USARSim V3.1.3

A Game-based Simulation of mobile robots

Prepared by Jijun Wang
Edited by Stephen Balakirsky

USARSim

Contents

USARSim ii
1 Please Tell Us About Your Project 1
2 Introduction 1
2.1 Background 1
2.2 What is USARSim 1
3 System Overview 3
3.1 System architecture 3
3.1.1 Unreal engine 4
3.1.2 Gamebots 4
3.1.3 Controller 4
3.2 Simulator components 5
3.2.1 Environment simulation 5
3.2.2 Sensor and effecter simulation 9
3.2.3 Robot simulation 10
3.2.4 Communications Simulation 10
3.2.5 Image Server 13
3.2.6 MultiView 14
4 Installation 20
4.1 Requirements 20
4.2 Install UT2004 20
4.2.1 Windows 20
4.3 Install USARSim 22
Linux 23
4.4 Install the controller 23
4.4.1 MOAST 24
4.4.2 Pyro 25
4.4.3 Player 26
5 Run the simulator 28
5.1 The steps to run the simulator 28
5.2 Examples 29
5.2.1 The testing control interface 29
5.2.2 MOAST 30
5.2.3 Pyro 31
5.2.4 Player 32
5.2.5 SimpleUI 33
5.3 Getting Starting Poses From Maps 36
5.3.1 Introduction 36
5.3.2 Adding starting poses to the map 36
5.3.3 Retrieving starting poses 43
5.3.4 Start Elevation 44
5.3.5 Standard Z starting value 44
5.3.6 Z Table 45

6 Coordinates, Units and Scale 45
6.1 Coordinates 46
6.2 Units and scale 47
7 Mission Package 48
8 Effecters 49
9 Communication & Control (Messages and commands) 50
9.1 TCP/IP socket 50
9.2 The protocol 50
9.3 Messages 51
9.4 Commands 67
10 Sensors 78
10.1 State Sensor 78
10.1.1 How the sensor works 78
10.1.2 How to configure it 78
10.2 Range Sensor 78
10.2.1 How the sensor works 78
10.2.2 How to configure it 79
10.3 Range Scanner Sensor 80
10.3.1 How the sensor works 80
10.3.2 How to configure it 80
10.4 Odometry Sensor 81
10.4.1 How the sensor works 81
10.4.2 How to configure it 82
10.5 GPS Sensor 82
10.5.1 How the sensor works 82
10.5.2 How to Configure it 83
10.6 INS Sensor 86
10.7 Encoder Sensor 88
10.7.1 How the sensor works 88
10.7.2 How to configure it 88
10.8 Touch Sensor 89
10.8.1 How the sensor works 89
10.8.2 How to configure it 89
10.9 RFID Sensor 90
10.9.1 How the sensor works 90
10.9.2 How to configure it 90
10.9.3 Choosing the right SensingMode 91
10.9.4 Detecting RFID Tags 92
10.9.5 Reading RFID Tags Memory 93
10.9.6 Writing RFID Tags Memory 93
10.9.7 Erasing RFID Tags Memory 93
10.10 Victim and False Positive Sensor 94
10.10.1 How the sensor works 94
10.10.2 How to configure it 94
10.11 Sound sensor 95
10.11.1 How the sensor works 95

10.11.2 How to configure it 95
10.12 Human-motion sensor 96
10.12.1 How the sensor works 96
10.12.2 How to configure it 96
10.13 GPS Sensor 96
10.13.1 How the sensor works 96
10.14 Robot Camera 97
10.14.1 How the sensor works 97
10.14.2 How to configure it 98
10.15 Omnidirectional Camera 98
10.15.1 How the sensor works 99
10.15.2 How to mount it 99
10.15.3 How to configure it 100
11 Effecters 100
11.1 Gripper 100
11.1.1 How the effecter works 100
11.1.2 How to configure it 101
11.2 RFID Releaser 101
11.2.1 How the effecter works 101
11.2.2 How to configure it 102
11.3 Roller Table 102
11.3.1 How the effecter works 102
11.3.2 How to configure it 102
11.4 103
11.5 Headlight 103
12 Robots 103
12.1 P2AT 103
12.1.1 Introduction 103
12.1.2 Configure it 104
12.2 StereoP2AT 106
12.2.1 Introduction 106
12.2.2 Configure it 106
12.3 P2DX 106
12.3.1 Introduction 106
12.3.2 Configure it 107
12.4 ATRVJr 107
12.4.1 Introduction 107
12.4.2 Configure it 108
12.5 HMMWV (Hummer) 108
12.5.1 Introduction 108
12.5.2 Configure it 109
12.6 SnowStorm 109
12.6.1 Introduction 109
12.6.2 Configure it 110
12.7 Sedan 110
12.7.1 Introduction 110

12.7.2 Configure it 111
12.8 Cooper 111
12.8.1 Introduction 111
12.8.2 Configure it 112
12.9 Submarine 112
12.9.1 Introduction 112
12.9.2 Configure it 113
12.10 Tarantula 113
12.10.1 Introduction 113
12.10.2 Configuration 114
12.11 Zerg 114
12.11.1 Introduction 114
12.11.2 Configuration 115
12.12 Talon 115
12.12.1 Introduction 115
12.12.2 Configure it 115
12.13 QRIO 116
12.13.1 Introduction 116
12.13.2 Configure it 116
12.14 ERS 116
12.14.1 Introduction 116
12.14.2 Configure it 117
12.15 Soryu 117
12.15.1 Introduction 117
12.15.2 Configure it 118
12.16 Kurt2D 118
12.16.1 Introduction 118
12.16.2 Configure it 119
12.17 Kurt3D 119
12.17.1 Introduction 119
12.17.2 Configure it 120
12.18 Lisa 120
12.18.1 Introduction 120
12.18.2 Configure it 121
12.19 TeleMax 121
12.19.1 Introduction 121
12.19.2 Configure it 123
12.20 AirRobot 123
12.20.1 Introduction 123
12.20.2 Configure it 124
12.21 Passarola 124
12.21.1 Introduction 124
12.21.2 Configure it 125
12.21.3 Extended USARSim command for Passarola robot 125
12.22 Rugbot 126
12.23 Kenaf 127

12.23.1 Introduction 127
12.23.2 Configure it 128
13 Controller 129
13.1 MOAST 129
13.2 Pyro 132
13.2.1 Simulator and world 132
13.2.2 Robots 133
13.2.3 Services 134
13.2.4 Brains 135
13.3 Player 135
13.3.1 Simulation and device configuration 136
13.3.2 Device Drivers 138
14 Advanced User 142
14.1 Build your arena 142
14.1.1 Geometric model 143
14.1.2 Special effects 144
14.1.3 Obstacles and Victims 144
14.2 Build your sensor 146
14.2.1 Overview 147
14.2.2 Sensor Class 147
14.2.3 Writing your own sensor 148
14.3 Build your effecter 149
14.3.1 Overview of the Effecter.uc class 149
14.3.2 Writing your own effecter 150
14.4 Build your robot 151
14.4.1 Step1: Build geometric model 151
14.4.2 Step2: Construct the robot 152
14.4.3 Step3: Customize the robot (Optional) 156
14.5 Build your controller 158
14.5.1 Embedding Unreal Client 158
14.5.2 Capturing Unreal Client 159
14.5.3 Using the Image Server 160
15 Information for Gamers 161
16 Bug report 161
17 Contributors 161
18 Acknowledgements 163

1 Please Tell Us About Your Project

We are constantly trying to improve USARSim. Part of this effort requires us to understand how this package is being used. We would greatly appreciate it if you could please fill out a brief survey that may be found at http://usarsim.sourceforge.net/pages/volunteer/Survey.html. This survey will be used to aid us in getting further support for the development of this package. It tells us who you are, what you are using the package for, and any suggested improvements. Thanks!

2 Introduction

This manual is written for version 3.1.2 of USARSim. The files may be found at the file release section of the USARSim web site (http://sourceforge.net/projects/usarsim). To install the base release, please check the code out of cvs (explained later in this document) or download the USARSim full archive. Maps and tools are also available on the website.

2.1 Background

Large-scale coordination tasks in hazardous, uncertain, and time stressed environments are becoming increasingly important for fire, rescue, and military operations. Substituting robots for people in the most dangerous activities could greatly reduce the risk to human life. Because such emergencies are relatively rare and demand full focus on the immediate problems there is little opportunity to insert and experiment with robots.

2.2 What is USARSim

USARSim was designed as a high fidelity simulation of urban search and rescue (USAR) robots and environments intended as a research tool for the study of human-robot interaction (HRI) and multirobot coordination. Since its initial release, it has been expanded to support many diverse environments including highway robots, the DARPA urban challenge, robotic soccer, submarines, humanoids, and helicopters. USARSim is designed as a simulation companion to the National Institute of Standards' (NIST) Reference Test Facility for Autonomous Mobile Robots for Urban Search and Rescue (Jacoff, et al. 2001). The NIST USAR Test Facility is a standardized disaster environment consisting of three scenarios: Yellow, Orange, and Red physical arenas of progressing difficulty. The USAR task focuses on robot behaviors, and physical interaction with standardized but disorderly rubble filled environments. USARSim supports HRI by accurately rendering user interface elements (particularly camera video), accurately representing robot automation and behavior, and accurately representing the remote environment that links the operator's awareness with the robot's behaviors.

High fidelity at low cost is made possible by building the simulation on top of a game engine. By offloading the most difficult aspects of simulation to a high volume commercial platform which provides superior visual rendering and physical modeling, our full effort can be devoted to the robotics-specific tasks of modeling

platforms, control systems, sensors, interface tools and environments. These tasks are in turn, accelerated by the advanced editing and development tools integrated with the game engine leading to a virtuous spiral in which a widening range of platforms can be modeled with greater fidelity in less time.

The current release of the simulation consists of: various environmental models (levels), models of commercial and experimental robots, and sensor models. As a simulation user, you are expected to supply the user interfaces, automation, and coordination logic you wish to test. For debugging and development "Unreal spectators" can be used to provide egocentric (attached to the robot) or exocentric (third person) views of the simulation. A test control interface is provided for controlling robots manually. Robot control programs can be written using the GameBot interface, MOAST System (http://moast.sourceforge.net/), Player interface, or Pyro middleware (please note that the pyro interface is out of date and not supported)..

3 System Overview

3.1 System architecture

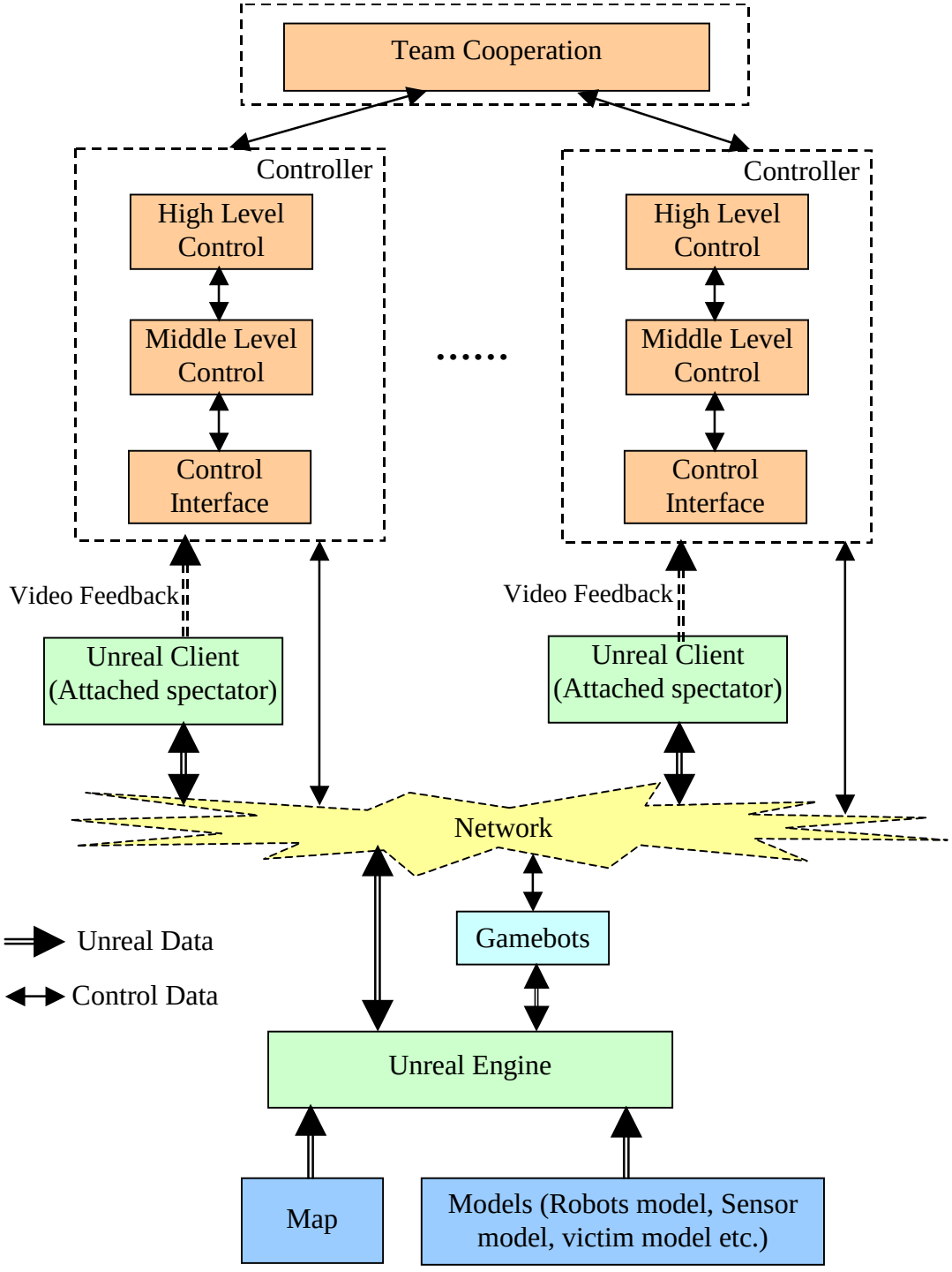


Figure 1: System Architecture

The system architecture is shown in Figure 1. Below the dashed boxes is the simulator that provides the interactive virtual environment for the users. The dashed box is the user side where you can use the simulator to aid in your research. The

system uses a client/server architecture. Above the network icon in Figure 1, is the client side. It includes the Unreal client and the controller or the user side applications. The Unreal client renders the simulated environment. In the Unreal client, through changing the viewpoint, we can get the view of the robot in the environment. All the clients exchange data with the server through the network. The server side is called the Unreal server. It includes the Unreal engine, Gamebots, the map, and the models (such as robot models, victim models, etc.). The Unreal server maintains the states of all the objects in the simulator, responds to the data from the clients by changing the objects' states and sends back data to both Unreal clients and the user side controllers.

In summary, the three main components that construct the system are 1) the Unreal engine that makes the role of server, 2) the Gamebots that communicates between the server and the client and 3) the Control client that controls the robots on the simulator.

3.1.1 Unreal engine

The Unreal engine used in the simulator is released by Epic Games (http://www.epicgames.com/) with Unreal Tournament 2004. Please note that a full license for the Unreal Tournament 2004 game is required (cost of about $40 at most software retailers) (http://www.unrealtournament.com/ut2004/). The demonstration version will not work with USARSim. It's a multiplayer combat-oriented first-person shooter for the Windows, Linux and Macintosh platforms. In addition to the amazing 3D graphics provided by the engine, the physics engine, which is known as the Karma engine, is also included in Unreal to obtain high quality reality. Unreal engine also provides a script language, Unreal Script, to the game developers to develop their own games. With the scripts, developers can create their objects (we call them actors) in the game and control these actors' behaviors. Unreal Editor is the 3D authoring tool that comes with the Unreal engine to help developers build their own maps, geometric meshes, terrain etc. For more information about Unreal engine, please visit the Unreal Technology page: http://www.unrealtechnology.com/html/technology/ue2.shtml.

3.1.2 Gamebots

The communication protocol used by Unreal engine is proprietary. This makes accessing Unreal Tournament from other applications difficult. Therefore, Gamebots (http://www.planetunreal.com/gamebots/), a modification to Unreal Tournament, is built by researchers to bridge Unreal engine with outside applications. It opens a TCP/IP socket in Unreal engine and exchanges data with the outside. USARSim enables Gamebots to communicate with the controllers. To support our own control commands and messages, some modifications are applied to Gamebots.

3.1.3 Controller

Controller is the user side application that is used for your research, such as robotics study, team cooperation study, human robot interaction study etc. Usually, the controller works in this way. It first connects with the Unreal server. Then it sends command to USARSim to spawn a robot. After the robot is created on the simulator,

the controller listens to the sensor data and sends commands to control the robot. The client/server architecture of Unreal makes it possible to add multiple robots into the simulator. However, since every robot uses a socket to communicate, for every robot, the controller must create a connection for it.

The Mobility Open Architecture Simulation and Tools (MOAST) framework is designed to allow researchers to concentrate their efforts in their area of expertise. To accomplish this, the framework provides a hierarchical, modular set of controllers, interfaces, and tools. The controllers conform to the hierarchical 4-D/RCS Reference Model Architecture (Albus, 2000) and provide behavior generation, world modeling, and sensor processing. The hierarchy supports control ranging from low-level servo control to high-level robot team control. To utilize this framework, experimental code connects to one or more of the standardized interfaces to obtain data from the robot(s) and exert control. MOAST is developed to fully integrate with the USARSim simulation system. More information about MOAST is located at http://moast.sourceforge.net/. A detailed explanation of the MOAST interfaces may be found in section 13.1.

Besides MOAST, USARSim also supports two other popular robot controllers, Pyro (out of date) and Player. The Pyro plug-in included in USARSim allows the use of Pyro to control the robot in the simulator. Pyro (http://pyrorobotics.org/) is a Python library, environment, GUI, and low-level drivers used to explore AI and robotics. The details of the Pyro plug-in are described in section 13.2.

The USARSim Player drivers are the device drivers that allow the control of robots and sensors in the simulator through Player as if they were real physical devices. Player is a robot device server that gives users simple and complete control over the sensors and actuators on the robot. For more information please visit Player website: http://playerstage.sourceforge.net/. A detailed explanation of the USARSim Player drivers can be found in section 13.3.

3.2 Simulator components

The core of the USARSim is the simulation of the interactive environment, the robots, and their sensors and effecters. We introduce the three core components separately in the following sections.

3.2.1 Environment simulation

Environment plays a very important role in simulations. It provides the context for the simulation and only with it, can the simulation make sense. Several specialized environments that are distributed for use with USARSim are described below. Users and developers are free to create additional usage areas for the simulation.

3.2.1.1 USAR Environment

USARSim was originally based upon simulated disaster environments in the Urban Search and Rescue (USAR) domain. The environments are simulations of the National Institute of Standards and Technology (NIST) Reference Test Facility for Autonomous Mobile Robots (http://www.isd.mel.nist.gov/projects/USAR/). NIST built three test arenas to help researchers evaluate their robot's performance.

These arenas are built from the AutoCAD models of the real arenas. To achieve high fidelity simulation, the textures used in the simulation are taken from the real environment. For all of the arenas, the simulated environments include:

- *Geometric models*: the model imported from the AutoCAD model of the arenas. They are the static geometric objects that are immutable and unmovable, such as the floor, wall, stairs, ramp etc.
- *Obstacles* simulation: that simulates the objects that can move and change their states. In addition, these objects can also impact the state of a robot. For example, they can change a robot's attitude. These objects include bricks, pipes, rubble etc.
- *Light* simulation: that simulates the light environment in the arena.
- *Special* effects *simulation*: that simulates the special items such as glass, mirrors, grid fenders etc.
- *Victim* simulation: is the simulation of victims that can have actions such as waving hands, groaning, and other distress actions.

All the virtual arenas are built with Unreal Editor. With it, users can build their own environment. For details please read section 14.1. In addition to the USAR arenas, outdoor areas and simulated collapsed buildings have been modeled. All of these arenas are available for download at http://sourceforge.net/projects/usarsim in the files area.

The real USAR arenas and simulated arenas are listed below:

The yellow arena: the simplest of the arenas. It is composed of a large flat floor with perpendicular walls and moderately difficult obstacles.

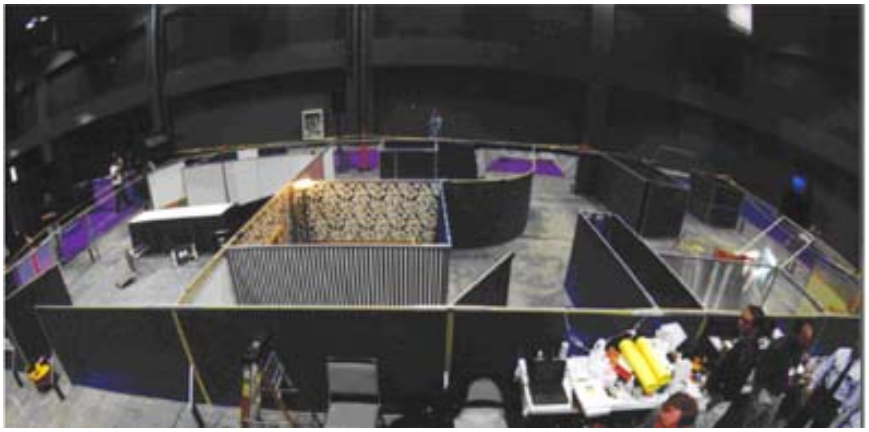


Figure 2: Yellow arena

Figure 3: Simulated yellow arena

The orange arena: a bi-level arena with more challenging physical obstacles such as stairs and a ramp. The floor is covered with debris including paper, pipes, and cinder blocks.

Figure 4: Orange arena

Figure 5: Simulated orange arena

The red arena: presents fewer perceptual difficulties but places maximal demand on locomotion. There are rubble piles, cement blocks, slabs and debris on the floor.

Figure 6: Red Arena

Figure 7: Simulated red arena

3.2.1.2 Road Environment

In addition to urban search and rescue, the simulator has been applied to the DARPA Urban Challenge (http://www.darpa.mil/grandchallenge). This is supported by the ARDA map, ARDA_RNDF.txt, and ARDA_MDFx.txt files. Formats for the Route Network Definition Files (RNDF) and Mission Description Files (MDF) are described on the DARPA web site. Manual control of a robot does not require a RNDF or MDF. Autonomous control or game scoring by DARPA rules will require those files. The robot SnowStorm is equipped with sensors to drive autonomously in this environment.

3.2.2 Sensor and effecter simulation

Sensors are important to robot control. Through checking the object's state or some calculation in the Unreal engine, three kinds of sensor are simulated in USARSim.

- Proprioceptive sensors

  These include battery state and headlight state.

- Position estimation sensors

  These include location, rotation, and velocity sensors.

- Perception sensors

  These include sonar, laser, pan-tilt-zoom (ptz) camera, touch sensor, and RFID tag reader.

All of the sensors in USARSim are configurable. A sensor can be easily mounted on the robot by adding a line into the robot's configuration file. When a

sensor is mounted, its name, type, position where it's mounted, and the direction it will face can be specified. For every kind of sensor, specific properties can be specified. Examples of these include the maximum range of the sonar, the resolution of the laser and FOV (field of view) of the camera. For more information about configuring a sensor please see section 10. For details of mounting a sensor on the robot please see section 12.

Effecters are very similar to sensors. They can be configured and mounted on the robot. However, instead of sending sensor data to the user, the main function of an effecter is to accept a command and execute the corresponding function in the virtual world. Currently, only headlights and RFIDReleaser effecters exist. The details of effecter can be found in section 0. How to equip an effecter is explained in section 12.

3.2.3 Robot simulation

Using the Karma rigid-body physics engine, which is embedded in Unreal Tournament 2004, we built a robot model to simulate the mechanical robot. The robot model includes chassis, parts (tires, linkage, camera frame etc.), and other auxiliary items such as cameras, headlights, etc. All the chassis and parts are connected through simulated joints that are driven by torques. Three kinds of joint control are supported in the robot model. The zero-order control makes the joint rotate by a specified angle. The first-order control lets the joint rotate under the specified rotational speed. The second-order control applies the specified torque on the joint. To help better organize and control these parts and joints, we introduced a mission package concept. A mission package represents a container of parts and joints. Sensors and effecters are connected to the robot platform through the mission packages. For instance, the camera pan-tilt frame is a kind of mission package that connects a camera to the robot. By controlling the pan-tilt frame, we can adjust the camera's pose. The robot receives the control command and sends out data through Gamebots.

With this robot model, users can build a new robot with little or no Unreal Script programming. For the steps of building your own robot, please read section 14.3.

In USARSim, a total of eight ground robots are provided for you: P2AT, P2DX, ATRV-Jr, Zerg, Tarantula, Talon, Telemax, and Soryu. In addition, USARSim includes four Ackerman-steered vehicles for outdoor scenes as well as testing driving algorithms: Hummer, Sedan, SnowStorm, and Cooper. Two legged robots, the QRIO and ERS, are also part of the simulation as well as a nautical vehicle (Submarine) and an aerial vehicle (Helicopter). Information about these robots can be found in section 12.

3.2.4 Communications Simulation

The purpose of the Wireless Communications Server is to act as a middle man for messages passed between the robots, dropping messages and connections between robots when not realistically feasible using wireless communication. It has been implemented in UnrealScript, and is automatically started when starting a BotDeathMatch (and hence a UsarDeathMatch).

The server listens on a port for connections from robots sending command messages for registering, listening and opening connections. Once connections between robots are set up, these are handled on different sockets (TcpLinks), allowing the server to listen for more commands, and allowing multiple connections to be handled.

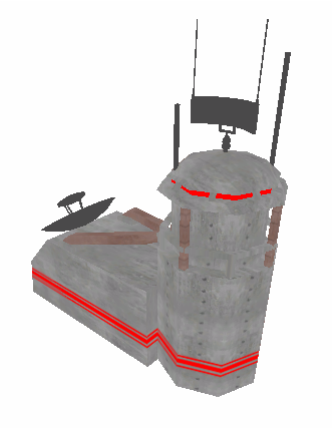
The opening of a connection or the closing of a connection while sending a message is decided using the path loss estimated using the Wall Attenuation Factor Model.

$$P(d)[dBm] = P(d_0)[dBm] - 10n\log(\frac{d}{d_0}) - \begin{cases} nW * WAF & nW < C \\ C * WAF & nW \geq C \end{cases}$$

The default values for $P(d_0)$ is -49.67 dBm, d_0 is 2 m, n is 1.09, C is 5 and WAF is 6.625, based on measurements taken in Research 1, at International University Bremen[1]. When the path loss reaches below -93 dBm the connections are closed/the attempt at opening a connection fails.

3.2.4.1 Communication Base Station:

A communication base station is provided with USARSim and should be used as a relay point for your robots. The communication base station is added into a world by issuing the following command: INIT {ClassName USARBot.ComStation} {Location *x,y,z*}. If you are unfamiliar with the INIT command, please read section 7.4.



3.2.4.2 Configuration:

Several options can be changed using the USARComServer.ini file.

