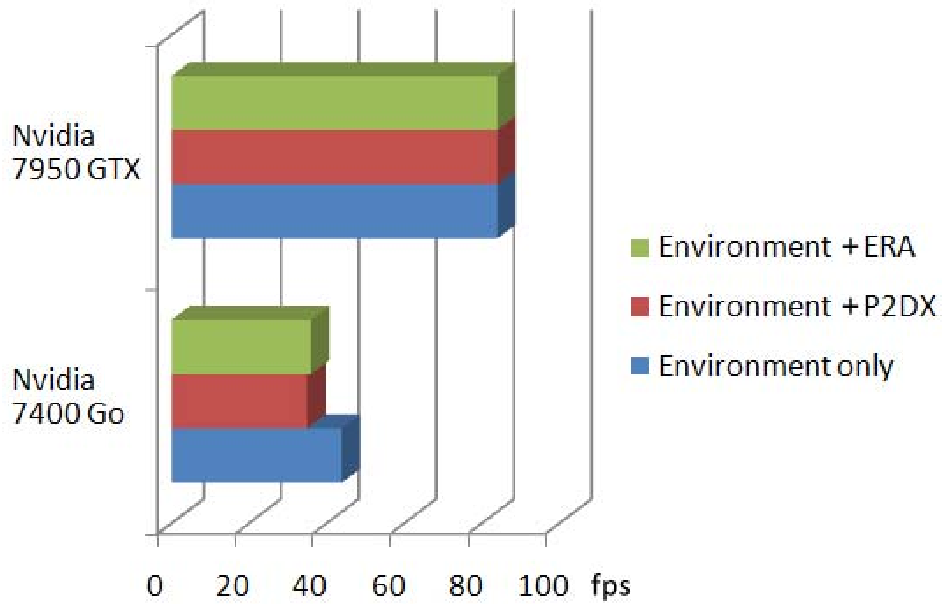

Figure 49. Performance comparison

Figure 50 shows an external view of the virtual TI building and its surroundings, while Figure 51 presents the setting of this building as seen from the ground floor hall.

Fig. 50. Virtual TI building and part of its immediate surroundings

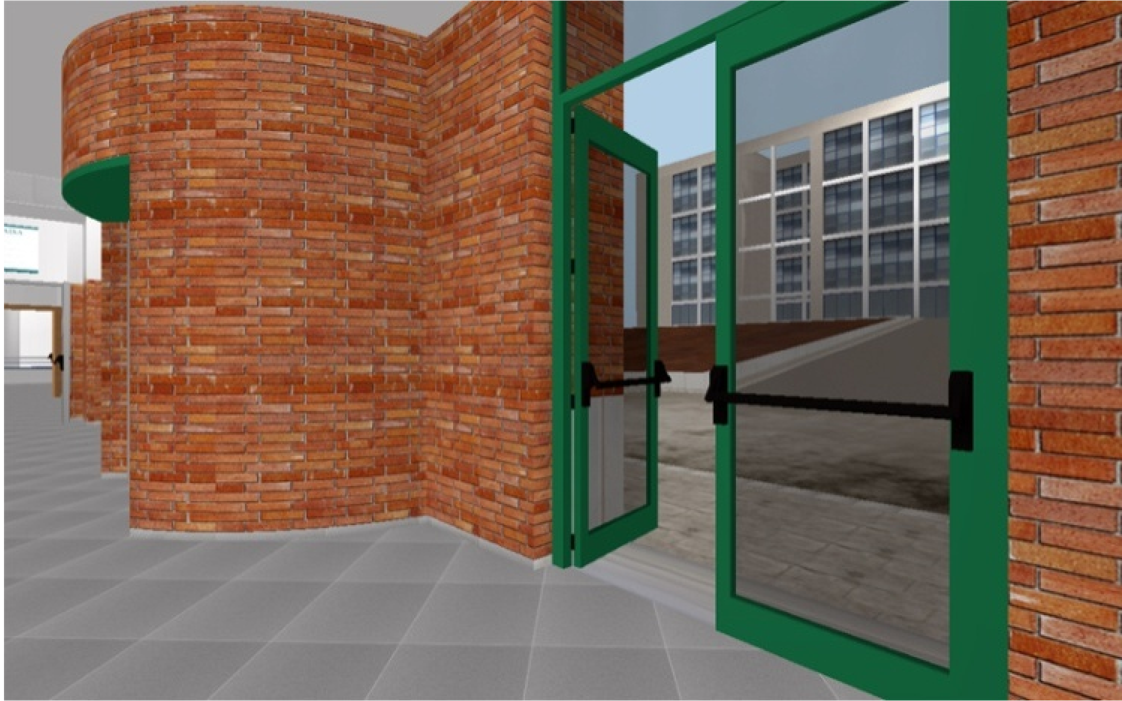

Fig. 51. Virtual library building seen from TI building ground floor hall.