For the **USARBotAPI.ComServer** class:
ListenPort: the port the server listens on
bDebug: Boolean flag indicating the printing of log messages

For the **USARBotAPI.ComConnection** class:

[1] Jacobs University Bremen as of Spring 2007

bDebug: Boolean flag indicating the printing of log messages
ePdo: The signal strength at a reference distance d_0, in dBm
eDo: The reference distance d_0
eN: The log factor n
eCutoff: The cutoff signal strength, in dBm
eMaxObs: The maximum number of obstacles (C in the formula above)
eAttenFac: The signal attenuation for each obstacle (WAF in the formula above)

For the **USARBotAPI.ComLink** class:
bDebug: Boolean flag indicating the printing of log messages

3.2.4.3 Message formats:

Registering:

REGISTER *RobotName IPAddress*;

The *RobotName* should be the same as the one the robot was created with in the simulation. The *IPAddress* is the IP Address of the computer on which the robot controller is running. The REGISTER word is not case sensitive.

Listening:

LISTEN *RobotName Port*;

The *RobotName* should be the one with which the robot was registered, and *Port* the port at which the program is listening. The LISTEN word is not case sensitive.

Opening a connection:

OPEN *RobotName Port HostRobot HostPort;*

RobotName is the robot that you want to connect to, and is listening for connections at *Port*. *HostRobot* is the robot that wants to setup the connection, and it should be listening at *HostPort* before sending this command to the server, in order for the connection to be successfully opened (if the signal strength is suitable). The OPEN word is not case sensitive.

Sending a message:

SEND *MessageLength Message*;

Once a connection has been established, this is used to send *Message* (with *MessageLength* characters) to the other endpoint of the connection. This supports binary message strings. The SEND word **is** case sensitive.

Closing a connection:

CLOSE;

This is used to close a connection. The CLOSE word **is** case sensitive.

Getting the path loss between two robots:

GETSS *Robot1 Robot2*;

This can be sent to the WSS over the control connection to get the path loss between *Robot1* and *Robot2*. The GETSS word is not case sensitive.

Responses:
All except the GETSS, SEND and CLOSE command get one of two responses from the server.
If the command is successfully carried out, the response is

OK;

If the command could not be carried out, the response is

Fail: *ErrorMessage*;

where *ErrorMessage* is a message describing why the command could not be carried out.
When GETSS is successfully carried out, the response is

OK:*PathLoss*;

Where *PathLoss* is the path loss between the robots in dBm. If it fails the response is a fail message as described above.

3.2.4.4 Sample Programs

Very simple sample programs using the server can be found in the USARSim CVS repository (http://sourceforge.net/cvs/?group_id=145394), under the folder Tools/WSS.

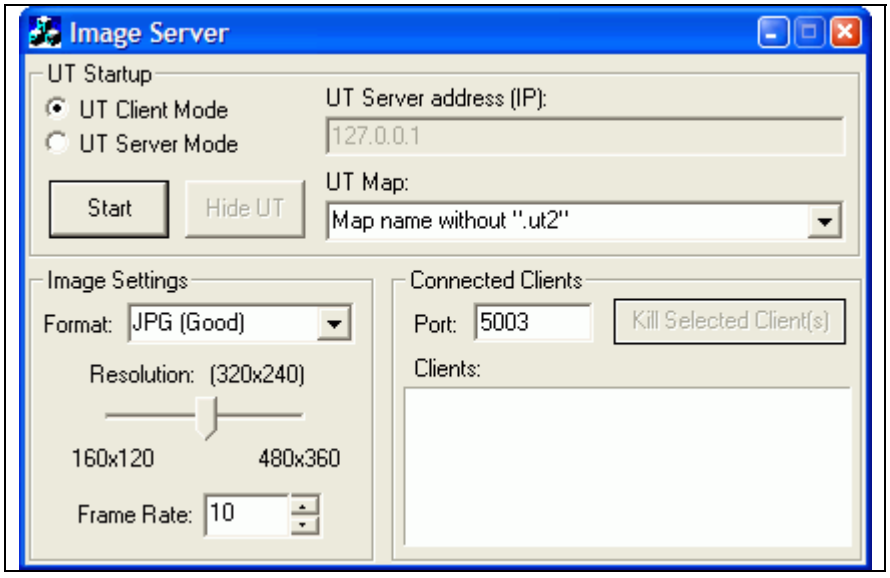


Figure 8: Image server control interface.

3.2.5 Image Server

The image server is a windows application that allows one to capture images from any of the cameras that are being used inside of the game engine. It may be run with UT running in either “client” or “server” mode. The control display from the image server may be seen in Figure 8.

3.2.5.1 Installation

The image server may be obtained from CVS or from the downloads page under the Tools area on Sourceforge as an install package.

3.2.5.2 Running the Image Server with USARSim in Server Mode

If you want to use the image server in the usual way do the following:

1. Start the UT server.
2. Start the image server.
3. In the image server select **UT Server Mode**, type in **UT Server address** IP, enter a map name and chose the format, resolution and fps of the images.
4. Click on Start.

> Remember: you must run image server on the same machine where UT server is running.

3.2.5.3 Running the Image Server with USARSim in Client Mode

If you plan to use USARSim in client mode, then do the following:

1. Start the image server.
2. In the image server select **UT Client Mode**, enter a map name and chose the format, resolution and fps of the images.
3. Click on Start.

3.2.6 MultiView

The USARSim client allows one to only see from the camera(s) of one robot at a time. To overcome this limitation, a special extension called MultiView was developed.

3.2.6.1 Enabling a map for MultiView

In order to enable the MultiView you have to add a special object into the map. You must use the Unreal Editor to do that. Open your map in the Unreal Editor (As shown in Figure 9, we will use the Soccer map as an example):

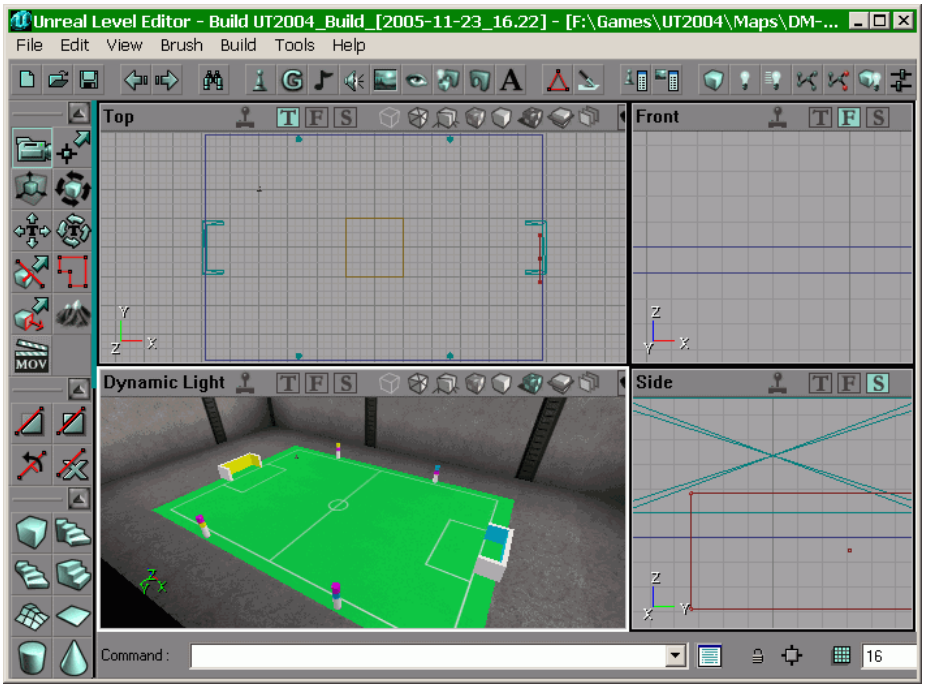


Figure 9: Soccer map in the unreal editor

Click on the pawn button as illustrated in Figure 10. It will open the Actor Classes window:

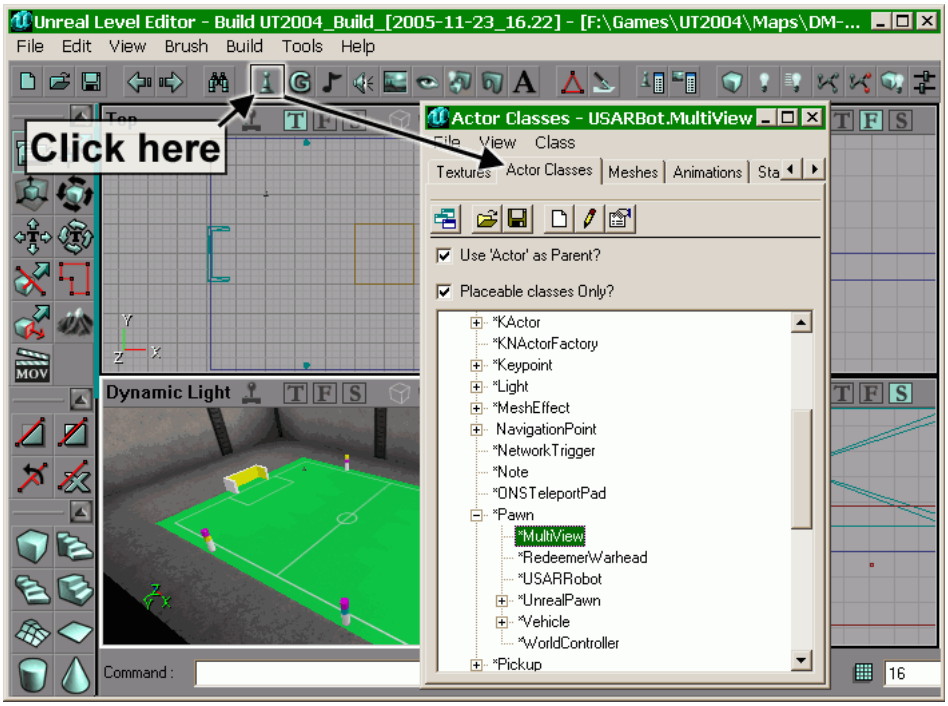


Figure 10: Click on the "pawn" button.

In the tree view search for MultiView and select it. The path to it is Actor/Pawn/MultiView. Now right click somewhere in the map (in the perspective view) and select "Add MultiView Here". This is shown in Figure 11.

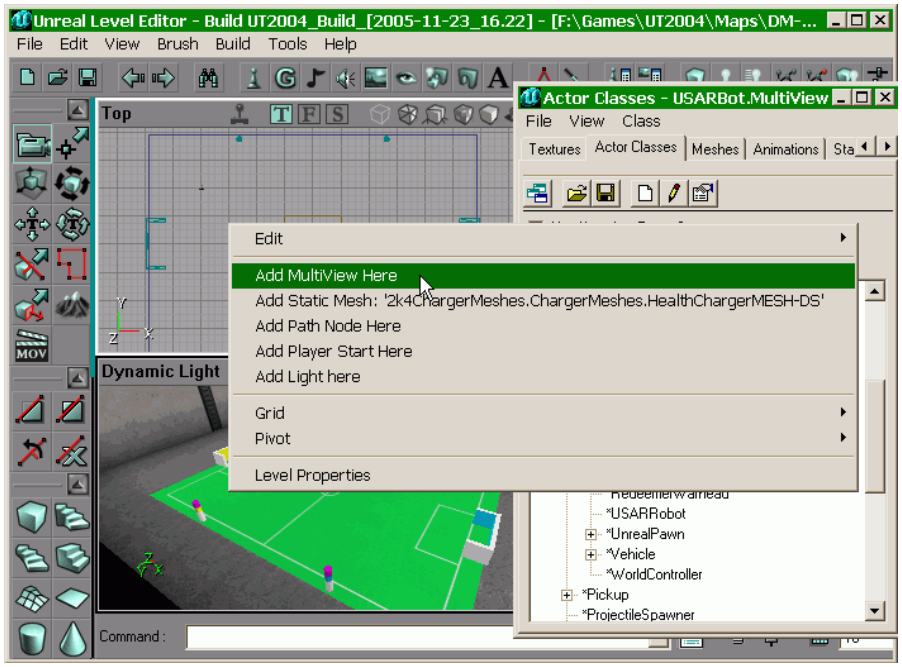


Figure 11: Adding the multiview pawn.

Close the Actor Classes view. The box shown in should have appeared in the map:

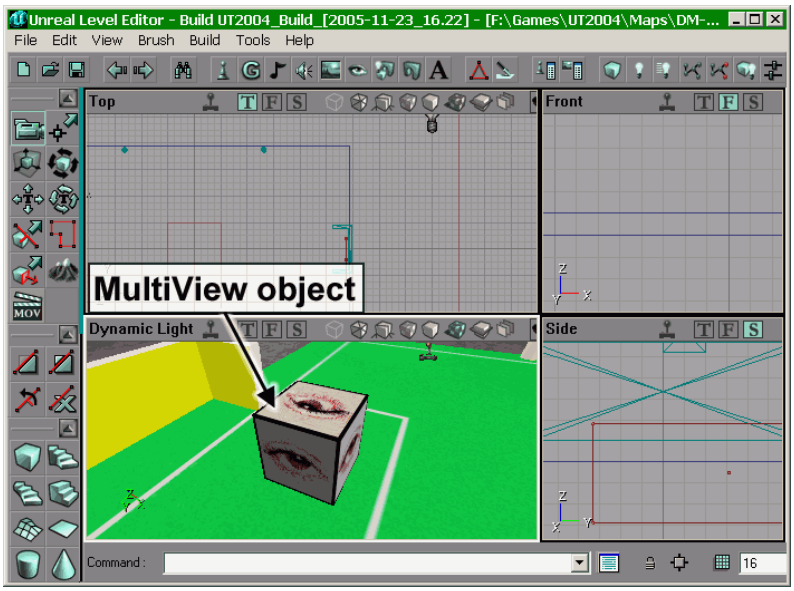


Figure 12: MultiView object added to map.

You can place it wherever you want, for example near the ceiling so to hide it from robot cameras. Now click on the rebuild button to rebuild the map as shown in Figure 13 and save the map.

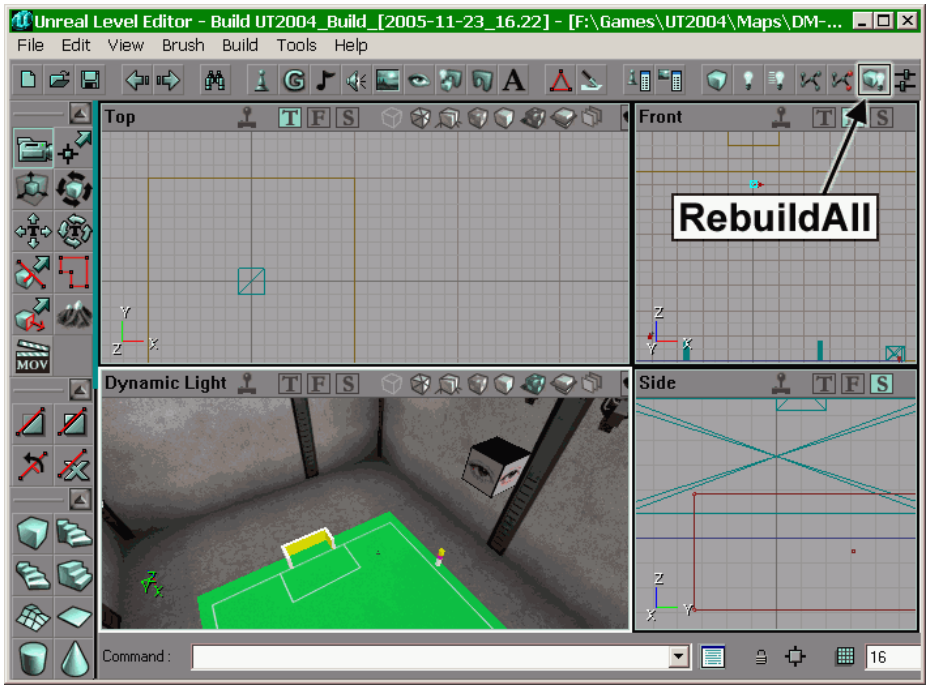


Figure 13: Rebuilding the world.

3.2.6.2 Using the MultiView

As shown in Figure 14, with MultiView you can capture the camera view from many robots at the same time.

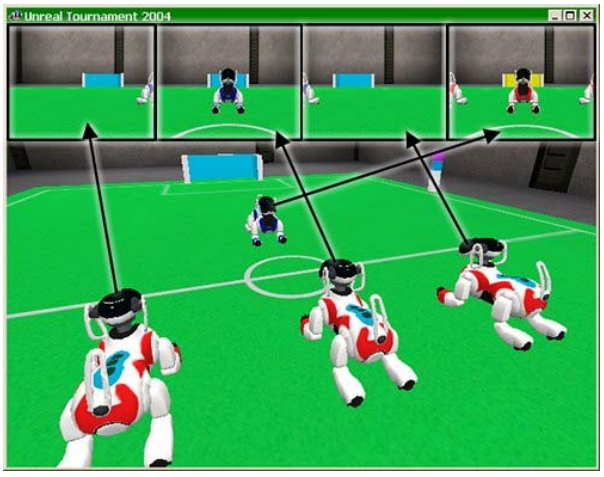


Figure 14: Multiple robot views using MultiView.

Just keep left clicking in the client until you find the MultiView view. You can write your own capturing image server, extract the images of each robot and send them to respective clients, as illustrated in Figure 15.

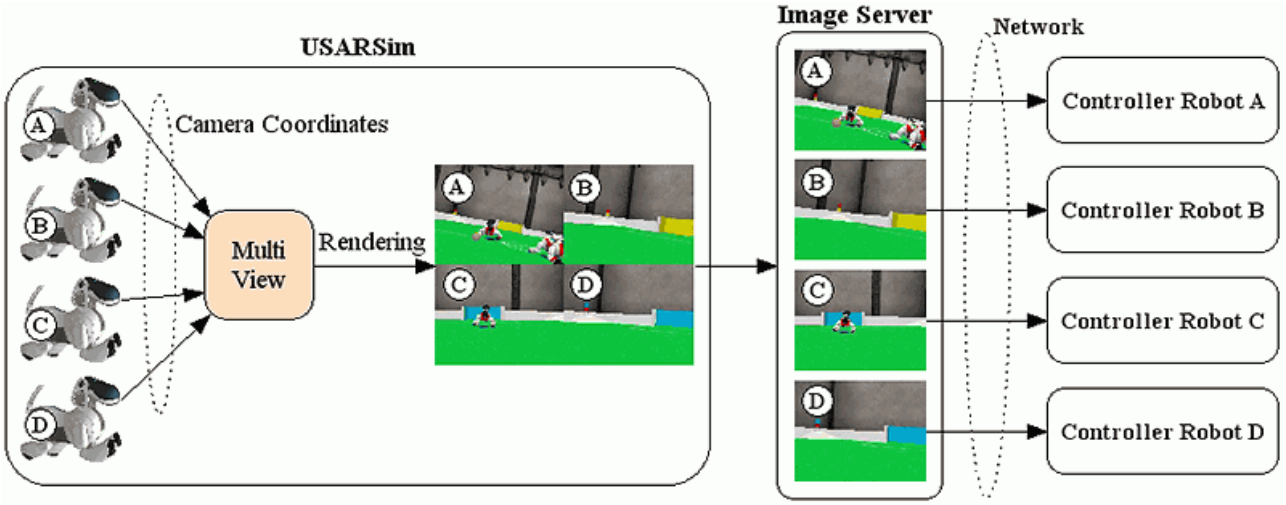


Figure 15: Example of using MultiView with a controller.

Otherwise you can use the image server that comes with USARSim and split the camera images at client side.

MultiView **can also render stereo cameras** in each subview. To do so you have to add this function to your robot class (subclassed from KRobot):

```
//(Called by KRobot Tick function)
//Stereo vision when viewing from multiview robot camera
simulated function SyncMultiView() {
  if(ViewManager != none)
  {
    if(isStereo)
        ViewManager.UpdateView(ViewNum, myCamera.Location,
myCamera.Rotation,
                               CameraZoom, true, uuStereoSpacing);
    else
        ViewManager.UpdateView(ViewNum, myCamera.Location,
myCamera.Rotation,
                               CameraZoom, false);
  }
}
```

where:

- **isStereo** : is a boolean variable you can use to enable/disable stereo rendering. Obviously, if you plan to always use stereo camera then you can

delete the **if**(isStereo) statement. You can declare this variable **config** so to be able to set its value inside *USARBot.ini* file.

- **uuStereoSpacing** : is a **vector** that defines the (eye to eye)/2 spacing in unreal units. 1m = 250 unreal units, so you can make a manual conversion. Otherwise you can use the converter to automatically convert a stereo spacing parameter expressed in meters to unreal units. To do so you have to add the following function in you robot class:

```
simulated function ConvertParam(USARConverter converter) {
  Super.ConvertParam(converter);
  if(converter!=None)
  {
    uuStereoSpacing = converter.LengthVectorToUU(stereoSpacing);
  }
  else
  {
    uuStereoSpacing = stereoSpacing;
  }
}
```

where:

- **stereoSpacing** : is a **vector** that defines the (eye to eye)/2 spacing in meters.

You can define a vector in defaultproperties like this:

```
defaultproperties {
  ...
  stereoSpacing=(Y=0.05) //5 cm: (eye to eye)/2 spacing in meter
  ...
}
```

3.2.6.3 Configuration

You can configure the MultiView in the *USARBot.ini* file. You will find the **[USARBot.MultiView]** section that contains the following parameters:

- **CameraNum** : maximum number of cameras that can be drawn at the same time.
- **WideMode** : if 0 the MultiView will split the screen using a box pattern. You can see in Fig.7 the pattern used for 4 cameras. If > 0 then the MultiView will use that number of columns before adding a raw. In Fig.6 you can see an example with WideMode=4.

- **CameraXres** : X resolution of robot camera. The final resolution of the camera subview will be <= CameraXres. If the overall resolution of camera subviews is larger than what can fit in the USARSim client, they will be scaled down to fit the screen. If for example you use 4 cameras in WideMode=4, each 200 pixel wide, and the USARSim client resolution is 640x480, each camera width will be scaled down to 160 pixel. But if you set WideMode=0 a different patter is used and the full camera resolution (200 pixel) will be used.
- **CameraYres** : Y resolution of robot camera. Same as above but for Y axis. In any case the aspect ratio (CameraXres/CameraYres) will be preserved. That means that a change in resolution will always affect both X and Y.
- **IsCameraLocked** : Don't consider this for now and leave it to **false**. It can be used with zone optimization to speed up subview rendering.

3.2.6.4 Limitations

There are two important known limitations:

1. MultiView supports only one camera (can be stereo) per robot. So, if you have a robot that mounts more than one camera (like Talon) you will see only from its first camera.
2. More cameras you use (same as saying: more robots you use) smaller will be the subviews. This can, or can not be a problem. Depends on the specific application.

4 Installation

4.1 Requirements

Operating System: Windows 2000/XP or. Linux
Software: UT2004 with the 3339 or later patch
Optional requirements: For the controller, we recommend MOAST, which is fully integrated with USARSim. In addition, Pyro (out of date) or Player may be used. The Pyro plug-in requires Pyro 2.2.1, and the Player USARSim drivers require Player 1.4rc2 or higher.

4.2 Install UT2004

4.2.1 Windows

Please note: if you have a previous version of USARSim installed, it must be uninstalled.
Users have three options to remove previous USARSim versions. Please note that %UT20024% refers to the Unreal Tournament main folder.

a. Use the Uninstaller (USARSim Full version 3.00 or higher only)
---

1. Run the Uninstall-USARSimFull.exe file located in the %UT2004% directory

b. Re-Install Unreal Tournament

---

1. Uninstall Unreal Tournament by going to
   Start->Programs->Unreal Tournament 2004->Uninstall Unreal Tournament 2004

2. Manually delete the %UT2004% folder from your computer.

3. Install Unreal Tournament.

4. Install the Official Unreal Tournament Patch V3369, which you can download at:
   http://data.unrealtournament.com/UT2004-WinPatch3369.exe

c. Manually Delete Files

---

1. Manually delete the following folders, if they exist:
   %UT2004%\Doc\
   %UT2004%\Tools\
   %UT2004%\BotAPI\
   %UT2004%\USARBot\
   %UT2004%\USARBotAPI\
   %UT2004%\USARMisPkg\
   %UT2004%\USARModels\
   %UT2004%\USARVictims\

2. Manually delete the following files, if they exist:
   %UT2004%\Animations\UDN_CharacterModels_K.ukx
   %UT2004%\Sounds\SEERVoices.uax
   %UT2004%\StaticMeshes\FreiburgRescue.usx
   %UT2004%\StaticMeshes\IUB_meshes.usx
   %UT2004%\StaticMeshes\spqrMapsMeshes.usx
   %UT2004%\StaticMeshes\spqrRobotMeshes.usx
   %UT2004%\StaticMeshes\TalonMeshes.usx
   %UT2004%\StaticMeshes\USAR_Robots.usx
   %UT2004%\StaticMeshes\USARSim_Hummer.usx
   %UT2004%\StaticMeshes\USARSim_Sedan.usx
   %UT2004%\StaticMeshes\USARSim_Submarine.usx
   %UT2004%\StaticMeshes\Sub.usx
   %UT2004%\StaticMeshes\USARSim_Cars.usx
   %UT2004%\StaticMeshes\Hummer.usx
   %UT2004%\StaticMeshes\USARSim_Vehicles_Meshes.usx
   %UT2004%\StaticMeshes\USARSim_VehicleParts_Meshes.usx
   %UT2004%\StaticMeshes\USARSim_Objects_Meshes.usx
   %UT2004%\StaticMeshes\USARSim_LeggedRobots_Meshes.usx
   %UT2004%\System\Hook.dll
   %UT2004%\System\make.bat

%UT2004%\System\make.csh
%UT2004%\System\usar_c.bat
%UT2004%\System\usar_s.bat
%UT2004%\System\usar_t.bat
%UT2004%\System\USARBot.ini
%UT2004%\System\USARBotAPI.ini
%UT2004%\System\USARSim.ini
%UT2004%\System\BotAPI.ini
%UT2004%\Textures\IUB_textures.utx
%UT2004%\Textures\SEERS_girls.utx
%UT2004%\Textures\SEERS_gWounded.utx
%UT2004%\Textures\SEERS_Wounded.utx
%UT2004%\Textures\SEERSTextures.utx
%UT2004%\Textures\SEERSTexturesSG.utx
%UT2004%\Textures\spqrMapsTextures.utx
%UT2004%\Textures\spqrRobotTextures.utx
%UT2004%\Textures\TalonTexture.utx
%UT2004%\Textures\USAR.utx
%UT2004%\Textures\submarine.utx
%UT2004%\Textures\USARSim_Cars.utx
%UT2004%\Textures\Hummer.utx
%UT2004%\Textures\USARSim_Vehicles_Textures.utx
%UT2004%\Textures\USARSim_VehicleParts_Textures.utx
%UT2004%\Textures\USARSim_Objects_Textures.utx
%UT2004%\Textures\USARSim_LeggedRobots_Textures.utx
%UT2004%\ChangeLog
%UT2004%\README

4.3 Install USARSim

For non-developers:

1) Install Unreal Tournament. You will get a folder, which we will refer to as %UT2004% for these instructions.
2) Install the Official Unreal Tournament Patch V3369, which you can download at: http://data.unrealtournament.com/UT2004-WinPatch3369.exe
3) Download the usarsim-2004 files in the download section of the USARSim project page: http://sourceforge.net/project/showfiles.php?group_id=145394
4) Extract the files you just downloaded into the %UT2004% folder.
5) Compile USARSim by running the "make.bat" script in the %UT2004%/System folder.

For developers:

1) Install Unreal Tournament. You will get a folder, which we will refer to as %UT2004% for these instructions.
2) Install the Official Unreal Tournament Patch V3369, which you can download at: http://data.unrealtournament.com/UT2004-WinPatch3369.exe
3) Temporarily rename the %UT2004% folder to "usarsim". Put yourself in the folder above "usarsim".

4) Check out the latest source code snapshot from SourceForge. For instructions on doing this, go to: http://sourceforge.net/cvs/?group_id=145394
5) This will check out the USARSim source code snapshot into your "usarsim" folder, merging it with what's already there.
6) Change "usarsim" back to %UT2004%.
7) Download the base files in the download section of the USARSim project page: http://sourceforge.net/project/showfiles.php?group_id=145394
8) Extract the base files you just downloaded into the %UT2004% folder.
9) Compile USARSim by running the "make.bat" script in the %UT2004%/System folder.
10) From this point on, you can run periodic "cvs update" commands to get the latest snapshot, and recompile as in (8).

Linux

6) Install UT2004
7) Install the patch
    a. Download ut2004 patch at http://www.unrealtournament.com/ut2004/downloads.php
    b. Download and run the shell script http://www.hetepsenusret.net/files/ut2k*/ut2k4-patch to install the patch. For more details about usage, please run
        i. $ ut2k4-patch --help

NOTE: To make the code under windows, run the file *make.bat* that is located in the system directory. For linux users, the file make.csh should be executed.

NOTE: In addition to the worlds, the file *AAA_MapBaseFiles_VX.XX.zip* located under the BaseFiles release of the Maps package is necessary for most worlds.

There is a testing control interface written in C++. If you don't want to install any controller software, you can copy USAR_UI to your machine and try it. USAR_UI may be found under the 'Tools' section of the Files release area on sourceforge. USAR_UI only works on Windows. You can use it to send commands to USARSim and investigate all the messages received from the Unreal server.

NOTE: When you restore the zipped file, please make sure it is restored under the correct directory. If your directory structure looks something like …\ut2004\ut2004, you need to move all the files under …\ut2004\ut2004 to …\ut2004.

4.4 Install the controller

This step is optional. Install a controller only when you want to use MOAST, Pyro (out of date) or Player to control a robot in USARSim.

4.4.1 MOAST

MOAST fully supports USARSim. MOAST and all of its related packages may be retrieved from sourceforge. Additional packages that must be downloaded include gtki (image extensions to gtk used for graphical user interfaces), and rcslib (an interprocess communications package). These packages may be found on the release section of the MOAST site (http://sourceforge.net/project/showfiles.php?group_id=148555). The rcslib archive is available as either source or pre-compiled for cygwin.

The MOAST code may be retrieved from CVS. To retrieve the code, enter the following commands:

```
$ cvs –d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/moast login
$ cvs –z3 –d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/moast co
–P devel
```

Information about accessing this CVS repository may be found in the document titled, "CVS (Version Control for Source Code)". Alternatively, the latest snapshot may be retrieved from the release section of the MOAST site.

MOAST requires a Linux style environment. This may be obtained by running a Linux operating system or by running cygwin under windows. More information on installing cygwin for a MOAST windows installation may be found under the "What else do I need" topic found at:
http://moast.sourceforge.net/Column%20With%20Contents.htm#_What_else_do.

In addition to the Linux style environment, a communications package known as rcs must be installed. This is available from the files area of the MOAST sourceforge repository in both source and pre-compiled cygwin formats. To install the pre-compiled version, simply download from http://www.sourceforge.net/project/showfiles.php?group_id=148555 and then unpack.

NOTE: When unpacking the pre-compiled version, use the following commands:

```
$ cd /
$ tar –zxvf path_to_rcslib.tgz/rcslibV0.2-cygwin.tgz
```

To install the source:
Install the rcs library at /usr/local directory

```
$ cd /
$ tar –zxvf path_to_rcslib.tgz/rcslibV0.2-src.tgz
$ cd /usr/local/rcslib
$ ./configure
$ make
$ make install prefix=/usr/local/rcslib
$ cd src/java
$ make
```

If you want to use the MOAST supplied GUI, you need to make sure GTK and GTKI are installed on your machine. GTK is a standard package and will be automatically installed during the cygwin installation and is usually installed on Linux operating systems. GTKI may be found in the file release area of the MOAST

repository. It should be downloaded and installed in the /usr/local/src directory. The following commands may be used to install GTKI:

```
$ cd /usr/local/src
$ tar –zxvf path_to_gtki-1.9.tgz/gtki-1.9.tgz
$ cd gtki-1.9.0
$ ./configure
$ make
$ make install
```

Then you need to set the PKG_CONFIG_PATH environment variable to tell the system where GTKI is located.

> TIP: Use the following command to set PKG_CONFIG_PATH:
> `$ export PKG_CONFIG_PATH = ”/path_to_gtki”`

MOAST may now be installed. Download the tar file from SourceForge or check the code out of CVS, and then execute the following commands:

```
$ cd /path_to_moast/
$ ./configure
$ make
```

> NOTE: If checking out from CVS, you will need to issue the command `$ ./bootstrap` before the configure command.

All of the code in the MOAST repository is built using autotools. This allows for the ./configure to figure out your system’s individual configuration and build the code appropriately

4.4.2 Pyro

Please note that the pyro interface is out of date and no longer supported. If someone would like to renew support for this interface, please mail the USARSim developer’s mailing list.

4.4.2.1 Windows

Pyro is designed for Linux. Although Python, the development language used by Pyro, works under any OS system, Pyro uses some features only supported by Linux, such as Linux environment variables, shell commands. This makes Pyro only work on Linux. We have made Pyro work under Windows. The modified code can be found on pyro_win.zip. To install Pyro under windows:

1) Follow the Pyro Installation web page (http://pyrorobotics.org/pyro/?page=PyroInstallation) to install all the packages/software needed by Pyro. Please remember download and install the windows version.
2) After you restore Pyro, you need not run ‘make’ to compile it. Since it uses gcc and gmake to compile files, you will need these installed on your machine or the makefile will not work. Furthermore, it also tries to use XWindow, so give up compiling it under windows. Since this step only

affects the plugged third-part robots or simulators, it has no impact on USARSim. After you installed Pyro, you need to download and unzip pyro_win.zip (from the "Tools" section of the file release page on the USARSim repository) to overwrite the files in the Pyro directory. When the system asks you whether you want to overwrite the file(s) or not, please select 'yes'.

NOTE: When you restore the zipped file, please make sure it is restored under correct directory. If your directory structure looks something like …\Pyro\Pyro, you need to move all the files under …\Pyro\Pyro to …\Pyro.

4.4.2.2 Linux

1) Following the Pyro Installation web page (http://pyrorobotics.org/pyro/?page=PyroInstallation) to install Pyro.
2) Download the pyro_linux.tar from the "Tools" section of the USARSim files release page and restore it to the Pyro directory to install the USARSim plug-in.

4.4.3 Player

If you are new to player and not restricted by hardware player drivers that are only available for player-1.6, you should definitely use the latest player release (currently player-2.0.3). Player is primarily used on Linux system and other POSIX platforms. For Windows there is a work around to get Player 1.6.5 work with MinGW http://sourceforge.net/mailarchive/message.php?msg_id=15368188.

For player 1.6, please follow the installation document in player1.6.tar.gz to install it. For other versions of Player, do NOT use the "configure" file in player.tar.gz or player1.6.tar.gz. You need to generate it by using GNU Autotools. The installation steps are:

1) Copy the …/usarsim directory in the player.tar.gz into your Player's server/drivers/ directory. So in your Player, you will have a directory like server/drivers/usarsim.
2) Open the "deviceregistry.cc" in the server/ directory and add the following lines before the definition of 'player_interface_t interfaces[]':

```
#ifdef INCLUDE_USARSIM
void UsBot_Register(DriverTable *table);
void UsPosition_Register(DriverTable *table);
void UsPosition3d_Register(DriverTable *table);
void UsSonar_Register(DriverTable *table);
void UsLaser_Register(DriverTable *table);
void UsPtz_Register(DriverTable *table);
#endif
```

And then add the following lines into the function 'register_devices()':

```
#ifdef INCLUDE_USARSIM
  UsBot_Register(driverTable);
  UsPosition_Register(driverTable);
  UsPosition3d_Register(driverTable);
  UsSonar_Register(driverTable);
  UsLaser_Register(driverTable);
  UsPtz_Register(driverTable);
#endif
```

3) Go to the server/drivers/ directory, add ‘usarsim’ to ‘SUBDIRS’ in the file “Makefile.am”.
4) Under the Player directory, append the following line into ‘acinclude.m4’ right after other PLAYER_ADD_DRIVER sentences. PLAYER_ADD_DRIVER([usarsim],[drivers/usarsim],[yes],)
5) Add “#undef INCLUDE_USARSIM” to ‘config.h.in’ file.
6) In the ‘configure.in’ file, add the following line to AC_OUTPUT. server/drivers/usarsim/Makefile
7) Go back to the Player directory, use the following commands to generate the "configure" file:

   $ aclocal
   $ autoconf
   $ automake --add-missing

8) Follow the Player User Manual to compile and install Player.

To install Player 2.0.3 and our USARSim Player drivers:

1) Download Player 2.0.3 from: http://sourceforge.net/project/showfiles.php?group_id=42445
2) Restore Player on your machine.
3) Restore player.tar.gz located in the “Tools” section of the USARSim file release area to your Player directory. Please make sure all the files are put under the correct directory. Execute the autogen_usarsim.sh script. This script patches some player files. It needs autoreconf installed on your machine.
4) Follow the Player User Manual to compile and install Player. That is, execute the following commands:

   $ ./configure
   $ make
   $ make install

   If you want to install Player in another directory rather than /usr/local, you need to use the command:

$ ./configure –prefix <your directory>

5 Run the simulator

5.1 The steps to run the simulator

As of version 2.2, the preferred method of running USARSim is to run in “client only” mode. This may be accomplished by executing the following:

start *path_to_bin_dir/ut2004 map_name*?game=USARBot.USARDeathMatch?spectatoronly=1?TimeLimit=0?quickstart=true -ini=usarsim.ini

where *map_name* is the name of the map, for example DM-USAR_yellow (the yellow arena). Additional maps are available from the files section of the USARSim sourceforge repository under the “Maps” section.

TIP: The files start_usar.bat located in the UT2004\System directory can save you time in typing command line arguments.

Start the Controller

After the Unreal server is started, you can run your own application.

When you start USARSim in -client mode only- you can use these debug commands in the UT console:

- showlog
This command opens a window where the log file is shown in real time. This is like having the DOS server window. Actually it's better because in the DOS server window you cannot see client debug output, only server related output. In client mode, with showlog command, you can see all.

- showdebug
With this command UT will print some debug info directly on the client window. One interesting thing is the Location of the camera. With this information you can quickly choose new starting points (= location / 250).

- editactor class=<classname>
This is a very powerful command. You can for example go with the spectator camera near a robot and type: editactor class=KRobot and the property window of that actor will open. It's just like in UnrealEditor, but this time you can change any parameter in real time, in the simulation, including physics :-) Play with it.

There are a lot of other useful commands that work only in client mode, follow this link:
http://wiki.beyondunreal.com/wiki/Console_Commands

After the Unreal client is started, you can attach the viewpoint to any robot in the simulator. Go to the Unreal client, click left mouse button, you will get the image

viewed from the robot. To switch to next robot, click left mouse button again. To return back to the full viewpoint, click the right mouse button. When your viewpoint is attached to a robot, you can press key 'C' to get a viewpoint that looks over the robot. Pressing 'C' again will bring you back to the viewpoint of the robot.

TIP: *Left mouse button* attaches your viewpoint to a robot. *Right mouse button* returns your viewpoint to full viewpoint. *Pressing 'C'*, let you switch viewpoint between robot's viewpoint and the overlook viewpoint.

Besides the step by step manual run of USARSim, you can embed step 1 and 2 into your application. That is, let your application start the Unreal server and client for you, and then start itself. The examples in the following section we will show you how to run USARSim manually and automatically.

5.2 Examples

There are five controllers in the USARSim package. We explain them separately in the following sections.

5.2.1 The testing control interface

USAR_UI is a testing interface written in Visual c++ 6.0. It only works on Windows. You can use it to send any commands to the server, and it will display all the messages that come from the server to you. Follow steps 1 and 2 in section 5.1 to start the Unreal server and client, and then execute usar.exe. This will pop up a window. To use the interface:

1) Click "Connect" button to connect to the server.
2) Type the spawning robot command in the command combo box, then click "send" to send out the command. An example spawning command looks like: "INIT {ClassName USARBot.P2DX} {Location 4.5,1.9,1.8}", where 'ClassName USARBot.P2DX' is the robot type. The "P2DX" may be replaced by Zerg, Talon, P2AT, P2DX, ATRVJr, Hummer, etc... The 'Location' is the initial position of the robot. Each map comes with a text file that contains recommended start points. Sample startpoints for the USAR arenas are given in Table 2.
3) After adding the robot to the simulator, you can try to give control commands through the command combo box. The messages from the server are displayed on the bottom text box. To view a message, double click it.
4) You can also use a joystick or keyboard+mouse to control the robot. To do this, click the "Command" button in the "Mode" group. To return to the command mode, click the right button of the mouse.

   For joystick:

   If you have set joystick enabled in Unreal, you need to disable it so the system will not be confused. The ways of using joystick are:

   - Pushing the joystick forward/backward will move the robot forward/backward.

- Pushing the joystick to left/right side will turn the robot to left/right.
- Pushing POV button up/down will tilt the camera
- Pushing POV button left/right will pan the camera.

For keyboard+mouse:

Since the interface and Unreal share the keyboard and mouse, when you control the robot, you MUST let the interface be active. Otherwise, the interface cannot get the input from the keyboard and the mouse. To control the robot,

- Up/Down Arrow key moves the robot forward/backward.
- Left/right Arrow key turns the robot to left/right.
- Move mouse up/down to tilt the camera.
- Move mouse left/right to pan the camera.

5.2.2 MOAST

MOAST is configured for a particular simulation environment through the use of its initialization file. Control of which MOAST modules will be run is controlled through a run script.

5.2.2.1 Configuration

Before starting MOAST, the Unreal server must be running. The operation of MOAST is controlled through the moast.ini file located in the dev/etc directory under the MOAST home directory. For the novice user, there are only two entries of interest in this file. The first is the entry **HOST_NAME** located under the **[USARSIM_API]** section. This should be set to the host that is running the unreal server. When the system is started, it will communicate with the Unreal Server to determine which world is in play and will then read that world's parameters.

These parameters (the second interesting item) are located in the section **[WorldName]** (where worldName is the name of the world in play) and have the following meaning:

UNREAL_UTM_OFFSET: This provides an offset between the location reported by the simulation and the location reported by the robot over its navigation channel. It is used to georeference arenas to the real world and utilizes the Universal Transverse Mercator (UTM) coordinate system. The offset is provide as a triplet of northing, easting, down.

UTM_LETTER: MOAST utilizes the letter 'S' for the southern hemisphere and 'N' for the northern hemisphere.

UTM_ZONE: The UTM zone. The zone locations may be found at http://www.dmap.co.uk/utmworld.htm.

UTM_START_POSE_COUNT: The number of start poses that are included in this section.

UTM_START_POSE_x: The starting location of the robot in offset coordinates.

NOTE: If the world being used is fictitious, then the **UNREAL_UTM_OFFSET** should be set to 0, 0, 0 and the letter and zone may be set to any value.

5.2.2.2 Running MOAST

The run script for starting MOAST is located in the dev/bin directory under the MOAST home directory. The file is named run. This file controls which levels of the MOAST hierarchy are automatically started. For example, setting SECT, VEH, and AM to no and PRIM to yes will allow control at the level of sending individual wheel velocities. Setting VEH to yes will allow waypoints to be sent to the vehicle. Full documentation on the levels of control is given on the MOAST webpage. For this example we will examine joystick control and waypoint control.

For joystick control, set SECT, VEH, and AM to no and PRIM to yes. Then execute:

$ ./run

This will bring up a prim shell that allows for various commands to be sent to the robot and status to be received. Typing a carriage return will print the robot status and typing a question mark (?) will show the possible commands at this level.

Try typing *vel .1 0*. This will cause the robot to drive in a straight line. Another way to control at this level is to run the *joystick* program. To run this, open a new window in the *bin* directory and run *./joytick*. Move the mouse into the window that appears and then use the keys r and f to accelerate/decelerate the left wheel and the up/down arrows to accelerate/decelerate the right wheel.

For waypoint control, change the VEH to yes in the run file. When this file is executed, the PRIM, AM, and VEH levels will automatically be started. The prompt on the screen will be the vehicle shell. Once again, typing a carriage return will show vehicle status and a question mark (?) will show the possible commands at this level.

Try typing the following:

> init
> dump
> mvl 1 0

These commands perform the following functions:

init: This initializes the robot platform. It is necessary before any movement commands will be accepted.

dump: This turns on a display of the robot's internal world model at this level.

mvl: This tells the robot to move to a position in local coordinates.

More information on running MOAST may be found at the MOAST website (http://moast.sourceforge.net).

5.2.3 Pyro

The Pyro plug-in embeds the loading of the Unreal server/client. To start Pyro, go to the pyro/bin directory. If you are using Windows OS, execute the pyro.py. If you are on Linux, run the shell file pyro. After the Pyro interface is launched,

1) Click the 'Simulator' button and select USARSim.py on the plugins\simulators directory.
2) Select the arena you want to load on the plugins\worlds\USARSim.py directory. NOTE: here USARSim.py is a directory and not a file. Pyro will automatically load the Unreal server and client for you. Under linux OS, the Unreal client is launched in another console. Using Ctrl+Alt+F7 and Ctrl+Alt+F8, you can switch between the two consoles.[2]
3) Click the 'Robot' button and select the robot you want to add on the plugins\robots\USARBot directory. You will see that the robot is added in the virtual environment.
4) You should now be able to control the robot using the 'Robot' menu.
5) To view the sensor data or camera state, you can select the 'Service…' from the 'Load' menu to load a service. On the plugins\services\USARBot directory, select the sensor you want to view.
6) You can also try to load a 'Brain' to control the robot. Click the 'Brain' button and select a brain on the plugins\brains. For example, you can select Slider.py or Joystick.py to control the robot. You also can select BBWander.py to let the robot wander in the arena.

For details about Pyro, please read section 13.2.

Tips: To *switch among windows*, you can use Alt+Tab.

To *get control from UT2004*, press Esc.

To *pause the simulator*, switch to the Unreal client and then press Esc.

5.2.4 Player

Player is a device server. So before you use Player, you need to start USARSim. Please follow step 1 and 2 in section 5.1 to launch USARSim. As mentioned above, it's hard to switch focus between the Unreal Client and other applications under Linux, we recommend you launch USARSim on another machine.

After USARSim is started,

1) You need to prepare the configuration file used for Player. To learn how to prepare the configuration file, please read the Player User Manual and section 13.3.1. A sample configuration file usarsim.cfg is included in the player.tar.gz file. You can simply copy this file to some place and use it to test Player. Before going to the next step, you need to change host name in the file to the host that is runing USARSim.
2) Go to the place where you store the configuration file, execute the following command:

   $ player <config file>
3) Start your Player client. For example, you can run the Player visualization tool, plyerv.

[2] In Linux KDE, UT2004 may not support switching focus to other applications. As a solution, we launch UT2004 on another console to let user switch between UT2004 and other applications.

Note: Player uses *absolute camera control*. By default, all the robots in USARSim use relative camera control. You need to change the USARBot.ini file to let Player work well. For details of changing camera control mode, please read section 12.

5.2.5 SimpleUI

SimpleUI is an example user interface under Windows. It may be downloaded from the tools section of the USARSim file repository and should be placed in the UT2004\Tools directory. To successfully use it, please make sure the FreeImage.dll, Info.html and SimpleUI.exe are in the same directory. Also, the file hook.dll MUST be in ut2004.exe's directory. Besides directly using Unreal client as the video feedback, this interface demonstrates how to use the video pictures on the user interface. SimpleUI can obtain video pictures either through locally capturing the Unreal Client or receiving them from the image server. The details about getting and using video pictures are explained in section 10.13 and 14.5. In SimpleUI, we simply display the video pictures without any image processing. How to run SimpleUI in local or remote mode is introduced below.

5.2.5.1 Using SimpleUI by locally capturing pictures

The steps of running SimpleUI are :

1) Start Unreal Server (see step 1 in section 5.1).
2) Execute SimpleUI.exe which is located on ut2004/Tools/SimpleUI/Release directory.
3) On the SimpleUI interface, set the following parameters and then click the 'Start' button.
    a. 'Command' group
    This group specifies the server that receives the control command. 'Host' is the IP address of the Unreal Server. 'Port' is the port number of Gamebots. By default, it's 3000.
    b. 'Robot' group
    This group defines which robot, and where the robot will be added. 'Model' is the robot type.
    'Position' is the location to add the robot. Please note, in different arenas, different position parameters are needed. The robots and starting positions are given in a combo box. To add new ones, make graphical or text changes to SimpleUI.rc. Text editing this file is obscure, since the robot names are not presented as strings.
    c. 'Video' group
    Since we want to get pictures locally, we select the 'Local' radio button. 'Resolution' sets the picture size. 'Frame Rate' sets the maximum video frame rate. The actual frame rate is decided by the current computer system's capability.
4) The Unreal Client will be launched by SimpleUI and a message box will be popped up to instruct you how to switch to the Unreal Client to set the viewpoint and then to switch back to SimpleUI. After you set the viewpoint

and click the ‘OK’ button on the message box, the Unreal Client will be hidden and the control interface will appear.

NOTE: Only press the ‘OK’ button when you have set the viewpoint correctly. If you pressed the ‘OK’ button before you set the viewpoint, you still can use the ‘Show UT’ button to launch Unreal Client and set the viewpoint.

5) On the control interface, you can monitor the camera pictures, sensor data and control the wheels and camera of the robot. The usage of the control interface is:
    a. ‘Video’ group
    In the image frame is the picture from the camera. Under the frame, ‘FPS’ is the actual video frame rate in frames per second. ‘Width’ and ‘Height’ is the size of the picture. ‘Show UT’/’Hide UT’ button displays or hides the Unreal Client. When the Unreal Client is displayed, you can reset the viewpoint on Unreal Client.
    b. ‘Range Sensor Data’ group
    If your vehicle uses sonar, IR, range sensors or LIDAR these will be shown graphically with a color code in this box.
    c. ‘Sensor Data’ group
    The sensor data is displayed on a sensor tree. You can open or close a branch to show or hide the detailed sensor data.
    d. ‘Wheels’ group
    This group controls how the robot moves. The arrow buttons control the robot in the corresponding direction with a fixed speed. The ‘Faster’ and ‘Slower’ buttons speed up or slow down the robot’s moving speed. The ‘Stop’ button stops the robot. The buttons on the top: ‘Turn More’, ‘Straight’ and ‘Turn Less’ are only effective for Ackerman steered vehicles. Turning a skid drive vehicle with separately powered wheels can be done by clicking the left or right arrow. This will send opposite velocities to each wheel.
    e. ‘Camera’ group
    This group controls the robot’s camera. Up and down arrow buttons tilt the camera up and down 10 degrees. Left and right arrow buttons pan the camera to left and right side 10 degrees. The ‘Zoom In’ and ‘Zoom Out’ buttons zoom in and zoom out the camera separately.
    f. ‘Light’ group
    The ‘Lights On’/‘Lights Off’ button turns the headlights on or off. The ‘Trace On’/’Trace Off’ will enable or disable leaving a green trace of where the robot has been. The trace will persist until you restart the server.

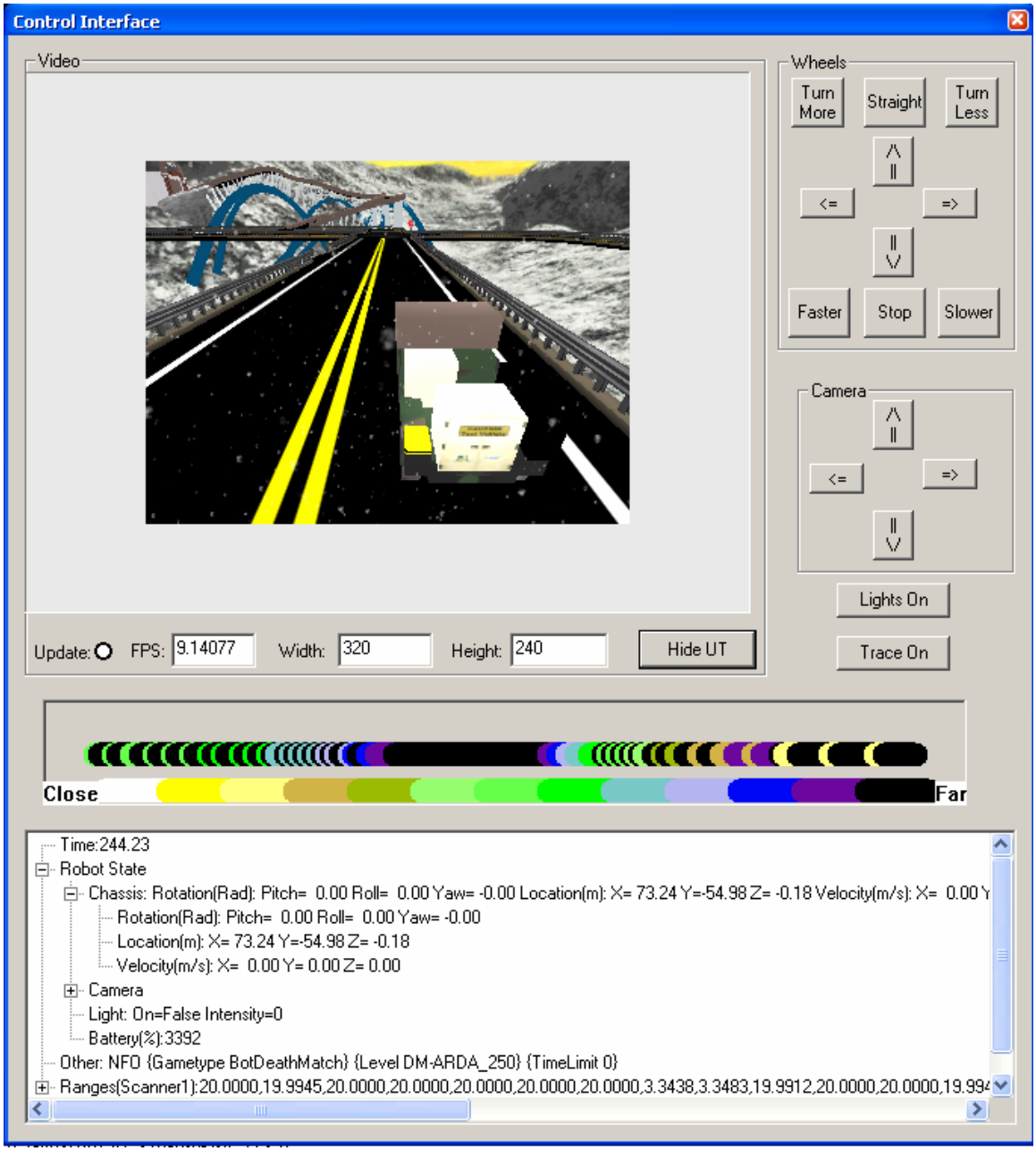


Figure 16: SimpleUI

5.2.5.2 Using SimpleUI by remotely receiving pictures

To run SimpleUI in remote mode, we need to start image server before we launch SimpleUI. Please see section 3.2.5 for information on starting the image server.