8. CONCLUSIONS AND FUTURE WORK

Three years ago it was decided to use USARSim and MATLAB as a basis for developing a high graphical quality simulator adapted to educational needs identified by the Engineering and Computer Science Department at University Jaume I in Castellón. Over time it has been demonstrated that the conjunction of those tools with 3D Studio Max and Adobe Photoshop allows 3D simulators to be created whose limits exist only in the imagination of their users.

Given the versatility of the present simulator a large number of possible improvements and future applications may be suggested. The ones which have the clearest priority are the following:

- Increasing the number of available robots including all those commonly used by students and researchers of the Engineering and Computer Science Department at the University Jaume I in Castellón.
- Introducing the specific furniture of each laboratory where experiments with robots are going to be carried through.
- Modeling the 3D objetcs needed to perform each test.
- Improving the lighting method for both interiors and exteriors in order to obtain one more similar to that of the real world.
- Creating objects with a center of mass configurable and independent of its geometric center.
- Programming MATLAB functions so that those actions most commonly required by robotics students and researchers can be carried out.

9. REFERENCES

[1] (2008) Iros'08: Workshop on robot simulators available software, scientific applications and future trends. [Online]. Available: http://www.robot.uji.es/research/events/iros08

[2] B. P. Gerkey, R. T. Vaughan, K. Stoy, A. Howard, G. S. Sukhatme and M. J. Mataric, "Most valuable player: a robot device server for distributed control," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1226-1231 vol. 3, 2001.

[3] MobileSim http://robots.mobilerobots.com/wiki/MobileSim

[4] L. Hughes and N. Bredeche, "Simbad: An Autonomous Robot Simulation Package for Education and Research," in *From Animals to Animats 9*, Springer LNCS vol. 4095, pp. 831-842, 2006.

[5] O. Michel, "Webots: Professional Mobile Robot Simulation," *International Journal of Advanced Robotic Systems*, vol. 1, num.1, pp. 39-42, 2004.

[6] "Unreal Tournament® at Epic," http://www.epicgames.com/ , 2011.

[7] Allison Mathis, Kingsley Fregene and Brian Satterfield, "Creating High Quality Interactive Simulations Using MATLAB® and USARSim," in *proceedins of the IROS Workshop Robots, Games and Research: Success Stories in USARSim,* 2009.

[8] S.Carpin, M. Lewis, Jijun Wang, S. Balakirsky and C. Scrapper, "USARSim: a robot simulator for research and education," *Robotics And Automation, 2007 IEEE International Conference on,* vol., no., pp. 1400-1405, 10-14 April 2007.

[9] C. Scrapper, S. Balakirsky and E. Messina, "MOAST and USARSim – A Combined Framework for the Development and Testing of Autonomous Systems," in *Proc. SPIE Defense and Security Symposium,* Orlando, 2006.

[10] S. Carpin, A. Birk, M. Lewis and A. Jacoff, "High fidelity tools for rescue robotics: results and perspectives," in *Robocup International Symposium 2005,* 2005.

[11] S. Carpin, T. Stoyanov, Y. Nevatia, M. Lewis and J. Wang, "Quantitative assessments of USARSim Accuracy", *Proceedings of PerMIS,* 2006.

[12] Autodesk, 3D Studio Max: 3D rendering software. http://www.autodesk.com/3dsmax.

[13] Karma User Guide http://udn.epicgames.com/Two/rsrc/Two/KarmaReference/KarmaUserGuide.pdf

[14] "Wolf's Tutorials," http://unreal.gamedesign.net/tutorials/5/wolftut5.html 2011.

[15] MATLAB USARSim ToolBox http://robotics.mem.drexel.edu/USAR/

[16] "Importing Karma Actors," http://udn.epicgames.com/Two/ImportingKarmaActors.html, 2011.

[17] M. Lewis, "USARSim and HRI: from Teleoperated Cars to High Fidelity Teams"