After the image server runs, we run SimpleUI in the following steps:

1) Execute SimpleUI.exe which is located on ut2004/Tools/SimpleUI/Release directory.

2) Similar to the step 3 in the last section, we set the ‘Command’, ‘Robot’ and ‘Video’ parameters on the interface. For the ‘Video’ group, because we want to run SimpleUI in remote mode, we need to select the ‘Remote’ radio button and set the ‘Host’ and ‘Port’ to the image server’s IP address and port number.
3) Click the ‘Start’ button to launch the control interface. On the interface, we will find the pictures are not the scenes viewed from the robot’s camera. This is because when we started the image server, we had no robot in the world. We couldn’t set the robot’s viewpoint at that time. So we need to go back to the image server to reset the viewpoint. On the ImageSrv interface, we click the ‘Show UT’ button to launch the Unreal Client. Go to the Unreal Client, we attach the viewpoint to the robot we just spawned in the world. Then we click the ‘Hide UT’ button on ImageSrv interface to hide the Unreal Client. When we switch to the SimpleUI interface, we will get the correct camera pictures.
4) Follow the usage introduced in the previous section to control the robot and its camera.

5.3 Getting Starting Poses From Maps

5.3.1 Introduction

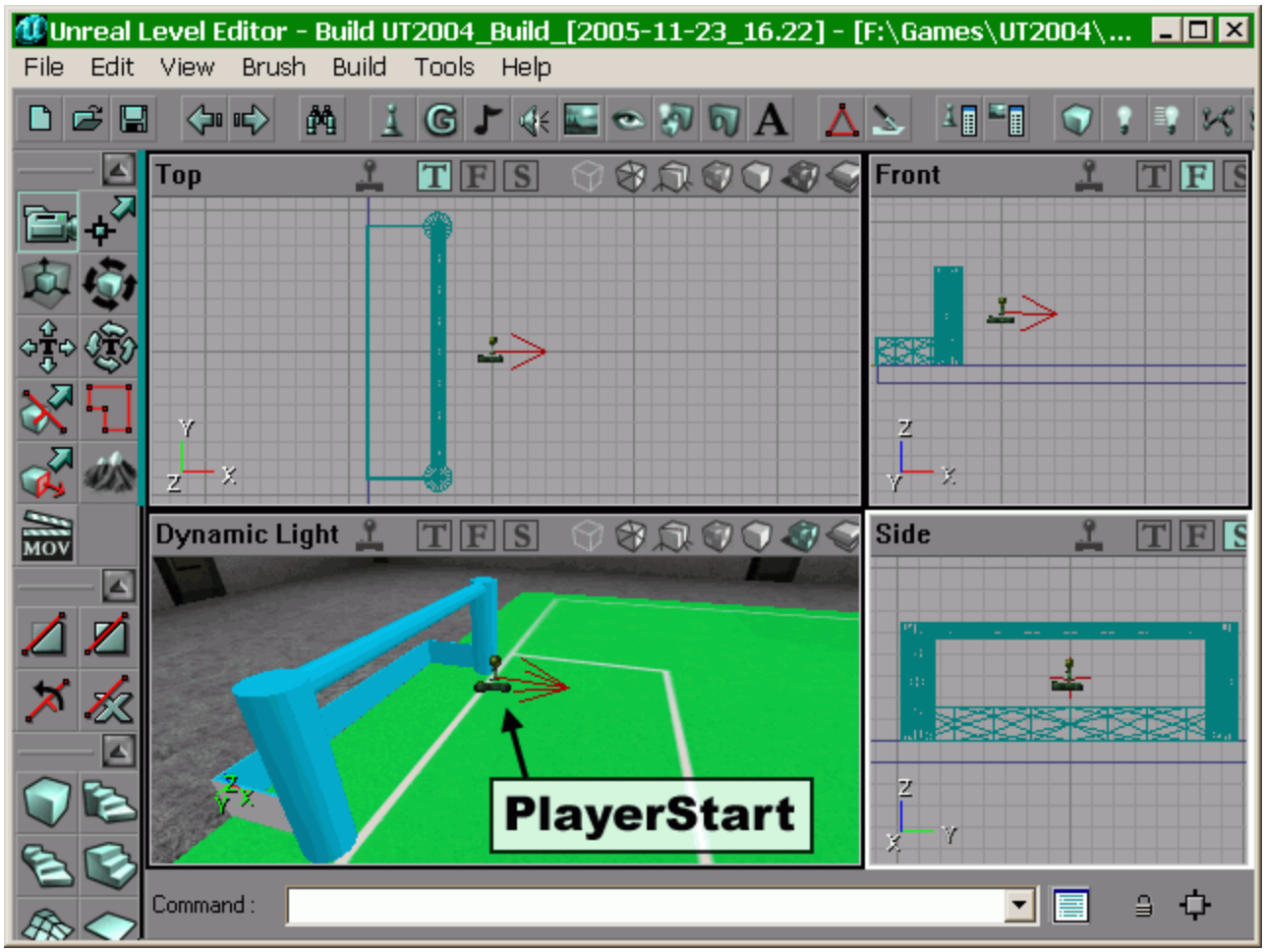
A common question when using USARSim is: "where I spawn the robots?". Usually you can find starting positions using the editor or directly in the UT client, with the **showdebug** console command. This approach isn't difficult, but cannot be automated. You have always to provide starting positions manually to your robot controller. If you change the map you will need to recompile the controller or to change some initialization file. That's why we introduced in our lab the following USARSim command:

GETSTARTPOSES

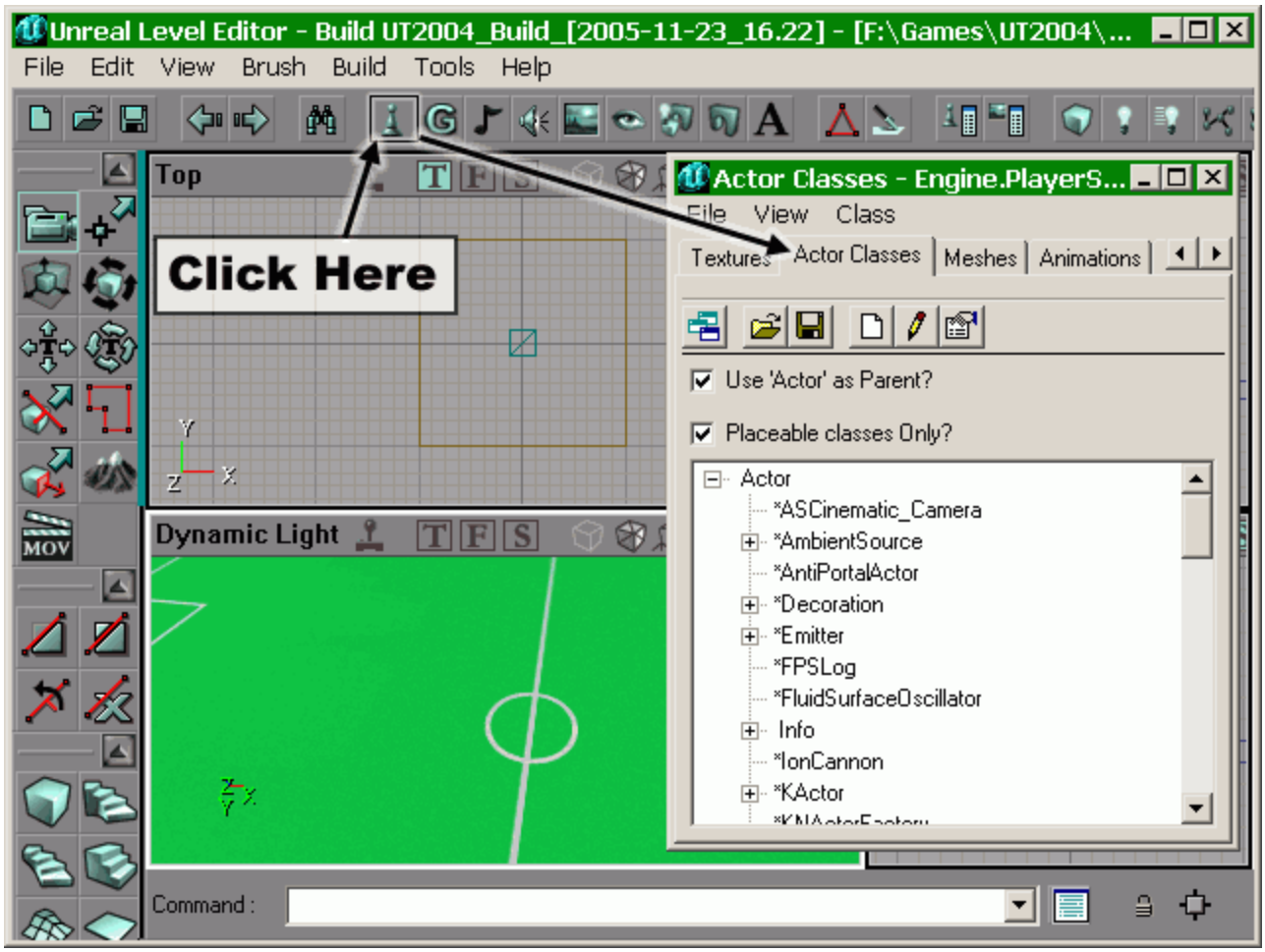
After connecting to USARSim you can send this command and you'll receive a string with all available starting positions for that map. This way starting positions are a map property and the controller can chose automatically where to start the robots.

5.3.2 Adding starting poses to the map

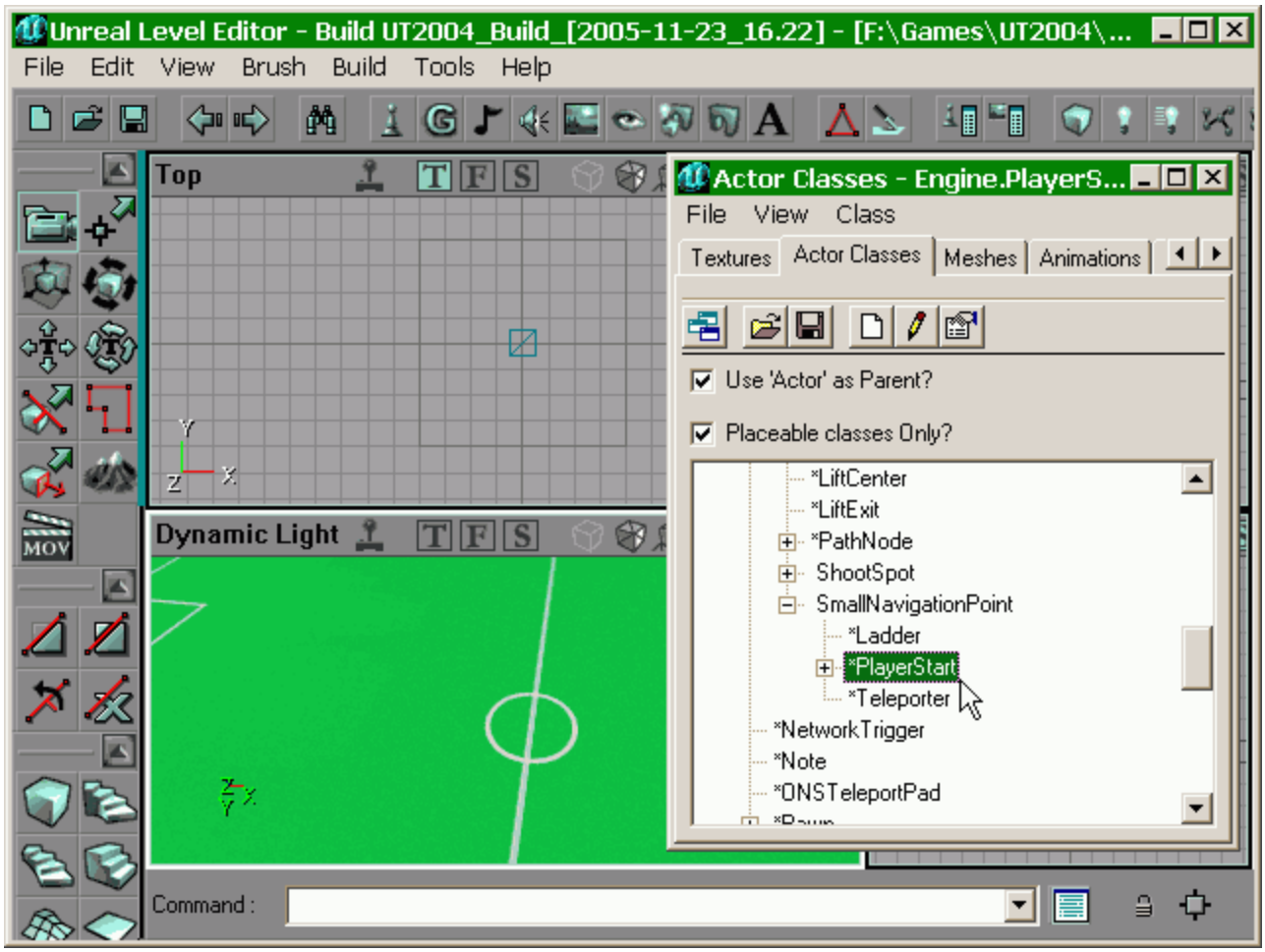
You have to populate the map with starting poses. Any NavigationPoint will do, however it's much easier to use PlayerStarts because you can see and adjust the orientation in the editor:



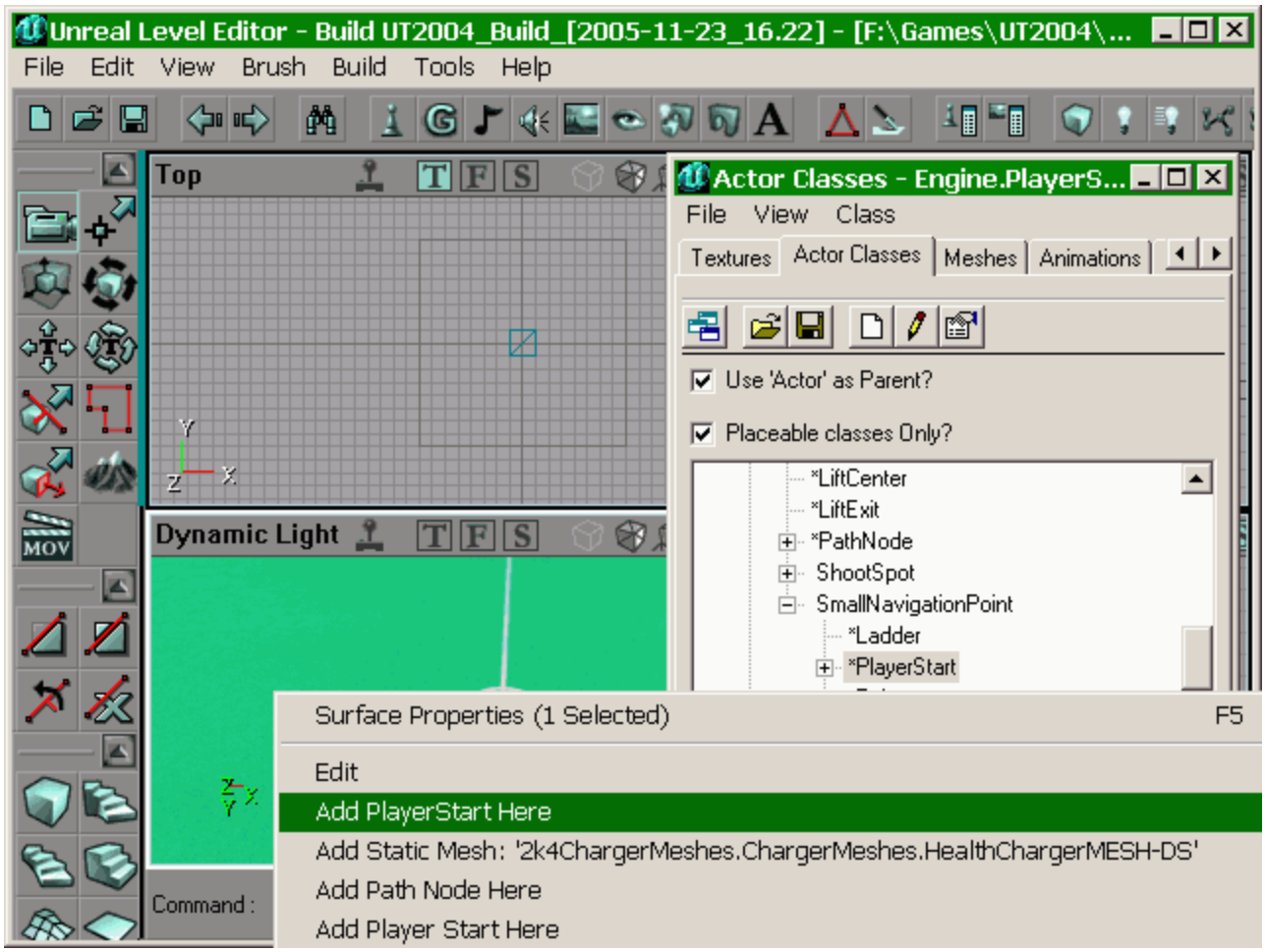
Any map must have at least 1 PlayerStart to be a valid map. You can add as many PlayerStarts as you want. Follow these steps to add one:



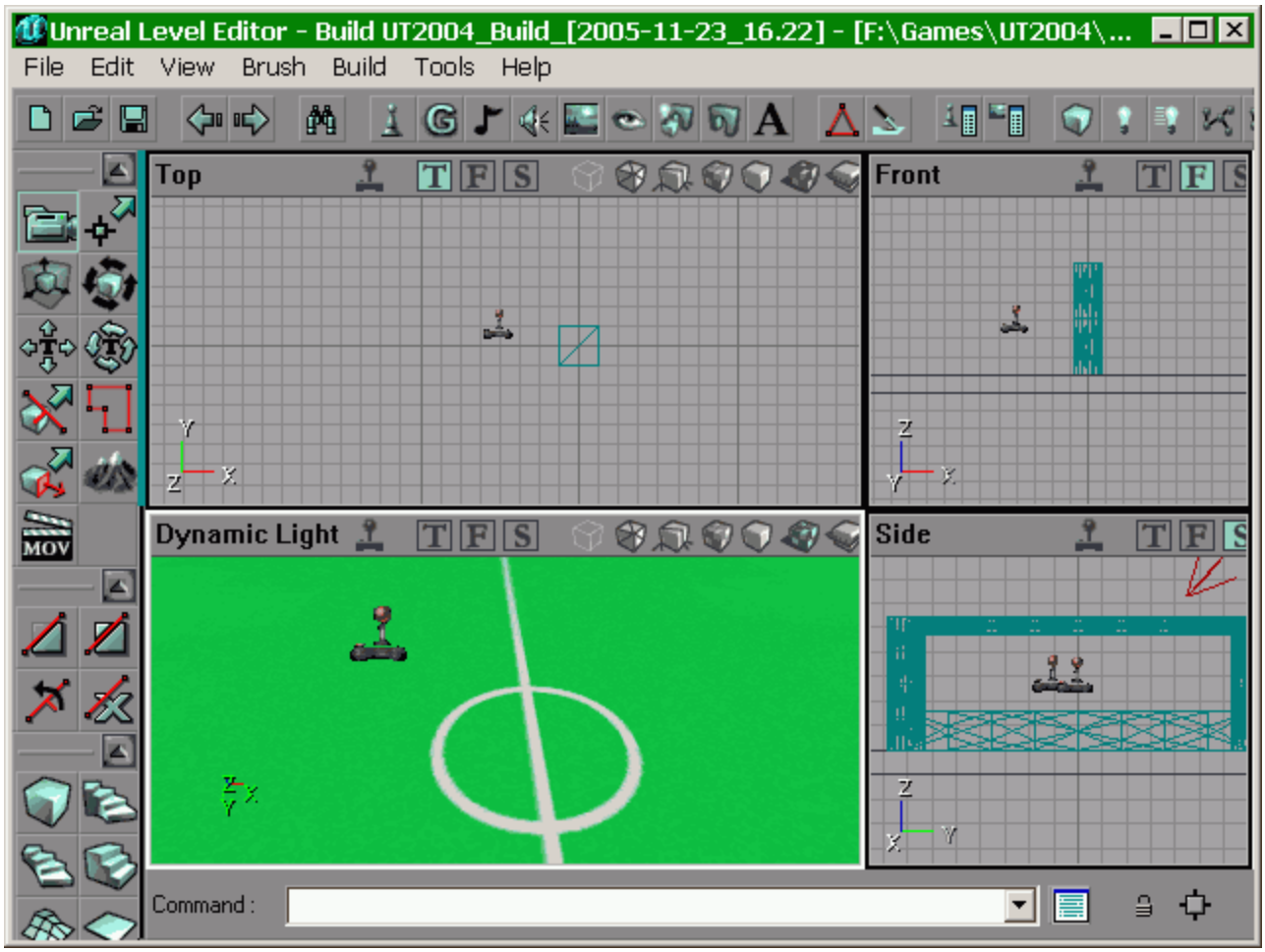
In the tree view find Actor/NavigationPoint/SmallNavigationPoint/PlayerStart and select it:



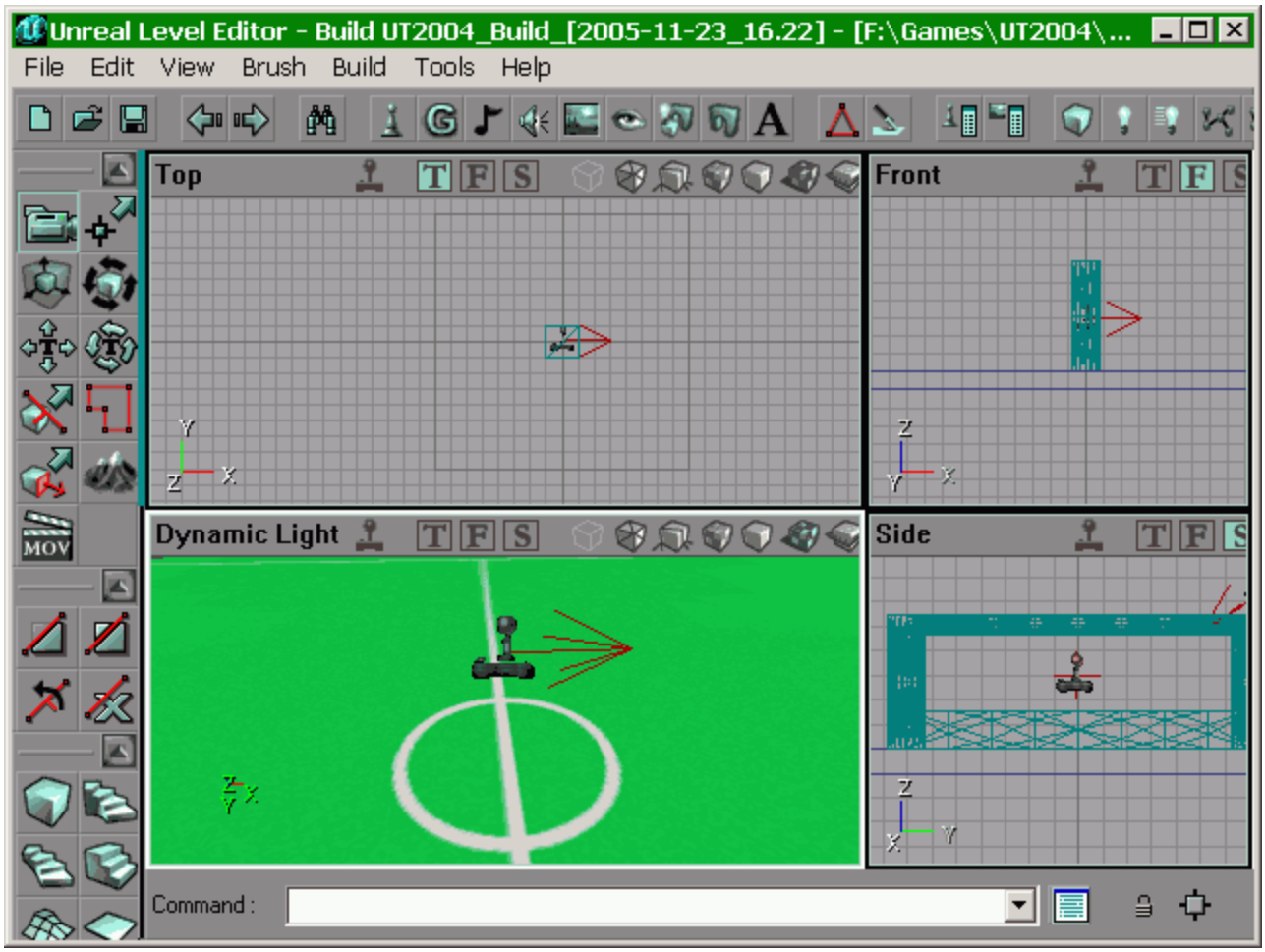
Now right click in the 3D map and chose "Add PlayerStart Here":



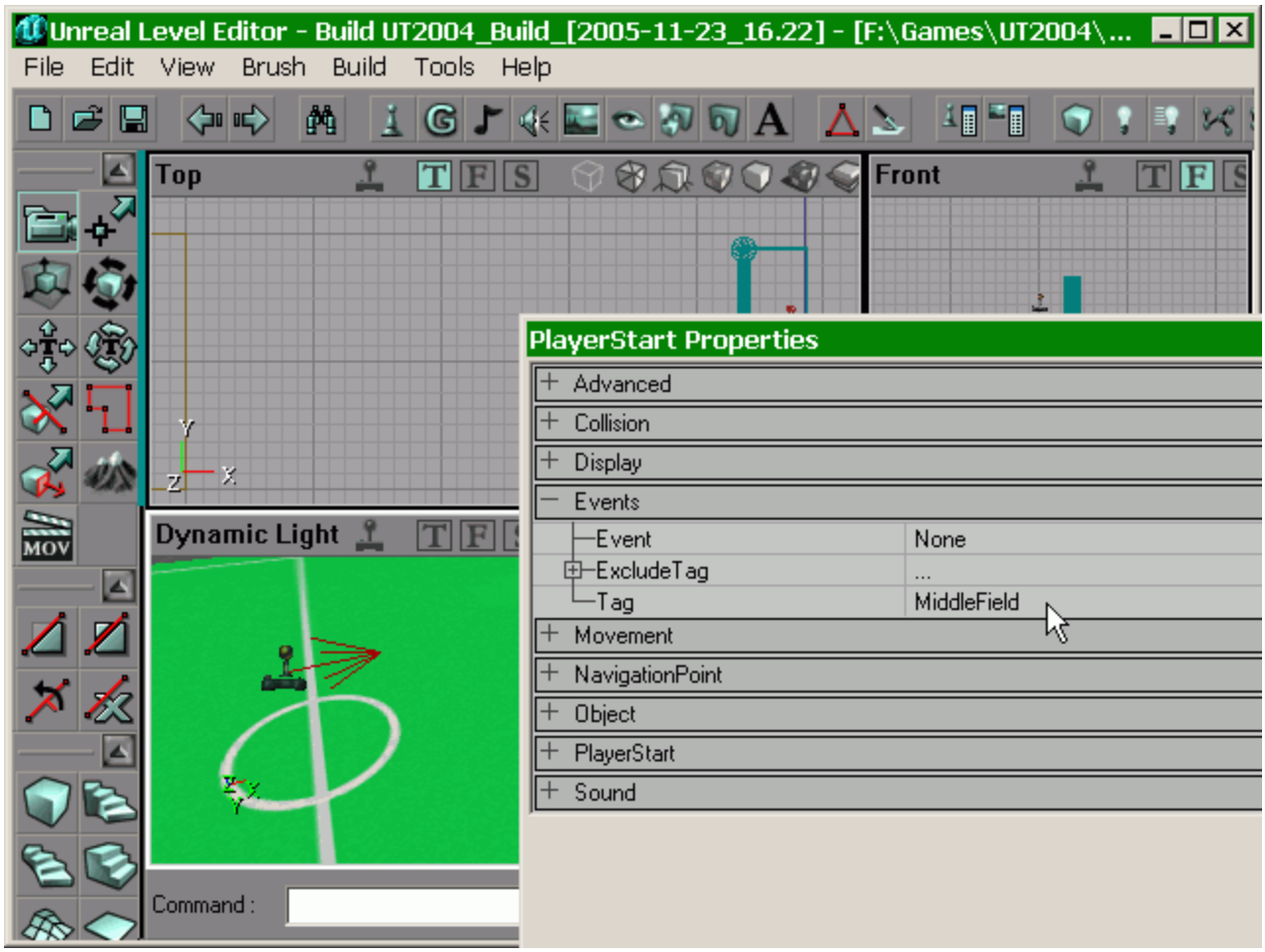
You have just placed a PlayerStart:



You can select it, move it and rotate it like any other object in the editor (ctrl + left click + drag to move, ctrl + right click + drag to rotate). Pay attention to place it not too high otherwise if you compile the map you'll receive a warning. I placed this PlayerStart in the middle field:



Every PlayerStart defines a starting position and orientation. But, when your controller connects to USARSim, how can you chose between different starting poses? You can assign an identifier to every PlayerStart using the Tag field (double click on the PlayerStart to open the properties window):



This way you can chose standard IDs for every starting pose that will allow you to chose the right one. Now save the map and close the editor. It's not necessary to compile the map because you don't need to create a weighted bot navigation graph.

5.3.3 Retrieving starting poses

Now, when you connect to USARSim, you can send this command before spawning any robot:

GETSTARTPOSES

You will receive a string like this one:

NFO {*StartPoses* 3} {*BlueGoal* -2.55,-0.00,-0.19 0.00,-0.03,-0.01 *MiddleField* 0.01,-0.00,-0.19 0.00,0.00,0.00 *YellowGoal* 2.57,-0.01,-0.19 0.00,0.00,3.13}

The syntax of this string doesn't follow strictly the USARSim standard. I decided for it only because it's simpler to parse. If you don't like it, it can be changed easily :-)

StartPoses is the number of starting positions available. For every starting position you'll receive it's tag, location and orientation:

BlueGoal -2.55,-0.00,-0.19 0.00,-0.03,-0.01

where *BlueGoal* is the tag, (-2.55,-0.00,-0.19) is the location and (0.00,-0.03,-0.01) is the orientation.

If there are no starting positions defined (that's bad because a map must have at least 1 PlayerStart), you'll receive:

NFO {*StartPoses* 0}

5.3.4 Start Elevation

As we want to improve spawning precision (we don't want robots to jump from the ground or fall from the sky when created), we need to have exact Z spawning values for the robots we are using. What we have to do is:

1. Find a way to place Player Starts in the map all at the same Z value.
2. Write a table that says, for every robot, how much to add or subtract from that Z value.

Why we need the Player Starts all at the same Z value? Because the map (the GETSTARTPOSES command) doesn't know what kind of robot we are going to spawn, therefore it can't adjust the Z value to match our needs. The best that it can do is to give us some standard Z value. We can then use that value to find the correct Z spawning value for our robot.

The GETSTARTPOSES command could also be modified to take some parameters, like the robot type for example, but I think it's not a good solution. Robot can change over time, can be added or deleted, we may want to use the same starting positions with different kind of robots etc... That's why, in my opinion, it's better to use some standard Z value.

It's also possible to modify the INIT command so that it accepts Player Starts tag names as locations. But right now I don't have time for that :-(I hope I will return to this solution in the future (or maybe someone else can work on it).

5.3.5 Standard Z starting value

Playing with the editor I discovered that all Player Starts are placed at a standard Z of 47 unreal units, that is 18.8cm. Also, if this value is modified by editing the Player Start position, it is reset to 47 after a map (re)build.

5.3.6 Z Table

Now that we know that the Z value returned by GETSTARTPOSES is always 18.8cm above ground (actually it may vary from 18 to 19 cm because the value is rounded to the nearest cm), it's easy to write the following table (other robots may be added if needed):

Robot	Position above ground	Delta Z
P2AT	-26 cm	-7cm
P2DX	-20 cm	-1 cm
Zerg	-6 cm	+13 cm
Talon	-13 cm	+6 cm
ATRVjr	-39 cm	-20 cm

- **Position above ground** is the Z spawning point for that robot assuming that the ground is at Z=0.
- **Delta Z** is the value to be added to the Z returned by GETSTARTPOSES command.

5.3.6.1 Example:

GETSTARTPOSES returns (these are the real values from the map you have seen in the flash movie):

NFO {*StartPoses* 4} {*Start1* -1.79,0.70,-0.19 0.00,0.00,0.00 *Start2* -4.10,0.19,-1.21 0.00,0.00,0.00 *Start3* -4.09,-1.79,-1.72 0.00,0.00,0.00 *Start4* -0.96,-1.86,-2.24 0.00,0.11,0.51}

If I want to spawn a **P2AT** at the Start4 position I will use as location:

-0.96,-1.86,-2.31

because Z = (-2.24 -0.07) m. If I want to spawn a **Zerg** at the same location I will use:

-0.96,-1.86,-2.11

because Z = (-2.24 + 0.13) m.

6 Coordinates, Units and Scale

In USARSim, there are two kinds of coordinates and units. One is the coordinate and unit system used in the Unreal Engine. Another is the coordinate and unit system used by the user applications interface. If you are programming in Unreal Engine, it’s your responsibility to convert from the application interface coordinate and unit system to the unreal engine coordinate and unit system. When you want to

send data back to the application interface, you must transform these coordinates and units back. In USARSim, all the conversions are implemented through the coordinate and unit converter class, USARConverter. To use your own coordinate and unit system in the application interface, you need to built your converter class and configure USARSim to use it rather than the default USARConverter.

Scale is the ratio of the real object size to the corresponding size in the virtual world. When you build a robot or world model, you must follow the scale. Otherwise, you will get incorrectly scaled data in the application interface. Of course, you can have your own scale. But you must change the converter class to make sure you get the correct data.

Tip: While creating your own coordinate system and scale is possible, it is not recommended. Instead you should use the standards outlined in section 6.2

6.1 Coordinates

Unreal Engine uses a left-hand coordinate system (Figure 17). The positive X-axis extends in front of you and the positive Y-axis is on your right hand. The positive Z-axis points straight up.

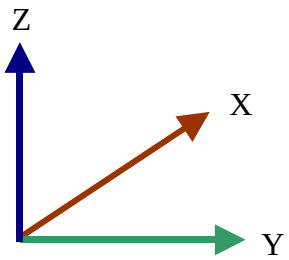


Figure 17: Left Hand Coordinate System

In the application interface, we use the right hand SAE J670 Vehicle Coordinate System (Figure 18). The only difference from the Unreal Engine coordinate system is that the positive Z-axis points straight down and not up.

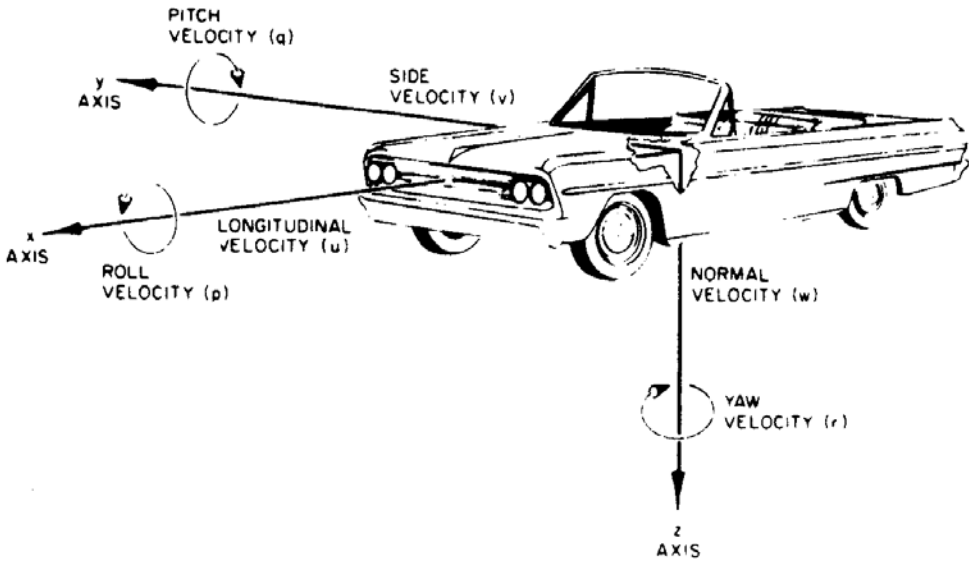


Figure 18: SAE J670 Vehicle Coordinate System

In Unreal, a rotator is represented by the following structure. Every element is an integer in Unreal Units.

```
struct Rotator
{
  var() config int Pitch, Yaw, Roll;
};
```

However, in the application interface we use a vector to describe a rotator. The X, Y and Z element represent the rotation angle around the corresponding axis. They are all floating point values expressed in radians.

6.2 Units and scale

The unit used in Unreal is called an UU (Unreal Unit). Unreal Engine uses it to represent both length and angle. The exceptions in Unreal Engine are: 1) it uses degrees instead of UU to count FOV (Field Of View); 2) in trigonometric functions, it uses radians. The unit conversion is summarized below:

250 UU = 1 m (Please use the function C_MeterToUU to convert from UU to meters)

32768 UU = 3.1415 radian = 180 degree = 0.5 circle

In the application interface we use SI units that are built upon the modern metric system. The base SI units along with the symbols used for abbreviations are listed in Table 1. By default, all the data sent out from USARSim is represented as a floating-point-number that has 4 digits after the decimal point.

Table 1: SI base units

Base quantity	SI base unit	
	Name	Symbol
length	Meter	m

mass	Kilogram	kg
time	Second	s
electric current	Ampere	A
thermodynamic temperature	Kelvin	K
amount of substance	Mole	mol
Luminous intensity	Candela	cd

NOTE: In your application, all the data you get from USARSim is in SI units and all the data you send to USARSim should also be in SI units.

7 Mission Package

The mission package concept is used throughout this manual. Therefore we introduce this concept before we explain any other detailed information about USARSim.

A mission package is a virtual parts and joints container used for organizational and control purposes. It is constructed of connected parts that work together to fulfill a behavior which is not related to the robot's mobility. For example, the camera's pan and tilt parts work together to adjust the camera's pose as shown in Figure 19. A part is connected to another by attaching a mount to a joint. Every part and mount has its own local coordinate frame. The joint control changes the relationship between the part's coordinate frame and the mount's coordinate frame. In general, a mission package is described as a part set, and every part has 0~N (N>0) mount locations. The parts connect to each other on the mounts through joints. Figure 20 is a more general example that depicts an arm mission package.

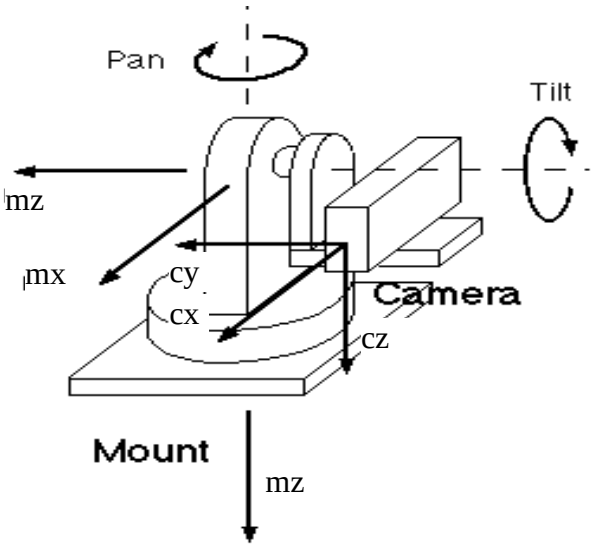


Figure 19: Mission package and its coordinates

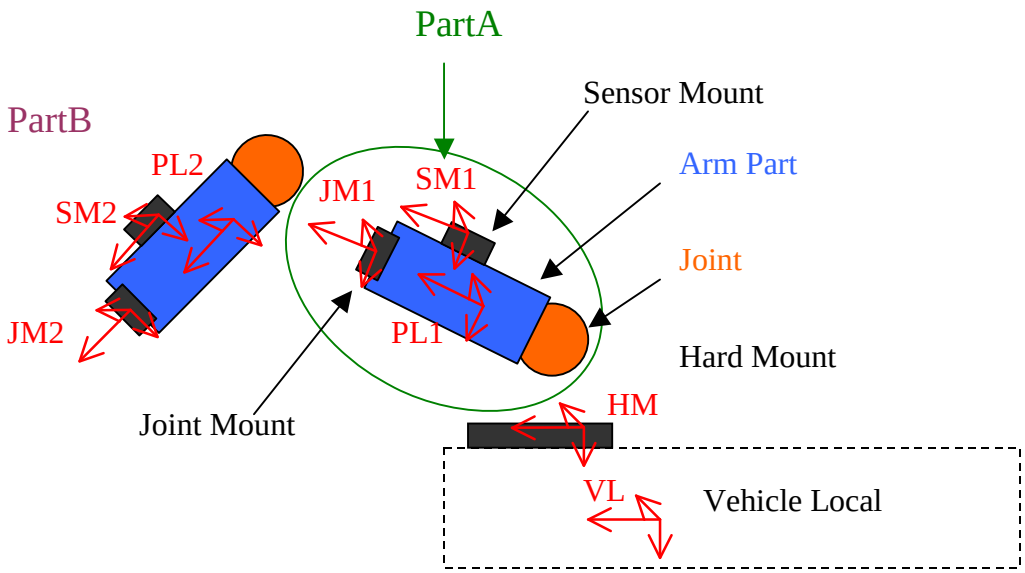


Figure 20: Arm mission package

When building a robot, we define a mission package as a list of part-joint pairs. Every pair defines a part and how it connects to its parent. The order of the pairs is always from the root (usually it's the robot platform) to leaf (terminal). In section 12, we will give detailed instructions to control the different parts of a mission package. A mission package's state is described as a series of the part's absolute angle from its initial mount position. We use the MISSTA message to deliver this information. This message is explained further in the next section.

8 Effecters

Effecters are defined in the USARSim as robotic subsystems that alter the state of the environment through direct interaction with objects and features found in the environment. These subsystems differ from the mobility subsystem and the mission package subsystem in that they usually don't require joint level control and are used to perform specific tasks like grab or release object in the environment. Because these devices are task-specific it is hard to formulate generalized control and status messages for these systems. In order to enable the possibility for autonomous control of these subsystems USARSim has imposed a restricted vocabulary that defines "opcodes" that are to be used by all effecters. This enables new effecters to develop as along as they honor the interface by implementing one or more of the opcodes.

Opcodes in USARSim are enumerated types that define different actions that an effecter can perform. The current opcodes that are defined in USARSim are:

Opcodes	**Description**
Activate	The activate opcode is used to turn on a process such as a wielder or a paint gun, or place the effecter in an active state such arming a disrupter.
Animate	The animate opcode is used for moving parts such as grippers or rollers

	on a roller table
Fire	The fire opcode implies a discrete action such as firing a nail from a nail gun or discharging a disrupter. It can also be used to initiate multiple firing through the use of the parameters
Release	The release opcode is used to drop an item into the environment or to detach a mount item.
Reset	The reset opcode is used for refreshing the effecter to its initial state or reloading the effecter.
NOP	The NOP, is a general no operation command that is send in order to return the effecter to an idle state.

9 Communication & Control (Messages and commands)

In this section, we introduce how to communicate between USARSim and your application. It will help you understand how to control the robots in the USARSim virtual environment.

9.1 TCP/IP socket

As was mentioned before, Gamebots is the bridge between Unreal and the controller. It opens a TCP/IP socket for communication purposes. The IP address of the TCP/IP socket is the IP address of the machine that runs the Unreal server. The default port number of the socket is 3000, and the maximum allowed number of connections number is 16. To change these parameters, we can go to the BotAPI.ini file in the Unreal system directory. The section [BotAPI.BotServer] of BotAPI.ini looks like:

```
[BotAPI.BotServer]
ListenPort=3000
MaxConnections=16
```

Where, 'ListenPort' is the port number of the socket. 'MaxConnections' is the maximum number of connections. It also decides the maximum number of robots you can add into the virtual world. You can change or add (if you cannot find the parameters in the INI file) the parameters to the values that you want.

9.2 The protocol

NOTE: A *Name* or *String* as referred to in this manual is defined as any consecutive collection of non-white-space printable ASCII characters (with no delimiters such as single or double quotes). This is ASCII characters 33 through 126. More information may be found at http://udn.epicgames.com/Two/UnrealScriptReference#Variables.

The communication protocol is the Gamebots protocol. All of the data (messages and commands) follow the format:

data_type {segment1} {segment2} …

where

data_type: specify the type of the data. It is upper case characters. Such as INIT, STA, SEN, DRIVE etc.

segment: is a list of name/value pairs. Name and value are separated by a space. For example, for "Location 4.5,1.9,1.8", the name is 'Location', the value is '4.5,1.9,1.8'. For the segment "Name Left Range 1.5", the names are 'Name' and 'Range', the values are 'Left' and '1.5'.

A message or command is constructed by a data_type and multiple segments. data_type and segments are separated by a space. Every message or command ends with "\r\n" that tells Gamebots the data is finished.

NOTE: Name/value pairs are separated by a space. Spaces **MUST NOT** be used elsewhere in the statement. For example one must use "Location 4.5,1.9,1.8" and not Location 4.5, 1.9, 1.8"

NOTE: When you send out a command, don't forget to append "\r\n" to tell Gamebots that the data is ended.

9.3 Messages

There are currently five types of message. A State message is the message class that reports the robot or mission package's state. Sensor messages contain the sensor data. Geometry messages report the robot, sensor, effecter, or mission package's geometry information. Configuration messages give the robot, sensor, effecter, or mission package's configuration information, and the response message provides a response status to certain command.

- State and Mission Package message

  Please note that the robot state message parameters depend on the type of robot that you are driving. For example, a robot of type "GroundVehicle" will not have the same state message as a robot of type "AerialVehicle." A robot state message looks like (for a ground vehicle):

  STA {Type *string*} {Time *float*} {FrontSteer *float*} {RearSteer *float*}{LightToggle *bool*} {LightIntensity *int*} {Battery *int*}

  Where:

  {Type *string*} 'string' describes the vehicle type. It will be one of the following values: "GroundVehicle", "LeggedRobot", "NauticVehicle", or "AerialVehicle".

{Time *float*}	'float' is the UT time in seconds. It starts from the time the UT server starts execution.
{FrontSteer *float*}	Note: parameter only available for robots of "GroundVehicle" type. Current front steer angle of the robot, in radians.
{RearSteer *float*}	Note: parameter only available for robots of "GroundVehicle" type. Current rear steer angle of the robot, in radians.
{SternPlaneAngle *float*}	Note: parameter only available for robots of "NauticVehicle" type. Current stern plane angle of the robot, in radians,
{RudderAngle *float*}	Note: parameter only available for robots of "NauticVehicle" type. Current rudder angle of the robot, in radians.
{LightToggle *bool*}	Indicate whether the headlight is turned on. 'bool' is true means on. False value means off
{LightIntensity *int*}	Light intensity of the headlight. Right now, it always is 100.
{Battery *int*}	'*int*' is the battery lifetime in second. It's the total time remaining for the robot to run.

Example: STA {Type GroundVehicle} {Time 62.01} {FrontSteer 0.0000} {RearSteer 0.0000} {LightToggle False} {LightIntensity 0} {Battery 3564}

A Mission Package state message reports the mission package links' current properties. Every mission package state includes a Name segment and several link segments. A mission package state message looks like:

MISSTA {Time *float*} {Name *string*} {Link *int*} {Value *float*} {Torque *float*} {Link *int*} {Value *float*} {Torque *float*} …

Where:

{Time *float*}	'float' is the UT time in seconds. It starts from the time the UT server starts execution.
{Name *string*}	'*String*' is the name of the mission package.
{Link *int*}	This parameter gives the link number that will be described by the next two parameters (Value and Torque). Please note that you will have as many as these parameters as you have links.
{Value *float*}	This parameter has two possible meanings. If the link being described is a prismatic joint, *float* gives the distance, in meters,

	from the original position of the link. If the link being described is a revolute joint,
{Torque *float*}	The Torque parameter gives the current torque of the link being described.

Example: MISSTA {Time 102.09} {Name CameraPanTilt} {Link 1} {Value 0.0000} {Torque -20.0000} {Link 2} {Value -0.1600} {Torque -20.0000}

- Sensor message

Every sensor message starts with "SEN". After it is an optional Time segment, {Time *float*}, that reports the current time in seconds in the virtual world. Whether the Time segment will appear or not is decided by the sensor's 'bWithTimeStamp' variable. For details information, please read section 10.

- Range Sensor

SEN {Type *string*} {Name *string* Range *float*} {Name *string* Range *float*} …

Where:

{Type *string*}	'*string*' is the sensor type. It can be either "Sonar" or "IR" which means it's a Sonar sensor or IR sensor.
{Name *string* Range *float*}	'*string*' is the sensor name, '*float*' is the range value in meters.

Example: SEN {Time 45.14} {Type Sonar} {Name F1 Range 4.4690} {Name F2 Range 1.9387} {Name F3 Range 1.9159} {Name F4 Range 1.6547} {Name F5 Range 0.8889} {Name F6 Range 0.7640} {Name F7 Range 1.1075} {Name F8 Range 2.0773}

- Laser Sensor

SEN {Type *string*} {Name *string*} {Resolution *float*} {FOV *float*} {Range *r1,r2,r3*…}

Where:

{Type *string*}	'*string*' is the sensor type. It can be "RangeScanner" or "IRScanner".
{Name *string*}	'*string*' is the sensor name.
{Resolution *float*}	'*float*' is the sensor's resolution in radians. With FOV, we can calculate the number of the data in the Range segment.
{FOV *float*}	'*float*' is the sensor's field of view in radians.
{Range *r1,r2,r3*…}	'*r1,r2,r3*…' is a series of range values in meters.

Example: SEN {Type RangeScanner} {Name Scanner1} {Resolution 0.0174} {FOV 3.1415} {Range 2.2109,2.2075,2.2124,2.2122,2.2156,2.2166,2.2207,2.2283,2.2334,2.2392,2.2421,2.2533,2.2609,2.2696,2.2806,2.2894,2.3011,2.3104,2.3243,2.3409,2.3519,2.3708,2.3834,2.4032,2.4229,2.4429,2.4628,2.4853,2.5051,2.5282,2.5538,2.5837,2.6126,2.6425,2.6696,2.7012,1.3823,1.3642,1.3321,1.3025,1.2733,1.2457,1.2208,1.1951,1.1729,1.1522,1.1305,1.1109,1.0922,1.0741,1.0568,1.0412,1.0266,1.0110,0.9976,0.9845,0.9714,0.9592,0.9474,0.9377,0.9269,0.9170,0.9074,0.9001,0.8906,0.8835,0.8757,0.8680,0.8622,0.8556,0.8483,0.8440,0.8376,0.8331,0.8288,0.8248,0.8206,0.8070,0.8155,0.9298,1.0322,1.1549,1.3122,1.5795,1.7881,1.7952,1.7916,1.7913,1.7909,1.7891,1.7879,1.7878,1.7880,1.7898,1.7923,1.7960,1.8005,1.8023,1.8063,1.8131,1.8189,1.8236,1.8328,1.8395,1.8468,1.8541,1.8658,1.8740,1.8874,1.8990,1.9084,1.9234,1.9373,1.9507,1.9686,1.9843,1.9991,2.0188,2.0387,2.0568,2.0787,2.1020,2.1261,2.1519,2.1782,2.2026,2.2309,2.2636,2.1613,2.2169,2.2754,2.3380,2.4081,2.4626,2.5146,2.5325,2.5659,2.5955,2.6276,2.6645,1.5562,1.5928,1.6312,1.6698,1.7121,1.7554,1.8054,1.8507,1.9076,1.9653,2.0274,2.0935,2.1908,2.2722,2.3553,2.3809,4.5275,4.7421,4.9283,4.7977,4.6330,4.4815,4.3422,4.2114,4.0851,3.9718,3.8664,4.7144,4.6967,4.6828,4.6715,4.6571,4.6439,4.6354,4.6221,4.6157,4.6093,4.6163,4.6132,4.6037,4.5944}

- Odometry sensor

  SEN {Type Odometry} {Name *string*} {Pose *x,y,theta*}

  Where:

{Name *string*}	'*string*' is the sensor name.
{Pose *x,y,theta*}	'*x,y*' is the estimated robot position relative to the start point in meters. '*theta*' is the head angle in radians relative to the start orientation.

  Example: SEN {Type Odometry} {Name Odometry} {Pose 0.2415,0.0029,-0.5157}

- GPS Sensor

  SEN {Type GPS} {Name *string*} {Latitude *int,float,char*} {Longitude *int,float,char*} {Fix *int*} {Satellites *int*}

  Where:

  {Name *string*} '*string*' is the sensor name, as given in

the USARBot.ini robot's definition.

{Latitude *int,float,char*} '*int*', '*float*', '*char*' provide the latitude degree, minute (as a decimal), and cardinal description (i.e. 'N' or 'S'), respectively. There are only two possible values for the '*char*' parameter: 'N' for North and 'S' for South.

{Longitude *int,float,char*} '*int*', '*float*', '*char*' provide the longitude degree, minute (as a decimal), and cardinal description (i.e. 'E' or 'W'), respectively. There are only two possible values for the '*char*' parameter: 'E' for East and 'W' for West.

{Fix *int*} '*int*' indicates whether or not a position was acquired. The fix is the same as the GGA format. Namely, a value of 0 means that the GPS sensor failed to acquire a position and a value of 1 means that a position was acquired.

{Stallites *int*} '*int*' gives the number of satellites tracked by the GPS sensor. This number is an inexplicit source of accuracy. The more satellites are tracked, the higher the position accuracy.

Example: SEN {Type GPS} {Name GPS1} {Latitude 47,40.3323,N} {Longitude 122,18.5977,W} {Fix 1} {Satellites 8}

- INS Sensor

The Inertial Navigation Sensor is a sensor that provides estimates of the vehicles current location and orientation, based on measurements of angular velocity and linear acceleration relative to the vehicles current pose.

SEN {Type INS} {Name *string*} {Location *x,y,z*} {Orientation *r,p,y*} …

Where:

{Name *string*} '*string*' is the sensor name.

{Location *x,y,z*} *'x,y,z'*, are float variables for estimated vehicle locations.

{Orientation *r,p,y*} *'r,p,y'*, are float variables for estimated vehicle orientation in radians. All radians are in the range [0,2PI].

Example: SEN {Type INS} {Name INS} {Location 1.23, 2.23, 0.12} {Orientation 4.57, 3.14, 1.507}

Example: SEN {Type Encoder} {Name ECLeft Tick -61} {Name ECRight Tick -282} {Name ECTilt Tick 0} {Name ECPan Tick 0}

- Encoder sensor

  SEN {Type Encoder} {Name *string* Tick *int*} {Name *string* Tick *int*} …

  Where:

  {Name *string* Tick *int*} '*string*' is the sensor name. '*int*' is the tick count.

  Example: SEN {Type Encoder} {Name ECLeft Tick -61} {Name ECRight Tick -282} {Name ECTilt Tick 0} {Name ECPan Tick 0}

- Touch sensor

  SEN {Type Touch} {Name *string* Touch *bool*} {Name *string* Touch *bool*} …

  Where:

  {Name *string* Touch *bool*} '*string*' is the sensor name. '*bool*' indicates whether the sensor is touching something. Value 'True' means the sensor is touching something.

  Example: SEN {Type Touch} {Name Front Touch True} {Name Left Touch False} {Name Right Touch False}

- RFID sensor

RFID tags are simulated by implementing the class USARBot.RFIDTag class in the Unreal Editor when editing a map. This class contains an integer id and a boolean bSingleshot variable that determines whether the tag is a single shot tag or a multi shot tag. They are deployed by placing them in UnrealEd. If the tag's id is set to -1 (the default), then the tag id will be set to a unique value automatically. Other values for the id will not be changed. If the tags are within the MaxRange

of the RFID sensor mounted on the robot then the server sends the following message to the client:

SEN {Type RFIDTag} {Name *string*} {ID *int*}{Location *float, float, float* } …

Where:

{Name *string*}	'*string*' is the sensor name.

- Victim Sensor

SEN {Type VictSensor} {Name *string*} {PartName *string*} {Location *x,y,z*} {PartName *string*} {Location *x,y,z*} …

Where:

{Name *string*}	'*string*' is the sensor name.
{PartName *string*}	'*string*' is the name of the victim part that was discovered by the sensor. It can be one of 7 values: "Head", "Arm", "Hand", "Chest", "Pelvis", "Leg", and "Foot". Please note that the sensor does not differentiate between real victim's part and false alarms. It is up to the controller to perform this task.
{Location *x,y,z*}	Relavite location, based on the sensor's position and rotation, of the victim part where x, *y*, and *z* are in meters.

Example: SEN {Type VictSensor} {Name VictimSensor} {PartName Leg} {Location 2.05,0.33,0.46} {PartName Leg} {Location 2.51,0.44,0.39}

- Human Motion Detection

SEN {Type HumanMotion} {Name *string*} {Prob *float*}

Where:

{Name *string*}	'*string*' is the sensor name.
{Prob *float*}	'*float*' is the probability of it's human motion.

Example: SEN {Type HumanMotion} {Name Motion} {Prob 0.81}

- Sound Sensor

SEN {Type Sound} {Name *string*} {Loudness *float*} {Duration *float*}

Where:

{Name *string*}	'*string*' is the sensor name.

{Loudness *float*}	'*float*' is the loudness of the sound.
{Duration *float*}	'*float*' is the duration of the sound.

Example: SEN {Type Sound} {Name Sound} {Loudness 17.22} {Duration 6.63}

- Geometry Information

  For a ground vehicle, the geometry information message looks like:

  GEO {Type GroundVehicle} {Name *string*} {Dimensions *x,y,z*} {COG *x,y,z*} {WheelRadius *float*} {WheelSeparation *float*} {WheelBase *float*}

  Where:

{Type GroundVehicle}	The Type parameter is hard coded and will always be "GroundVehicle" for a ground vehicle GEO message.
{Name *string*}	'*string*' is the robot's name.
{Dimensions *x,y,z*}	'*x*' defines the robot's length, '*y*' defines the robot's width, and '*z*' describes the robot's height. Please note that these values are in meters.
{COG *x,y,z*}	'*x*', '*y*', and '*z*' identify the position of the center of gravity, in meters, calculated from the chassis origin.
{WheelRadius *float*}	'*float*' is the radius of the robot's wheels, in meters.
{WheelSeparation *float*}	'*float*' is the wheel separation, in meters. The wheel separation defines the distance between two wheels along the length (x axis) of the robot's chassis.
{WheelBase *float*}	'*float*' is the wheel base, in meters. The wheel base defines the distance between two wheels along the width (y axis) of the robot's chassis.

  Example: GEO {Type GroundVehicle} {Name ATRVJr} {Dimensions 0.7744,0.6318,0.5754} {COG 0.0000,0.0000,-0.1000} {WheelRadius 0.1922} {WheelSeparation 0.5120} {Wheelbase 0.3880}

For a legged robot, the geometry information message looks like (still to be expanded):

GEO {Type LeggedRobot} {Name *string*} {Dimensions *x,y,z*} {COG *x,y,z*}

Where:

{Type LeggedRobot}	The Type parameter is hard coded and will always be "LeggedRobot" for a legged

robot GEO message.

{Name *string*} '*string*' is the robot's name.

{Dimensions *x,y,z*} '*x*' defines the robot's length, '*y*' defines the robot's width, and '*z*' describes the robot's height. Please note that these values are in meters.

{COG *x,y,z*} '*x*', '*y*', and '*z*' identify the position of the center of gravity, in meters, calculated from the chassis origin.

Example: GEO {Type LeggedRobot} {Name ERS} {Dimensions 0.3190,0.1800,0.2780} {COG 0.0000,0.0000,0.0000}

For a nautic vehicle, the geometry information message looks like (still to be expanded):

GEO {Type NauticVehicle} {Name *string*} {Dimensions *x,y,z*} {COG *x,y,z*}

Where:

{Type NauticVehicle} The Type parameter is hard coded and will always be "NauticVehicle" for a nautic vehicle GEO message.

{Name *string*} '*string*' is the robot's name.

{Dimensions *x,y,z*} '*x*' defines the robot's length, '*y*' defines the robot's width, and '*z*' describes the robot's height. Please note that these values are in meters.

{COG *x,y,z*} '*x*', '*y*', and '*z*' identify the position of the center of gravity, in meters, calculated from the chassis origin.

Example: GEO {Type NauticVehicle} {Name Submarine} {Dimensions 6.5364, 1.1428, 1.9929} {COG 0.0000,0.0000,0.0000}

For an aerial vehicle, the geometry information message looks like (still to be expanded):

GEO {Type AerialVehicle} {Name *string*} {Dimensions *x,y,z*} {COG *x,y,z*}

Where:

{Type AerialVehicle} The Type parameter is hard coded and will always be "AerialVehicle" for an aerial vehicle GEO message.

{Name *string*} '*string*' is the robot's name.

{Dimensions *x,y,z*}	'*x*' defines the robot's length, '*y*' defines the robot's width, and '*z*' describes the robot's height. Please note that these values are in meters.
{COG *x,y,z*}	'*x*', '*y*', and '*z*' identify the position of the center of gravity, in meters, calculated from the chassis origin.

Example: GEO {Type AerialVehicle} {Name AirRobot} {Dimensions 9.0082, 6.6897, 2.6128} {COG 0.0000,0.0000,0.0000}

For sensors or effecters, the geometry message looks like:

GEO {Type *string*} {Name *string* Location *x,y,z* Orientation *x,y,z* Mount *string*} {Name *string* Location *x,y,z* Orientation *x,y,z* Mount *string*} …

Where:

{Type *string*}	'*string*' is the sensor/effecter's type name.
{Name *string* Location *x,y,z* Orientation *x,y,z* Mount *string*}	Specifies how an item is mounted on the robot. The '*string*' after 'Name' is the item's name, and the '*string*' after 'Mount' is the item's mounting base which is also another item. The '*x,y,z*' after 'Location' and 'Orientation' are the item's relative location and orientation to the mounting base. If more than one sensor/effecter is the same type, multiple '{Name *string* Location …}' segments will appear in the GEO message.

Example: GEO {Type Camera} {Name Camera Location 0.0820,0.0002,0.0613 Orientation 0.0000,-0.0000,0.0000 Mount CameraTilt}

For a mission package, the geometry message is expressed in the following format to tell us how the mission package and its elements are 'installed' together to the robot.

GEO {Type MisPkg} {Name *string*} {Link *int*} {ParentLink *int*} {Location *x,y,z*} {Orientation *x,y,z*} {Link *int*} {ParentLink *int*} {Location *x,y,z*} {Orientation *x,y,z*} …

Where:

{Type MisPkg}	The Type parameter is hard coded and will always be "MisPkg" for a mission package GEO message.
{Name *string*}	'*string*' is the mission package's name.
{Link *int*}	'*int*' is the link number of the described link.

{ParentLink *int*}	'*int*' is the parent link number of the described link. If the described link is mounted directly on the robot's chassis, the parent link number will be -1.
{Location *x*,*y*,*z*}	*x*, *y*, and *z* gives the x, y, and z coordinate, in meters, of the described link relative to its parent's position and based on its parent's orientation.
{Orientation *x*,*y*,*z*}	*x*, *y*, and *z* gives the x, y, and z orientation, in radians, of the described link relative to its parent's orientation. Please note that the coordinate system is rotated using a ZYX order.

Example: GEO {Type MisPkg} {Name CameraPanTilt} {Link 1} {ParentLink -1} {Location 0.1239,0.0000,-0.2036} {Orientation 3.1415,0.0000,0.0000} {Link 2} {ParentLink 1} {Location 0.0000,0.0000,0.0599} {Orientation 1.5707,0.0000,0.0000}

- Configuration Information

  For a ground vehicle, the configuration message looks like:

  CONF {Type GroundVehicle} {Name *string*} {SteeringType *string*} {Mass *float*} {MaxSpeed *float*} {MaxTorque *float*} {MaxFrontSteer *float*} {MaxRearSteer *float*}

  Where:

{Type GroundVehicle}	The Type parameter is hard coded and will always be "GroundVehicle" for a ground vehicle CONF message.
{Name *string*}	'*string*' is the robot's name.
{SteeringType *string*}	'*string*' is one of the following: "AckermanSteered" or "SkidSteered" or "OmniDrive", as dictated by the steering type of the robot.
{Mass *float*}	'*float*' is a the robot's mass, in kilograms.
{MaxSpeed *float*}	'*float*' is the maximum spin speed of the robot's wheels, in radians per second.
{MaxTorque *float*}	'*float*' is the maximum torque of the robot, in unreal units.
{MaxFrontSteer *float*}	'*float*' is the maximum steering angle for robot's front wheels, in radians. Please note that this value will be 0 for skid steered vehicles.
{MaxRearSteer *float*}	'*float*' is the maximum steering angle for robot's rear wheels, in radians. Please note that this value will be 0 for skid steered vehicles.

Example: CONF {Type GroundVehicle} {Name ATRVJr} {SteeringType SkidSteered} {Mass 50.0000} {MaxSpeed 2.0943} {MaxTorque 60.0000} {MaxFrontSteer 0.0000} {MaxRearSteer 0.0000}

For a legged robot, the configuration message looks like (still to be expanded):

CONF {Type LeggedRobot} {Name *string*} {SteeringType *string*} {Mass *float*}

Where:

{Type LeggedRobot}	The Type parameter is hard coded and will always be "LeggedRobot" for a leged robot CONF message.
{Name *string*}	'*string*' is the robot's name.
{SteeringType *string*}	'*string*' gives the number of legs of the legged robot. As of now this value can only be "TwoLegs" or "FourLegs".
{Mass *float*}	'*float*' is a the robot's mass, in kilograms.

Example: CONF {Type LeggedRobot} {Name QRIO} {SteeringType TwoLegs} {Mass 50.0000}

For a nautic vehicle, the configuration message looks like (still to be expanded):

CONF {Type NauticVehicle} {Name *string*} {SteeringType *string*} {Mass *float*}

Where:

{Type NauticVehicle}	The Type parameter is hard coded and will always be "NauticVehicle" for a nautic vehicle CONF message.
{Name *string*}	'*string*' is the robot's name.
{SteeringType *string*}	'*string*' gives the steering type of the nautic vehicle. As of now this value can only be "VariableDepth".
{Mass *float*}	'*float*' is a the robot's mass, in kilograms.

Example: CONF {Type NauticVehicle} {Name Submarine} {SteeringType Underwater}{Mass 50.0000}

For an aerial vehicle, the configuration message looks like (still to be expanded):

CONF {Type AerialVehicle} {Name *string*} {SteeringType *string*} {Mass *float*}

Where:

{Type AerialVehicle}	The Type parameter is hard coded and will

	always be “AerialVehicle” for an aerial vehicle CONF message.
{Name *string*}	‘*string*’ is the robot’s name.
{SteeringType *string*}	‘*string*’ gives the steering type of the aerial vehicle. As of now this value can only be “RotaryWing”.
{Mass *float*}	‘*float*’ is a the robot’s mass, in kilograms.

Example: CONF {Type AerialVehicle} {Name AirRobot} {SteeringType RotaryWing} {Mass 50.0000}

For a sensor or effecter, a configuration message looks like:

CONF {Type *string*} {*Name Value*} {*Name Value*} …

Where:

‘{Type *string*}’ specifies the sensor type. ‘*string*’ is the type name.

‘{*Name Value*}’ is the name value *pair* that describes the feature of this sensor type. Different sensor types have different name value pairs. For detailed information, please refer to section 10 about how to configure the sensor.

Example: CONF {Type Camera} {CameraDefFov 0.8727} {CameraMinFov 0.3491} {CameraMaxFov 2.0943} {CameraFov 0.8726}

For an effector, the configuration message looks like thisL

CONF {Type Effecter} {Name *Value*} {Opcode *OpcodeName*} {MaxVal *Value*} {MinVal *Value*} …

Where:

‘{Type Effecter}’ specifies that this message is a configuration message for effecters

‘{Name *Value*}’ is the name value *pair* that indicates the unqiue name of the effecter.

‘{Opcode *OpcodeName*}’ is the name of the opcode that the effecter implements. Refer to Section 8 for a list of the opcodes.

‘{MaxVal *Value*}’ is the name value *pair* that indicates the upper bound for the value that is accepted by the effecter.

‘{MinVal *Value*}’ is the name value *pair* that indicates the lower bound for the value that is accepted by the effecter.

NOTE: An effecter can implement more than one opcode and the CONF message returns the configuration information for all effecters on the robotic platform.

Therefore the segment with the name value pair indicates a new effecter and the opcode name value pair indicates the opcode the effecter specified in by name implements.

Example: CONF {Type Effecter} {Name Roller} {Opcode Animate} {MaxVal 6}{MinVal -6}

For a mission package, the configuration information describes each link's properties. The message looks like:

CONF {Type MisPkg} {Name *string*} {Link *int*} {JointType *string*} {MaxSpeed *float*} {MaxTorque *float*} {MinRange *float*} {MaxRange *float*} {Link *int*} {JointType *string*} {MaxSpeed *float*} {MaxTorque *float*} {MinRange *float*} {MaxRange *float*} …

Where:

{Type MisPkg}	The Type parameter is hard coded and will always be "misPkg" for a mission package CONF message.
{Name *string*}	'*string*' is a mission package's name.
{Link *int*}	'*int*' is the mission package's link number that will be described.
{JointType *string*}	'*string*' can either be "Revolute" or "Prismatic", as determined by the type of joint being described.
{MaxSpeed *float*}	'*float*' describes the joint's maximum speed, in rad/s.
{MaxTorque *float*}	'*float*' describes the joint's maximum torque.
{MinRange *float*}	For a revolute joint, '*float*' is the minimum absolute angle that the joint can rotate to. For a prismatic joint, '*float*' is the minimum distance that the joint can move to.
{MaxRange *float*}	For a revolute joint, '*float*' is the maximum absolute angle that the joint can rotate to. For a prismatic joint, '*float*' is the maximum distance that the joint can move to.

NOTE: For revolute joints, if the MinRange parameter is greater than the MaxRange parameter, the joint does not have any constraints.

Example: CONF {Type MisPkg} {Name CameraPantilt} {Link 1} {JointType Revolute} {MaxSpeed 0.17} {MaxTorque 20.00} {MinRange 1} {MaxRange 0} {Link 2} {JointType Revolute} {MaxSpeed 0.17} {MaxTorque 20} {MinRange -0.16} {MaxRange 0.40}

- Response Message

  A response message is delivered to describe the status of a command that has been sent to USARSim. Response messages are used so that users can tell whether or not particular commands were successfully executed. There are three different response messages. The first response message is issued after a SET {Type Viewports} command. The second response messaged is issued after a SET {Type Camera} command. The third response message is a generic message used for sensors and effecters.

  After a SET {Type Viewports} command, the following response message will be issued:

  RES {Time *float*} {Type Viewports} {Config *string*} {Status *string*} {Viewport1 *string*} {Status *string*} {Viewport2 *string*} {Status *string*} {Viewport3 *string*} {Status *string*} {Viewport4 *string*} {Status *string*}

  Where:

{Time *float*}	'*float*' is the timestamp in the virtual world when the message is sent out. It is represented in seconds.
{Type Viewports}	"Viewport" is hard coded into this parameter.
{Config *string*}	'*string*' describes the current viewport configuration. This parameter will be either "SingleView" or "QuadView".
{Status *string*}	'*string*' is the status of the viewport configuration after the SET command has been issued. The status will be "OK" when the viewport configuration was successfully changed. Otherwise, the status will be "Failed".
{Viewport1 *string*}	'*string*' is the name of the camera currently attached to viewport1. If viewport1 has been disabled, this parameter will be "Disabled". If viewport1 has been attached to a non-existent camera, this parameter will be "None".
{Status *string*}	'*string*' is the status of viewport1 after the SET command has been issued. The status will be "OK" when the camera attached to viewport1 was successfully changed. Otherwise, the status will be "Failed".
{Viewport2 *string*}	'*string*' is the name of the camera currently attached to viewport2. If viewport2 has been disabled, this parameter will be "Disabled". If viewport2 has been attached to a non-existent camera, this parameter will be

	"None".
{Status *string*}	'*string*' is the status of viewport2 after the SET command has been issued. The status will be "OK" when the camera attached to viewport2 was successfully changed. Otherwise, the status will be "Failed".
{Viewport3 *string*}	'*string*' is the name of the camera currently attached to viewport3. If viewport3 has been disabled, this parameter will be "Disabled". If viewport3 has been attached to a non-existent camera, this parameter will be "None".
{Status *string*}	'*string*' is the status of viewport3 after the SET command has been issued. The status will be "OK" when the camera attached to viewport3 was successfully changed. Otherwise, the status will be "Failed".
{Viewport4 *string*}	'*string*' is the name of the camera currently attached to viewport4. If viewport4 has been disabled, this parameter will be "Disabled". If viewport4 has been attached to a non-existent camera, this parameter will be "None".
{Status *string*}	'*string*' is the status of viewport4 after the SET command has been issued. The status will be "OK" when the camera attached to viewport4 was successfully changed. Otherwise, the status will be "Failed".

Example: RES {Time 56.68} {Type Viewports} {Config SingleView} {Status OK} {Viewport1 Camera} {Status OK} {Viewport2 Disabled} {Status OK} {Viewport3 Disabled} {Status OK} {Viewport4 Disabled} {Status OK}

After a SET {Type Camera} command, the following response message will be issued:

RES {Time *float*} {Type Camera} {Name *string*} {FOV *float*} {Status *string*} …

Where:

{Time *float*}	'*float*' is the timestamp in the virtual world when the message is sent out. It is represented in seconds.
{Type Camera}	"Camera" is hard coded into this parameter.
{Name *string*}	'*string*' is the name of the camera that will be described by the next two parameters (i.e. FOV and Status).

{FOV *float*}	'*float*' is the current field of view of the camera being described, in radians. The current field of view is the field of view after a SET {Type Camera} has been issued.
{Status *string*}	'*string*' is the status for the camera's field of view after the SET command has been issued. The status will be "OK" when the camera's field of view was successfully changed. Otherwise, the status will be "Failed".

Example: RES {Time 426.76} {Type Camera} {Name Camera} {FOV 0.7853} {Status OK} {Name Camera2} {FOV 0.7853} {Status OK}

The generic response message, issued for other sensors and effecters, looks like:

RES {Time *float*} {Type *string*} {Name *string*} {Status *string*}

Where:

{Time *float*}	'*float*' is the timestamp in the virtual world when the message is sent out. It is represented in seconds.
{Type *string*}	'*string*' is the sensor or effecter's type.
{Name *string*}	'*string*' is the sensor or effecter's name.
{Status *string*}	'*string*' is the status after the sensor or effecter execute a command. Usually, it's "OK" means the command is successfully executed or "Failed" means the execution is failed. For camera's zoom in/out command, the status is the camera's current FOV in radians. The detailed information for every sensor and effecter is listed in sections 10 and 0.

Example: RES {Time 61.20} {Type Odometry} {Name Odo1} {Status OK}

9.4 Commands

In USARSim all the values in the commands are case insensitive. However, the data_type and names are case sensitive and the format must be exactly followed. The supported commands are:

- Add a robot to UT world:

  INIT {ClassName *robot_class*} {Name *robot_name*} {Location *x,y,z*} {Rotation *r,p,y*}

  Where:

  {ClassName *robot_class*} '*robot_class*' is the class name of the

	robot. It can be USARBot.ATRVJr, USARBot.Zerg, USARBot.P2AT, USARBot.P2DX, USARBot.Hummer, and any other robots built by the user.
{Name *robot_name*}	'robot_name' is the robot's name. It can be any string you want. If you omit this block, USARSim will give the robot a name.
{Location *x,y,z*}	'*x,y,z'* is the start position of the robot in meters from the world origin. For different arenas, we need different positions. The recommended positions are listed on Table 2 for the USAR arenas. Recommended start locations for worlds are given in a text file that is included with the world download. Worlds are available in the "maps" file release area on sourceforge.
{Rotation *r,p,y*}	'*r,p,y*' is the starting roll, pitch, and yaw of the robot in radians with North being 0 yaw.

Table 2 Recommended start position for the arenas

Arena	**Recommended Start Position**
Yellow	4.5,1.9,1.8
Orange	1.2,-2.3,1.16
Red	0.76,2.3,1.8

Example: INIT {ClassName USARBot.P2DX} {Location 4.5,1.9,1.8} {Name R1} will add a pioneer P2DX robot.

- Control the Robot:

There are seven kinds of control command. The first kind controls the left and right side wheels of a skid steered robot. The second kind controls the front and rear wheels of an Ackerman steered robot. The third kind controls an underwater robot. The fourth kind controls an aerial vehicle. The fifth kind controls the wheels of an OmniDrive robot. The sixth kind controls a specified joint of the robot. The seventh kind controls the angle of multiple joints of a robot, which is convenient for flipper, leg, and arm control.

- DRIVE {Left *float*} {Right *float*} {Normalized *bool*} {Light *bool*} {Flip *bool*}

  Where:

{Left *float*}	'*float*' is spin speed for the left side wheels. If we are using normalized values, the value range is –100 to 100 and corresponds to the robot's minimum and maximum spin speed. If we use absolute values, the value will be the real spin speed in radians per second.
{Right *float*}	Same as above except the values affect the right side

wheels.

{Normalized *bool*} Indicates whether we are using normalized values or not. The default value is 'False' which means absolute values are used to control wheel spin speed.

{Light *bool*} '*bool*' is whether turn on or turn off the headlight. The possible values are True/False.

{Flip *bool*} If a robot rolls over or otherwise tips off of its wheels, this will 'right' the robot. If '*bool*' is True, this command will flip the robot to its 'wheels down' position.

Example: DRIVE {Left 1.0} {Right 1.0} will drive the robot moving forward with spin seed 1 radian per second.
DRIVE {Left -1.0} {Right 1.0} will turn the robot to left side.
DRIVE {Light true} will turn on the headlight.
DRIVE {Flip true} will flip the robot

- DRIVE {Speed *float*} {FrontSteer *float*} {RearSteer *float*} {Normalized *bool*} {Light *bool*} {Flip *bool*}

  Where:

{Speed *float*} '*float*' is the spin speed for the wheels that are powered. If we use normalized values, the value range is –100 to 100 and corresponds to the robot's minimum and maximum spin speed, respectively. Otherwise, the value is the absolute spin speed, in radians per second.

{FrontSteer *float*} '*float*' specifies the steer angle of the robot's front wheels. If we use normalized values, the value range is -100 to 100 and corresponds to the robot's minimum and maximum steer angle, respectively. Otherwise, the value is the absolute steer angle, in radians.

{RearSteer *float*} '*float*' specifies the steer angle of the robot's rear wheels. If we use normalized values, the value range is -100 to 100 and corresponds to the robot's minimum and maximum steer angle, respectively. Otherwise, the value is the absolute steer angle, in radians.

{Normalized *bool*} Indicates whether we are using normalized values or not. The default value is 'False' which means absolute values are used.

{Light *bool*} '*bool*' is whether turn on or turn off the headlight. The possible values are True/False.

{Flip *bool*} If a robot rolls over or otherwise tips off of its wheels, this will 'right' the robot. If '*bool*' is True, this command will flip the robot to its 'wheels down' position.

Example: DRIVE {Speed -1.0} will drive the robot backward with a spin speed of 1 rad/sec.
DRIVE {Speed 1.0} {FrontSteer 0.523599} will drive the robot 30^{o} forward and to the left with a spin speed of 1 rad/sec.
DRIVE {Speed 1.0} {FrontSteer -0.523599} will drive the robot 30^{o} forward and to the right with a spin speed of 1 rad/sec.

- DRIVE {Propeller *float*} {Rudder *float*} {SternPlane *float*} {Normalized *bool*} {Light *bool*}

Where:

{Propeller *float*}	'*float*' is the spin speed for the propellers. If we use normalized values, the value range is –100 to 100 and corresponds to the propeller's minimum and maximum spin speed, respectively. Otherwise, the value is the absolute propeller's spin speed, in radians per second.
{Rudder *float*}	'*float*' specifies the angle of the robot's rudders. If we use normalized values, the value range is -100 to 100 and corresponds to the rudders' minimum and maximum steer angle, respectively. Otherwise, the value is the absolute rudder angle, in radians.
{SternPlane *float*}	'*float*' specifies the angle of the robot's stern planes. If we use normalized values, the value range is -100 to 100 and corresponds to the stern planes' minimum and maximum angle, respectively. Otherwise, the value is the absolute stern plane angle, in radians.
{Normalized *bool*}	Indicates whether we are using normalized values or not. The default value is 'False' which means absolute values are used.
{Light *bool*}	'*bool*' is whether turn on or turn off the headlight. The possible values are True/False.

Example: DRIVE {Propeller 1.0} will drive the robot forward with a propeller's spin speed of 1 rad/sec.
DRIVE {Propeller 1.0} {Rudder 0.523599} will drive the robot 30^{o} forward and to the right with a spin speed of 1 rad/sec.
DRIVE {Speed 1.0} {FrontSteer -0.523599} will drive the robot 30^{o} forward and to the left with a spin speed of 1 rad/sec.
DRIVE {Light true} will turn on the robot's headlights.

- DRIVE {AltitudeVelocity *float*} {LinearVelocity *float*} {LateralVelocity *float*} {RotationalVelocity *float*} {Normalized *bool*}

Where:

{AltitudeVelocity *float*}	'*float*' is the altitude velocity (i.e up/down). If we use normalized values, the value range is -100 to 100 and corresponds to the robot's minimum and maximum altitude velocity, respectively. Otherwise, the value is the absolute altitude velocity, in meters per second.
{LinearVelocity *float*}	'*float*' is the linear velocity (i.e forward/backward). If we use normalized values, the value range is -100 to 100 and corresponds to the robot's minimum and maximum linear velocity, respectively. Otherwise, the value is the absolute linear velocity, in meters per second.
{LateralVelocity *float*}	'*float*' is the lateral velocity (i.e left/right). If we use normalized values, the value range is -100 to 100 and corresponds to the robot's minimum and maximum lateral velocity, respectively. Otherwise, the value is the absolute lateral velocity, in meters per second.
{RotationalVelocity *float*}	'*float*' is the rotational velocity. If we use normalized values, the value range is -100 to 100 and corresponds to the robot's minimum and maximum rotational velocity, respectively. Otherwise, the value is the absolute rotational velocity, in radians per second.
{Normalized *bool*}	Indicates whether we are using normalized values or not. The default value is 'False' which means absolute values are used.

Example: DRIVE {AltitudeVelocity 1} will elevate the robot at a rate of 1 meters per second.
DRIVE {RotationalVelocity 0.1} will rotate the robot at a rate of 0.1 radians per second.
DRIVE {LinearVelocity -3} will make the robot go backward at a rate of 3 meters per second.

- DRIVE {WheelNumber *int*} {WheelSpeed float} {WheelSteer float}{WheelNumber *int*} {WheelSpeed float} {WheelSteer float} …

  Where:

{WheelNumber *int*}	'*float*' is the number of an OmniDrive robot's wheel as defined in the USARBot.ini.
{WheelSpeed *float*}	'*float*' is the spin speed, in rad/s, of the chosen wheel.
{WheelSteer *float*}	'*float*' is the steer angle ,in rad, of the chosen wheel

Example: DRIVE {WheelNumber 0} {WheelSpeed 3.14} {WheelSteer 0.75} {WheelNumber 1} {WheelSpeed -3.14} {WheelSteer 0.75}

This command will turn both wheels by 0.75 rad while giving them different velocities. If you use the LISA robot setup, the robot rotates on place.

- DRIVE {Name *string*} {Steer *int*} {Order *int*} {Value *float*}

  Where:

  {Name *string*} '*string*' is the joint name.
  {Steer *int*} '*int*' is the steer angle of the joint.
  {Order *int*} '*int*' is the control mode. It can be 0-2.
  0: zero-order control. It controls rotation angle.
  1: first-order control. It controls spin speed.
  2: second-order control. It controls torque.
  {Value *float*} '*float*' is the control value. For zero-order control, it's the rotation angle in radians. For first-order control, it's the spin speed in radians/second. For second-order control, it's the torque.

  Example: DRIVE {Name LeftFWheel} {Steer 1.57} will steer the left front wheel 90 degrees.
  DRIVE {Name LeftFWheel} {Order 1} {Value 0.175} will make the left front wheel spin at 0.175 radians/second, i.e. 10 degrees/second

- MULTIDRIVE {*string float*} {*string float*}…

  Where:

  {*string float*} The '*string*'-'*float*' pair describes a joint to move and a position to move to. '*string*' is the name of a joint, as described in the USARBot.ini file. '*float*' is the absolute angle, in radians, that the joint should move to. Please note that any number of '*string*'-'*float*' pairs can be sent using a single MULTIDRIVE command to move multiple joints at the same time.

  Example: MULTIDRIVE {FRFlipper -1} {FLFLipper -1} will move the two front flippers to -1 radians,
  MULTIDRIVE {RRFlipper 1} {RLFLipper 1} will move the two rear flippers to 1 radians.

- Control a joint:

  This command is used to drive a joint. The command looks like:

SET {Type Joint}{Name *string*}{Opcode *string*} {Params *p1*,*p2*}

Where:

{Name *string*}	'*string*' is the joint's name.
{Opcode *string*}	'*string*' is the operation code. The available codes are: '*Angle*' or '*0*': set the extra angle in radian the joint will move. '*Velocity*' or '*1*': set the spin speed in radian per second for the joint. '*Torque*' or '*2*': set the torque applied on the joint.
{Params *p1*,*p2*}	'*p1*' is the value corresponds to the Opcode. '*p2*' only uses for the KCarWheelJoint to set the steer angle.

Example 1: SET {Type Joint} {Name UpperArm} {Opcode Angle} {Params 1.25}

Example 2: SET {Type Gripper} {Name Gripper} {Opcode Open} {Params 0.8}

- Control the viewports:

When USARSim initially starts up, users can freely move around the world using the mouse and keyboard. Pressing the left mouse button of the mouse attaches the view to the robot's viewport controller. The robot viewport controller currently supports two configurations: SingleView and QuadView. The SingleView configuration provides users with a single view coming from a single camera. The QuadView configuration provides user with four views, giving them the possibility of viewing up to four cameras simultaneously (the screen is divided up into four equal portions, each of which is used for a camera). When a robot is added to a world, the viewport configuration is automatically set to accommodate for the maximum number of cameras. In other words, if a robot has zero or one camera, it will automatically start in SingleView; otherwise, it will automatically start in QuadView. Since a robot might have more than four cameras and users might want to cycle through all the cameras using SingleView, USARSim gives the possibility of configuring the viewports as follows:

SET {Type Viewports} {Config *int*/*string*} {Viewport1 *string*} {Viewport2 *string*} {Viewport3 *string*} {Viewport4 *string*}

Where:

{Type Viewports}	"Viewports" is a hard-coded parameter.
{Config *int*/*string*}	This parameter defines the viewports' configuration. It can be one of two values:

	"0" or "SingleView" for one viewport. "1" or "QuadView" for four viewports.
{Viewport1 *string*}	'*string*' is the name of the camera that will be attached to viewport 1. Optionally, 'string' can be set to "Disable" to disable viewport 1. Viewport 1 is the only viewport in "SingleView" and the Top-Left viewport in "QuadView".
{Viewport2 *string*}	'*string*' is the name of the camera that will be attached to viewport 2. Optionally, 'string' can be set to "Disable" to disable viewport 2. Viewport 2 is the Top-Right viewport in "QuadView".
{Viewport3 *string*}	'*string*' is the name of the camera that will be attached to viewport 3. Optionally, 'string' can be set to "Disable" to disable viewport 3. Viewport 3 is the Bottom-Left viewport in "QuadView".
{Viewport4 *string*}	'*string*' is the name of the camera that will be attached to viewport 4. Optionally, 'string' can be set to "Disable" to disable viewport 4. Viewport 4 is the Bottom-Right viewport in "QuadView".

Example: SET {Type Viewports} {Config QuadView}
SET {Type Viewports} {Config QuadView} {Viewport1 Camera1} {Viewport2 Camera2} {Viewport3 Disable} {Viewport4 Disable}
SET {TypeViewports} {Viewport1 Disable}

- Control a camera:

NOTE: There is a difference between moving and controlling a camera. Moving a camera (i.e. pan/tilt) is achieved by controlling the mission package that the camera is attached to (see the next subsection). Controlling a camera is used to set its field of view.

SET {Type Camera} {Name *string*} {FOV *float*} {Name *string*} {FOV *float*} …
Where:

{Type Camera}	"Camera" is a hard-coded parameter.
{Name *string*}	'*string*' is the name of the camera, as described in the USARBot.ini file, for

which we want to change its field of view.

{FOV *float*} '*float*' is the desired camera's field of view in radians. Smaller fields of view give a zoom-in effect. If '*float*' is zero, the default field of view will be used. If '*float*' is greater than the maximum field of view, the maximum field of view will be used. If '*float*' is smaller than the minimum field of view, the minimum field of view will be used.

Example: SET {Type Camera} {Name Camera} {FOV 1} will set the field of view of camera "Camera" to 1 radian.
SET {Type Camera} {Name Camera} {FOV 0} will set the field of view of camera "Camera" to its default field of view.

- Control a Mission Package:

A mission package is constructed of a series of connected elements. Of course, we can control the joints one by one to set the mission package's pose. Here, we provide another command to directly set the package's terminal pose and let USARSim control every element's joint for us. If a camera is mounted on a pan/tilt mission package, we can use this command to control the camera's pose. Using mission package control commands, we can have multiple cameras and control them separately. The command's format is:

MISPGK {Name *string*} {Link *int*} {Value *float*}{Order *int*} {Link *int*} {Value *float*}{Order *int*} …

Where:

{Name *string*} '*string*' is the mission package's name.

{Link *int*} '*int*' is the link number that will be moved using the next parameters.

{Value *float*} The '*float*' parameter is the value used to move the link. What this parameter describes depends on the order given. If the order is 0, '*float*' is the absolute angle, in radians, for a revolute joint or the distance, in meters, for a prismatic joint. If the order is 1, '*float*' is the velocity, in m/s. If the order is 2, '*float*' is the torque.

{Order *int*} '*int*' is optional (0 by default) and indicates the control mode. '0' means angle control, '1' means speed control, and '2' means torque control.

Example: MISPKG {Name CameraPanTilt} {Link 1} {Value 1.5} {Link 2} {Value 0}
MISPKG {Name TalonArm} {Link 1} {Value 0.70} {Link 3} {Value 0.1} {Order 1}

TIPS: Pan/tilt mission packages enable the use of multiple cameras with independent control on each of them.

- Control a sensor/effecter:

  This command is used to send a command to a sensor or effecter. The command looks like:

  SET {Type *string*}{Name *string*}{Opcode *string*} {Params *value*}

  Where:

{Type *string*}	'*string*' is the sensor or effecter's type.
{Name *string*}	'*string*' is the sensor or effecter's name.
{Opcode *string*}	'*string*' is the operation code. Different sensors or effecters. Valid opcodes are defined in Section 8.
{Params *value*}	'*value*' are the parameters associated with the operation command.

  Example: SET {Type Odometry} {Name Odo1} {Opcode RESET} {Params 1}

- Control the Trace

  USARSim has the capability of tracing the path that a robot takes. Colored Navigation Points are dropped into the world as the robot travels; effectively "tracing" the robot's path. Tracing can be used with the following command:

  Trace {On *bool*} {Interval *float*} {Color *int*/*string*}

  Where:

{On *bool*}	'*bool*' tells USARSim to start/stop tracing the robot's path. Default value is false.
{Interval *float*}	'*float*' is a number that dictates how many seconds will go by before USARSim drops a Navigation Point. Please note that this number is in unreal units and that its default value is 0.
{Color *int*/*string*}	The value for this parameter can be entered as an *int* or a *string*. It determines the color of the trace as follows: '*0*' or '*Red*' – Red Trace [DEFAULT] '*1*' or '*Yellow*' – Yellow Trace

'*2*' or '*Green*' – Green Trace
'*3*' or '*Cyan*' – Cyan Trace
'*4*' or '*White*' – White Trace
'*5*' or '*Blue*' – Blue Trace
'*6*' or '*Purple*' – Purple Trace

- Query the robot/sensor/effecter/mission package

  There are two types of query command. One queries the geometry information, and another queries the configuration information.

  - GETGEO {Type *string*} {Name *string*}
    The "{Name *string*}" is optional. If it's omitted, the command queries the geometry information for all the sensors/effecters with the specified type. Otherwise, only the sensor/effecter with the name and type will be queried. The return message is a GEO message.
    Example: GETGEO {Type Sonar}
    GETGEO {Type MisPkg}
    GETGEO {Type Robot}
    GETGEO {Type Effecter}
  - GETCONF {Type *string*} {Name *string*}
    Queries the configuration information for a type (when "{Name *string*}" is omitted) or specified sensor/effecter. The return message is a CONF message.
    Example: GETCONF {Type RangeScanner}
    GETCONF {Type MisPkg}
    GETCONF {Type Robot}
    GETCONF {Type Effecter}

- Manage viewpoint

  SET {Type Camera} {Robot *string*} {Name *string*}{Client *ip*}

This command sets the viewpoint of the specified Unreal Client to a robot's camera. The unreal client is defined by '{Client *ip*}' where '*ip*' is the client's IP address. Please note USARSim doesn't support the loopback ip address. So don't use "127.0.0.1" as the parameter. The robot is specified by '{Robot *string*}' where '*string*' is the robot's name. And the camera is specified by '{Name *string*}' where '*string*' is the camera's name. Once the client's viewpoint is set, we can NOT manually change it until we release the viewpoint control. To release the control, we send another SET command without '{robot *name*}'. For example, we can send "SET {Type Camera} {Client 10.0.0.2}" to release the viewpoint control on client 10.0.0.2.

We can use this command at anytime and anyplace. This command can be sent either from a robot's controller or from other applications such as the ImageServer.

NOTE: USARSim doesn't support the **loopback** ip address. Please don't use "127.0.0.1" as the parameter.

10 Sensors

In USARSim, every sensor is an instance of a sensor class (a sensor type). All of the objects of a sensor class have the same sensor capability. You can configure the capability of a sensor class to satisfy your needs or you can create a new sensor from an existing sensor class and change its properties to get a new type of sensor.

In USARSim, all of the sensors can add noise and apply distortion to their data except for the state sensor and robot camera. This noise is applied to the output values reported by the sensor and not to the control values. For example, the angle between range scans for a scanning laser is always correct, and a sensor will always point to the location that is commanded. The distortion curve is a function such that output_data = distortion(input_data), and the function itself is defined by a series of (x,y) points connected by straight line segments. If the input_data is outside of the defined range, zero will be returned. For the sensor output, changing parameters will give different quality sensor data. Every sensor has a "Weight" attribute associated to it[3]. Besides this, it's also possible to associate a timestamp to the sensor data. To do this, we can configure the sensor with the variable 'bWithTimeStamp' set to true in the configuration file.

In this section, we will explain how the sensors work and how to configure them. To learn how to build your own sensor, please read section 14.2.

10.1 State Sensor

10.1.1 How the sensor works

The State sensor reports the robot's state. Basically, it just checks the robot's state in the Unreal engine and then sends it out. Please go to section 7.3 for a detailed look at the state message.

10.1.2 How to configure it

None. We do not need to configure it.

10.2 Range Sensor

10.2.1 How the sensor works

The range sensor is used to detect distance. There are two types of range sensor in USARSim: sonar and IR. Basically, the range sensor is simulated by emitting a line from the sensor position along the direction of the sensor in the Unreal world. The first point met by the line is the hit point. And the distance between the hit point and the sensor is the returned range value. If the range is beyond the sensor's detection range, the sensor will return the maximum detection range. Before the data is sent back, a random number is added to simulate random noise. Then a distortion curve is used to interpolate the range data to simulate the real range sensor.

For the sonar sensor, instead of emitting one line from the sensor, it tries to emit several lines within its beam cone. The shortest distance detected by the lines is the

[3] Currently, this attribute is only used to calculate the robot's payload. It will not affect the sensor's dynamic characteristic. The real physical variable used in Unreal Engine is the "Mass" variable.

value returned by the sonar sensor. For IR sensor, only one line is used. However the line can cross through transparent materials (glass).

10.2.2 How to configure it

We configure the range sensor in the USARBot.ini file. The sonar and IR sensor's configuration looks like:

```
[USARBot.SonarSensor]
HiddenSensor=true
bWithTimeStamp=true
Weight=0.4
MaxRange=5.0
BeamAngle=0.3491
Noise=0.05
OutputCurve=(Points=((InVal=0.000000,OutVal=0.000000),(InVal=5.000000,
OutVal=5.000000)))

[USARBot.IRSensor]
HiddenSensor=true
bWithTimeStamp=true
MaxRange=5.0
Noise=0.05
OutputCurve=(Points=((InVal=0.000000,OutVal=0.000000),(InVal=5.000000,
OutVal=5.000000)))
```

Where

HiddenSensor	Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it is not necessary to show the sensor. When you want to confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
bWithTimeStamp	Indicates whether the timestamp in the Unreal Engine will be associated with the sensor data.
Weight	The weight of the sensor in kg.
MaxRange	The maximum distance that can be detected in meters.
BeamAngle	The sensor's detection cone in radians.
Noise	The relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data where random(noise) returns a value between –noise and +noise.
OutputCurve	The distortion curve. It is constructed by a series of points that describe the curve.

10.3 Range Scanner Sensor

10.3.1 How the sensor works

The range scanner sensor is very similar to the range sensor. In USARSim, we treat the range scanner sensor as a series of range sensors. The data is obtained by rotating the range sensor from the start direction to the end direction in a fixed step. The step interval is calculated from the resolution. The sensor can work in two modes. In the automatic mode, it automatically scans data in specified time intervals. In manual mode, it only works when it gets a scan command; and for every scan command, it only scans once.

There are two kinds range scanners, RangeScanner and IRScanner. The RangeScanner sensor uses the range sensor (only emits one detection line) to scan the environment. While the IR scanner uses the IR sensor (the detection line can cross transparent materials) to scan the environment. Both sensors use the SET command to control the scan behavior. We list the Opcode, Params and returned response Status below:

Table 3 Range scanner control command

Opcode	Params	Status
SCAN	None	"OK": successfully scanned "Failed": didn't scan. It may be caused by an invalid command.

10.3.2 How to configure it

The RangeScanner and IRScanner sensor share the same configuration format. We only list RangeScanner's configuration below. For IRScanner, the only difference is that the section name should be USARBot.IRScanner.

```
[USARBot.RangeScanner]
HiddenSensor=False
bWithTimeStamp=False
Weight=0.4
MaxRange=1000.000000
ScanInterval=0.5
Resolution=800
ScanFov=32768
bPitch=false
bYaw=true
Noise=0.0
OutputCurve=(Points=((InVal=0.000000,OutVal=0.000000),(InVal=1000.0000
00,OutVal=1000.000000)))
```

Where

HiddenSensor Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it's not necessary to show the sensor. When you want to confirm if

	the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
bWithTimeStamp	Indicates whether the timestamp in the Unreal Engine will be associated with the sensor data.
Weight	The weight of the sensor in kg.
MaxRange	The maximum distance that can be detected.
ScanInterval	It is the time interval between scanning used in automatic mode.
Resolution	The scan resolution, the step length of rotating from start direction to the end direction. The unit is integer. 65535 means 360 degrees.
ScanFov	The scan field of view as an integer. 65535 means 360 degrees.
bPitch	Boolean value that indicates the scan plane. True means scanning in the tilt plane (x-z plane).
bYaw	Boolean value that indicates the scan plane. True means scanning in the pan plane (x-y plane).
Noise	The relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data where random(noise) returns a value between –noise and +noise.
OutputCurve	The distortion curve. It is constructed by a series of points that describe the curve.

Note: Too much sensor data will impact the system. Do not set the resolution or scan frequency too high.

10.4 Odometry Sensor

10.4.1 How the sensor works

The simulated odometry sensor uses the robot's left and right wheel encoders to estimate the robot's pose. It describes a pose as x, y position and (head's) theta angle relative to the start location and robot's head direction. The positive x-axis and y-axis are the robot's head direction and right hand direction in the start location. The sensor applies a very simple algorithm to calculate the pose by using the wheel's diameter, left and right wheel's separation and the wheels' spin angle. The sensor's errors come from the diameter and wheel separation measurement error, the encoder's resolution and the error introduced by the simple algorithm.

When we use the sensor we need to specify which wheels are the left and right wheels used in pose estimation. If we don't specify the wheels, the sensor will try to find and use the left-most wheel and right-most wheel to calculate the pose. While using the sensor, we can reset the sensor to use the current location and head direction as the pose estimation's reference point. The command we used to reset the sensor is SET. And the Opcode, Params and returned response Status are listed below:

Table 4 Odometry sensor control command

Opcode	Params	Status

RESET	None	"OK": successfully reset "Failed": didn't reset the sensor. It may be caused by an invalid command.

10.4.2 How to configure it

We configure the odometry sensor in the USARBot.ini file. The configuration looks like:

```
[USARBot.OdometrySensor]
HiddenSensor=true
bWithTimeStamp=False
Weight=0.4
ScanInterval=0.2
EncoderResolution=0.01
LeftTire=LeftFWheel
RightTire=RightFWheel
```

Where

HiddenSensor	Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it's not necessary to show the sensor. When you want to confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
bWithTimeStamp	Indicates whether the timestamp in the Unreal Engine will be associated with the sensor data.
Weight	The weight of the sensor in kg.
ScanInterval	The time interval in seconds between pose estimates.
EncoderResolution	The minimum wheel spin angle the sensor can recognize in radians.
LeftTire	The left wheel that will be used in pose estimating. If it is a null string, the left-most wheel will be used.
RightTire	The right wheel that will be used in pose estimating. If it is a null string, the right-most wheel will be used.

10.5 GPS Sensor

10.5.1 How the sensor works

The GPS sensor finds the current robot position in meters and converts it to latitude and longitude. The GPS sensor follows a modified version of the NMEA 183 GGA GPS format; modified to keep the structure of the USARSim communication interface. Please see the "Messages" section (Section 8.3) of this manual for the specific SEN message employed by the GPS sensor. The error model is based on a line-of-sight signal model that dictates how many satellites can be used by the receiver. The number of satellites seen is, in turn, used to dictate the how much error

is injected into the readings. The sensor assumes a flat earth where all X-axis motion is converted to latitude and all Y-axis motion is converted to longitude. While, in most cases, the global X-axis points to the North, it is worthwhile noting that this is not always the case due to singularities that may occur. Indeed, the sensor handles singularities that occur at the earth's pole (at 90 degree North and 90 degree South) and on the longitude (at 180 degree West and 180 degree East). In other words, and as an example, driving along the X-axis at 89 degrees and 59.9 minutes will increase the latitude component of the GPS until 90 degree North is reached. At that point, the latitude component will decrease (meaning that the global X-Axis now points to the south). See Figure 1 for an example of the flat world representation at different quadrants of the earth. These singularities exist and are taken into account due to the flat assumption of the virtual worlds and the spherical shape of the earth.

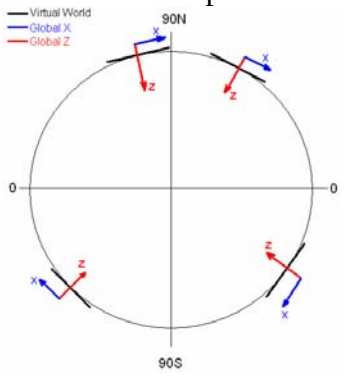


Figure 21 - Singularity Representation

10.5.2 How to Configure it

The common bHiddenSensor and bWithTimeStamp variables are included in the [USARBot.GPSSensor] section of the USARBot.ini file that, when set to true, hide the sensor and add time information to the GPS message, respectively. In addition, the amount of noise injected into the sensor is adjusted with the maxNoise and minNoise variables. The maxNoise variable dictates the maximum amount of noise injected in the sensor when the receiver can only use four satellites. The minNoise variable dictates the maximum amount of noise injected in the sensor when the receiver sees the maximum number of satellites: twelve.

Please note that the outdoor worlds have to follow the typical skybox procedure of using the wm_texture.sky texture all around the world (this texture will then be replaced by the world's skybox). Failure to comply with this standard will result in a non-working GPS sensor.

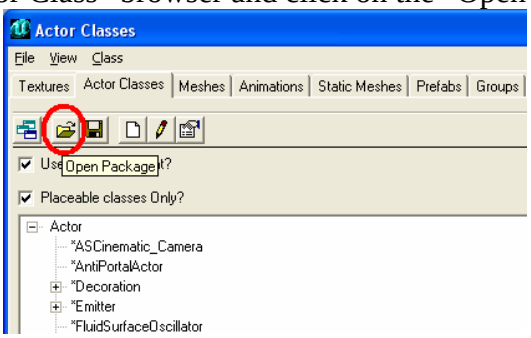
The main problem with using a GPS sensor in a virtual environment is the mapping of a virtual location to a real one. Since USARSim worlds do not inherently have a GPS coordinate associated with them, the GPS sensor provides three ways to allocate a GPS coordinate to a virtual world: 1) adding a special object in the virtual world (a ReferenceGPSCoordinate object), 2) using the USARBot.ini file, or 3) modifying the actual GPSSensor class. A GPS reference point is acquired in this order: if a ReferenceGPSCoordinate object is found inside the map, it is used as the reference point. Otherwise the ZeroZeroLocation inside the GPSSensor section of USARBbot.ini is used. If there is no ZeroZeroLocation inside the USARBot.ini file, a

default ZeroZeroLocation is used. The next paragraphs describe how to assign a GPS coordinate to a virtual map, using one of these three techniques.

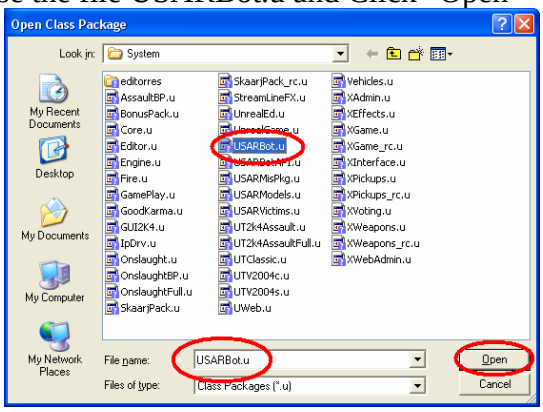
- Reference GPS Coordinate

A ReferenceGPSCoordinate object has been created so that world creators can add a GPS coordinate to a particular point in a map.

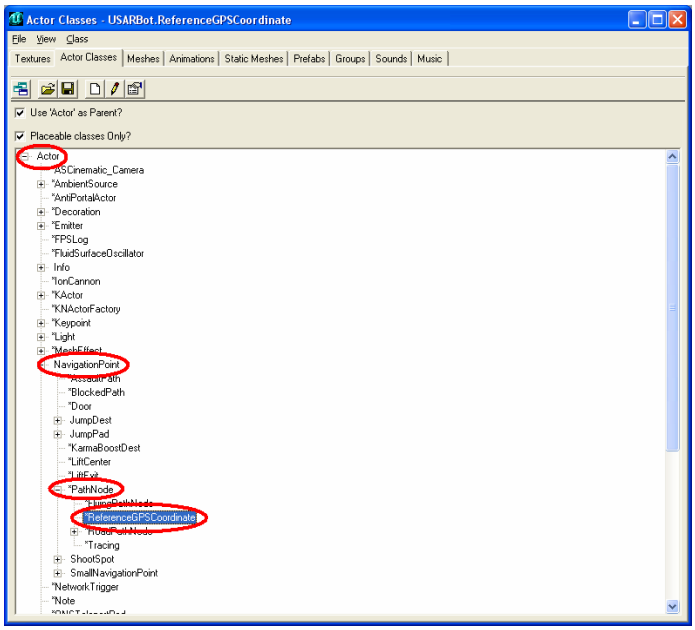
1) Make sure that USRASim is compiled
2) Open the desired map in UnrealEd
3) Open the "Actor Class" browser and click on the "Open Package" icon



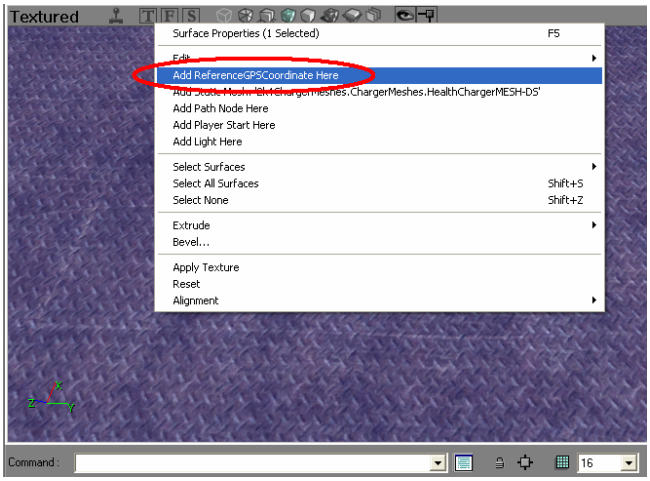
4) Enter or Choose the file USARBot.u and Click "Open"



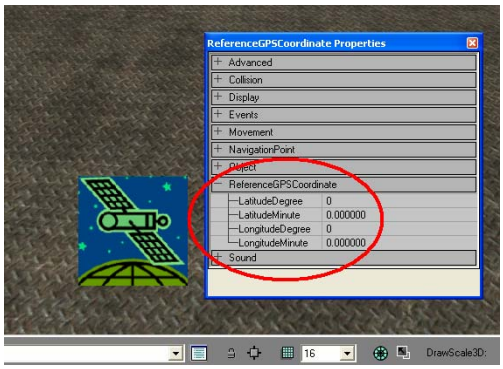
5) Navigate to Actor->NavigationPoint->PathNode and Select ReferenceGPSCoordinate



6) Right-Click the location that you want to give a GPS coordinate in the 3D view of the UnrealEd and select “Add ReferenceGPSCoordinate Here”



7) A new icon shows up in the world. Place it to your desired location and double-click on it to bring up the properties. Change the ReferenceGPScoordinate properties to desired values. Please note that a South direction is achieved by giving negative values to the degree and minutes of the latitude component and that a West direction is achieved by giving negative values to the degree and minutes of the longitude component.



- USARBot.ini File

Open the USARBot.ini file and find the [USARBot.GPSSensor] section. Under that section add or modify the line:

ZeroZeroLocation=(LatitudeDegree=39,LatitudeMinute=8.0273,LongitudeDegree=-359,LongitudeMinute=59.9996)

Please note that a South direction is achieved by giving negative values to the degree and minutes of the latitude component and that a West direction is achieved by giving negative values to the degree and minutes of the longitude component. The latitude and longitude components given in that section will provide the GPS coordinate referring to the (0,0) location of the map.

- GPSSensor.uc File

If no reference location is found in the map and in the USARBot.ini file, a default GPS coordinate will be used for the (0,0) map location. That GPS coordinate is defined in the defaultproperties section of the GPSSensor.uc file.

10.6 INS Sensor

How the sensor works

The INS Sensor implemented in USARSim uses the Gaussian random number generator found in USARUtils.uc to add noise to the ground truth reading of angular velocities and distance traveled per time step. These values are then used to estimate the robot's current location and orientation. Depending on the configuration, errors may cumulate and therefore diverge slowly from the ground truth.

General description of how algorithm works

1. Calculates angular velocity from ground truth
2. Uses Gaussian random number generator to add noise to each of the angular velocities components
3. Uses these angular velocity components to update the vehicle's estimation of the current orientation
4. Calculates the total distance traveled in one time step.
5. Adds Gaussian noise to distance
6. Decomposes the distance traveled in that time step into distance vectors using polar coordinates transforms on the current estimation of the vehicle's orientation.
7. Add these estimates to the estimates of the vehicle's location.

**NOTE:* All noise is proportional to rate of change, more change causes more error.
The INS sensor uses USARSim ground truth for location and orientation as the initial estimate of the robot's location and orientation. This sensor contains a 'SET' command that allows the user to set the current INS estimation of the position and orientation. **NOTE:* This will not set the sensor estimation back to ground truth. Also, if the Drifting mode is active, this command will reset the drifting to 0.

Opcode	Params	Status
POSE	x,y,z,r,p,y	"OK": successfully set the robot's estimation of pose "Failed": didn't set the robot's estimate of pose. It may be caused by an invalid command or the absence of commas between arguments.

9.5.2 How to configure it

The Gaussian random number generator uses the Box-Muller method and is based on a mean and sigma. This function is included in a class called USARUtils.uc. This function requires persistent variables, and therefore needs to be implemented as an object and not as static functions. The sensor works in two ways. In its simplest form it does not drift over time and its accuracy is determined by the sigma parameter only. It is however also possible to set the sensor so that it drifts over time, thus resembling error dynamics due to double integration observed in real sensors. When drifting is enabled, the *Precision* parameter characterizes the drift rate. Please note that when drifting, components drift in possibly different directions and at different rates. The initialization of the USARUtils object is in sensor.uc so that all sensors can utilize this function. Therefore, all sensors can use a "Sigma" in the USARBot.ini file to adjust the distribution of noise.

```
[USARBot.INS]
HiddenSensor=true
bWithTimeStamp=False
Weight=0.1
ScanInterval=0.2
Drifting=false
Precision=1000
Sigma=0.1
```

Where

HiddenSensor	Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it's not necessary to show the sensor. When you want to confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
bWithTimeStamp	Indicates whether the timestamp in the Unreal Engine will be associated with the sensor data.
Weight	The weight of the sensor in kg.

ScanInterval	The time interval in seconds between pose estimates.
Mean	Sets the mean used by the Gaussian random number generator used to produce noise in the sensor.
Sigma	Sets the standard deviation used by the Gaussian random number generator used to produce noise in the sensor.
RightTire	The right wheel that will be used in pose estimating. If it is a null string, the right-most wheel will be used.
Drifting	Boolean value. If true the sensor drifts, if false it does not (default is false)
Precision	Numeric value indicating the rate of drifting. The higher the value, the less the drifting (default is 1000). This parameter has effect only when Drifting is true.

10.7 Encoder Sensor

10.7.1 How the sensor works

The sensor measures a part's spin angle around the sensor's axis. The returned value is a tick count which is the real angle divided by the sensor's resolution. How we mount the sensor will decide what axis's spin angle will be measured. The sign of the count is also decided by the direction we mount the sensor. For example, mounting the sensor on the front or back side of a wheel will give us a different count sign. Similar to the INU sensor, we can use the SET command to reset the tick count. The Opcode, Params and returned response Status are listed below:

Table 5 Encoder sensor control command

Opcode	Params	Status
RESET	None	"OK": successfully reset "Failed": didn't reset the sensor. It may be caused by an invalid command.

10.7.2 How to configure it

We configure the sensor in the USARBot.ini file. The configuration looks like:

```
[USARBot.EncoderSensor]
HiddenSensor=true
bWithTimeStamp=False
Weight=0.4
Resolution=0.01745
Noise=0.005
```

Where

HiddenSensor	Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it's not necessary to show the sensor. When you want to

	confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
bWithTimeStamp	Indicates whether the timestamp in the Unreal Engine will be associated with the sensor data.
Weight	The weight of the sensor in kg.
Resolution	The minimum spin angle the sensor can recognize in radians.
Noise	The relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data where random(noise) returns a value between –noise and +noise.

NOTE: Mounting the sensor on the front or back side of a wheel will give us a different count sign.

10.8 Touch Sensor

10.8.1 How the sensor works

We use the same method used by the range sensor to simulate the touch sensor. Every touch sensor is treated as a button. We emit several lines from the button's surface to detect the objects in front of the sensor. When one object is close enough to the sensor (less than 4.7mm), the sensor will send out a touch signal.

10.8.2 How to configure it

We configure the sensor in the USARBot.ini file. The configuration looks like:

```
[USARBot.TouchSensor]
HiddenSensor=true
bWithTimeStamp=False
Weight=0.4
Diameter=0.01
```

Where

HiddenSensor	Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it's not necessary to show the sensor. When you want to confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
bWithTimeStamp	Indicates whether the timestamp in the Unreal Engine will be associated with the sensor data.
Weight	The weight of the sensor in kg.
Diameter	The diameter of the sensor button in meter.

10.9 RFID Sensor

10.9.1 How the sensor works

The RFIDSensor simulates a RFID reader and is used to detect RFID tags that are in sensor range. It also allows one to read and write their memory. RFID tags can be added in the virtual world either from UnrealEd or they can be dynamically released by the RFIDReleaser effecter.

Refer to http://en.wikipedia.org/wiki/RFID for further information on RFID technology.

10.9.2 How to configure it

Like any other USARSim class, the sensor can be configured from the USARBot.ini file. In the [USARBot.RFIDSensor] section you can set these parameters:

- **SensingMode** [default: **Attenuation**]: it can be **Radius**, **Obstacle**, **Attenuation**. It specifies what signal propagation model to use.

Radius and Obstacle mode parameters:

- **MaxRange** [default: **3**] (**m**): sensor range. It's only used by Radius and Obstacle sensing modes.
- **MaxSingleShotRange** [default: **6**] (**m**): same as MaxRange but for single shot tags.

Attenuation mode parameters:

- **dBmTXPower** [default: **36**] (**dBm**): sensor TX power in dBm.
- **dBmRXSensibility** [default: **-88**] (**dBm**): sensor RX sensibility in dBm.
- **dBObstacleAttenuation** [default: **3.5**] (**dB**): attenuation due to an obstacle.

Other parameters:

- **bTraceRFIDs**:[default: **false**] : if set to true then USARSim will draw 3D lines from sensor to rfid tag (slow, use it only for debugging).
- **bAlwaysReadRFIDmem** [default: **false**]: if you want to read tag memory every time you detect it. That means that if you sense 10 times per second, you will read the memory content of all tags that are in range 10 times per second.
- **HiddenSensor** [default: **true**]: show/hide the sensor in the simulator. We recommend setting it to true if it's not necessary to show the sensor. When you want to confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
- **bWithTimeStamp** [default: **true**]: indicates whether include the timestamp in the the sensor data message.

- **Weight** [default: **0.4**] (**kg**): The weight of the sensor in kg.

10.9.3 Choosing the right SensingMode

10.9.3.1 Radius

Radius mode (Figure 22) only takes into account the sensor range (**MaxRange**). The sensor can read any tag in this range ignoring obstacles.

Traced lines (if **bTraceRFIDs** is **true**): all the lines are green.

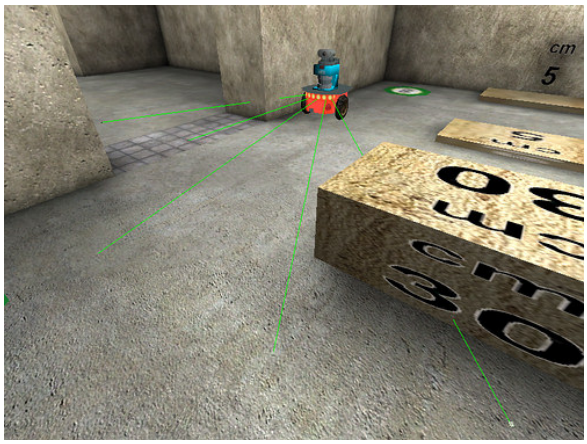


Figure 22: Radius mode

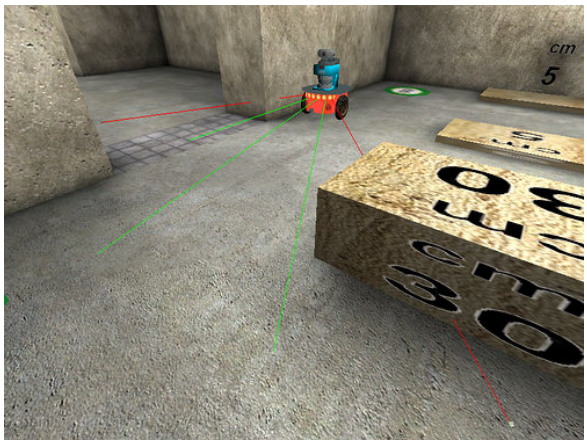


Figure 23: Obstacle mode

10.9.3.2 Obstacle

Obstacle mode (Figure 23) is the same as Radius, but it also takes into account obstacles. Tags that are in sensor range but lie behind an obstacle are invisible to the sensor.

Traced lines (if **bTraceRFIDs** is **true**): green lines for reachable tags, red lines for hidden tags.

10.9.3.3 Attenuation

Attenuation mode (Figure 25) uses a signal attenuation model. It considers both distance and obstacles. The signal strength is function of the transmitter power (**dBmTXPower**) and distance:

$$P_s = \text{dB mTXPower} - 2 \cdot [10 \cdot (5.25 + 2 \cdot \log_{10}(d))]$$ [4]

[4] MaxStram Application Note: “Indoor Path Loss”

note: the factor 2 means that we are considering forward and backward propagation (considering the passive tag as the source of the reflected signal).

If the received signal is lower than the sensor sensibility (**dBmRXSensibility**) then the tag is out of range. If there is an obstacle we attenuate the signal by a fixed amount (**dBObstacleAttenuation**). In this simplified model we consider only one obstacle. This is an acceptable approximation because of the very short range of the passive RFID tags.

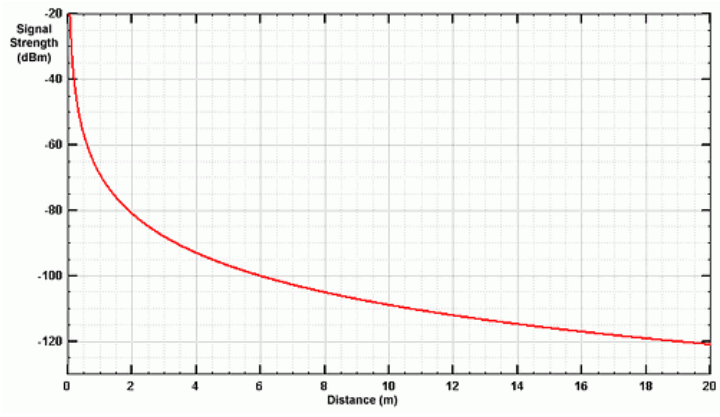


Figure 24: Signal strength over distance with dBmTXPower = 36

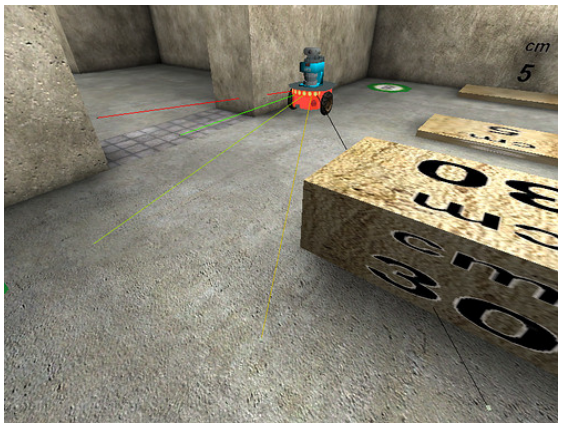


Figure 25: Attenuation mode

Traced lines (if **bTraceRFIDs** is **true**): line color represents the signal strength (from green to red). If the signal is too low the line is black (tag is unreachable).

10.9.4 Detecting RFID Tags

When you are near one or more RFIDTags you will receive the following string:

SEN {*Type* RFID} {*Name* RFID} {*ID* 1} {*Mem* 0}

if you detect one RFID, or:

SEN {*Type* RFID} {*Name* RFID} {*ID* 1} {*Mem* 0} {*ID* 2} {*Mem* 0} ...

if you detect more then one at the same time.

- **ID**: is the unique RFIDTag identifier.
- **Mem**: Is the RFIDTag memory content, "0" if empty. This field can be deactivated by setting to false **bAlwaysReadRFIDmem**.

10.9.5 Reading RFID Tags Memory

To read the memory of the RFIDTag you can use the following command:

> **SET** {*Type* RFID} {*Name* RFID} {*Opcode* Read} {*Params* RFIDTagID}

Where RFIDTagID is the ID of the RFIDTag you want to read.

Success:	**RES** {*Time* ...} {*Type* RFID} {*Name* RFID} {*Status* OK} {*ID* RFIDTagID} {*Mem* MemoryContent}
Failure:	**RES** {*Time* ...} {*Type* RFID} {*Name* RFID} {*Status* Failed}

For example, failure can happen when the RFID tag is out of the sensor range.

10.9.6 Writing RFID Tags Memory

Using the same RFIDSensor you can write the RFIDTag memory using this command:

> **SET** {*Type* RFID} {*Name* RFID} {*Opcode* Write} {*Params* RFIDTagID MemoryContent}

Where RFIDTagID is the ID that identifies the RFIDTag you want to write, MemoryContent is the string you want to write in the RFIDTag.

Success:	**RES** {*Time* ...} {*Type* RFID} {*Name* RFID} {*Status* OK}
Failure:	**RES** {*Time* ...} {*Type* RFID} {*Name* RFID} {*Status* Failed}

10.9.7 Erasing RFID Tags Memory

To erase the RFIDTag memory you can both write a "0" string or, more easily:

> **SET** {*Type* RFID} {*Name* RFID} {*Opcode* Write} {*Params* RFIDTagID}

Where RFIDTagID is the ID that identifies the RFIDTag you want to erase.

Success:	**RES** {*Time* ...} {*Type* RFID} {*Name* RFID} {*Status* OK}
Failure:	**RES** {*Time* ...} {*Type* RFID} {*Name* RFID} {*Status* Failed}

10.10 Victim and False Positive Sensor

10.10.1 How the sensor works

The Victim and False Positive Sensor simulates a victim location sensor. The sensor's operation is very similar to that of the Range Scanner. Indeed, the victim sensor sends out a number of traces (as defined by its configuration section) and looks to see if they hit one of the following: a victim part (e.g. leg, arm, head, etc…) or a false alarm (e.g. leg, arm, head, etc…). The sensor returns all the victim's parts that have been hit (including possible false alarms) and it is up to the controller/user to determine whether the responses indicate a victim or a false alarm.

Please note that, for proper operation, the victim sensor should always be mounted on a camera.

10.10.2 How to configure it

The Victim and False Positive Sensor configuration in the USARBot.ini file looks like:

```
[USARBot.VictSensor]
HiddenSensor=true
Distance=6
HorizontalFOV=0.6981317
HorizontalStep=0.0698131
VerticalFOV=0.6981317
VerticalStep=0.0698131
bWithTimeStamp=true
bShowResults=false
Mean=0.0
Sigma=0.01
```

Where

HiddenSensor	Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it's not necessary to show the sensor. When you want to confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
Distance	The maximum detection range, in meters. The traces will be sent 'Distance' meters.
HorizontalFOV	The field of view in the horizontal direction (x-y plane), in radians.
HorizontalStep	The amount of radians between two traces in the horizontal direction (x-y plane).
VerticalFOV	The field of view in the vertical direction (x-z plane), in radians.
VerticalStep	The amount of radians between two traces in the vertical

	direction (x-z plane).
bWithTimeStamp	Indicates whether the timestamp in the Unreal Engine will be associated with the sensor data.
bShowResults	Indicates whether the graphical interface sgould be displayed. Useful for testing purposes but will be set to 'false' in most cases.
Mean	Sets the mean used by the Gaussian random number generator used to produce noise in the sensor.
Sigma	Sets the standard deviation used by the Gaussian random number generator used to produce noise in the sensor.

10.11 Sound sensor

10.11.1 How the sensor works

The Sound sensor detects sound sources in USARSim. The sound sensor finds all the sound sources and calculates the source that is the loudest at the robot's location. The loudness decreases with the square of the distance. Currently, the only available sound sources are victims. Please refer to section 14.1.3 about how to associate a sound source to a victim.

10.11.2 How to configure it

The sound sensor configuration in the USARBot.ini file looks like:

```
[USARBot.SoundSensor]
HiddenSensor=True
Weight=0.4
Noise=0.05
OutputCurve=(Points=((InVal=0.000000,OutVal=0.000000),(InVal=1000.000000,OutVal=1000.000000)))
```

Where

HiddenSensor	Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it's not necessary to show the sensor. When you want to confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
Weight	The weight of the sensor in kg.
Noise	The relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data where random(noise) returns a value between –noise and +noise.
OutputCurve	The distortion curve. It is constructed by a series of points that describe the curve.

10.12 Human-motion sensor

10.12.1 How the sensor works

The Human motion sensor simulates a pyroelectric sensor. It's simulated by finding all the victims that are in the FOV of the sensor within the testing range. The closest moving victim will be checked. Its distance from the robot and its motion speed and amplitude are used to calculate the probability of whether it is a human motion.

10.12.2 How to configure it

The human-motion sensor configuration in the USARBot.ini file looks like:

```
[USARBot.HumanMotionSensor]
HiddenSensor=True
Weight=0.4
MaxRange=1000
FOV=60
Noise=0.1
OutputCurve=(Points=((InVal=0.000000,OutVal=0.000000),(InVal=1000.0000
00,OutVal=1000.000000)))
```

Where

HiddenSensor	Boolean value is used to indicate whether the sensor will be visually shown in the simulator. Setting it to true will hide the sensor. We recommend setting it to true if it's not necessary to show the sensor. When you want to confirm if the sensor is placed in the correct position and has the correct direction, you can temporarily set it to false.
Weight	The weight of the sensor in kg.
MaxRange	The maximum detecting range in meters.
FOV	The field of view of the sensor in integer. 65535 means 360 degrees.
Noise	The relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data where random(noise) returns a value between –noise and +noise.
OutputCurve	The distortion curve. It is constructed by a series of points that describe the curve.

10.13 GPS Sensor

10.13.1 How the sensor works

The GPS sensor finds the position in meters and then converts it to latitude and longitude. It assumes a flat earth and assumes that the Y axis points due north. All Y axis motion is converted to latitude. All X-axis motion is longitude. The

sensor does not check for an open sky and may thus report GPS when it would not be available.

Any robot can navigate from relative GPS. Relating the absolute GPS to the map requires an RNDF. For RNDF specifications, see http://www.darpa.mil/grandchallenge/docs/RNDF_MDF_Formats_120606.pdf. DARPA's Dec 1, 2006 revision specifies that latitude and longitude are fixed point numbers in the ITRF00 reference frame. The current RNDF uses the previous specification, which gives latitude and longitude as floating point numbers in the WGS84 frame.

Typical output is
SEN {Type GPS} {Name GPS1} {Latitude 47, 40, 5899} {Longitude -122, 18, 9360}
This example represents 47°N 40.5899', 122°W 18.9360'

The map on which GPS is used should have exactly one GPSStart object. This gives the latitude and longitude of the origin of the map. It also has scale factors that specify how many meters correspond to one degree of latitude or longitude at this position. The scale factors may be negative to point the axes the opposite way.

10.14 Robot Camera

10.14.1 How the sensor works

The Camera is a special sensor in USARSim. The scenes viewed from the camera are captured by attaching the viewpoint to the camera in the Unreal engine. USARSim currently supports any amount of cameras. We can use the SET {Type Camera} command to control a camera's field of view and the MISPKG command to control a camera's pan-tilt frame. Information about the camera commands and response messages can be found in the preceding sections of this manual. Every camera has a default, maximum and minimum field of view. If the field of view in the SET command is out of the camera's FOV range, it will adjust it to the minimum or maximum FOV range.

We provide two ways to simulate camera feedback.

1) Directly using the Unreal Client as video feedback.

   This is the easiest way. However, it can't simulate the frame rate of the real robot. There are two ways to directly use the Unreal Client. One is using the Unreal Client as a separate sensor panel. The other is embedding the Unreal Client into the user interface. For details about embedding the Unreal Client into user's application please see section 14.5.1.

2) Capturing the scenes in Unreal Client and using these pictures as video feedback.

   This approach is very close to how the real camera works, but it's technically difficult. The camera feedback can be either directly or remotely received

a. Directly capture Unreal Client
   The idea is to capture the pictures in the Unreal Client and use them on the interface. There are many capturing technologies. We use Detours (http://www.research.microsoft.com/sn/detours/) to access the back buffer of DirectX and get the scene pictures. The advantage of this approach is that even if the Unreal Client is hidden (is covered by other windows or out of the desktop) we can still get the scene image. Hook.dll (available from the tools area of the file release downloads) is the library provided by USARSim that captures the Unreal Client picture into a block of shared memory. It must be in the \UT2004\system directory. Details about using Hook.dll can be found in section 14.5.2.
b. Using the image server to get camera pictures
   The Image server is an extra server that runs with the Unreal Client. It uses the method introduced in the previous paragraph to capture pictures and send them out through the network. The pictures can be sent out in raw format or jpeg format. After sending out a picture, the server waits for an acknowledgement from the client and then sends out the next picture at the specified frame rate speed. The steps to start the image server are listed in section 3.2.5. How to communicate with the server and use the pictures are explained in section 14.5.3.

10.14.2 How to configure it

The robot camera's configuration in the USARBot.ini file looks like:

```
[USARBot.RobotCamera]
Weight=0.4
CameraDefFov=0.7854
CameraMinFov=0.3491
CameraMaxFov=2.0944
```

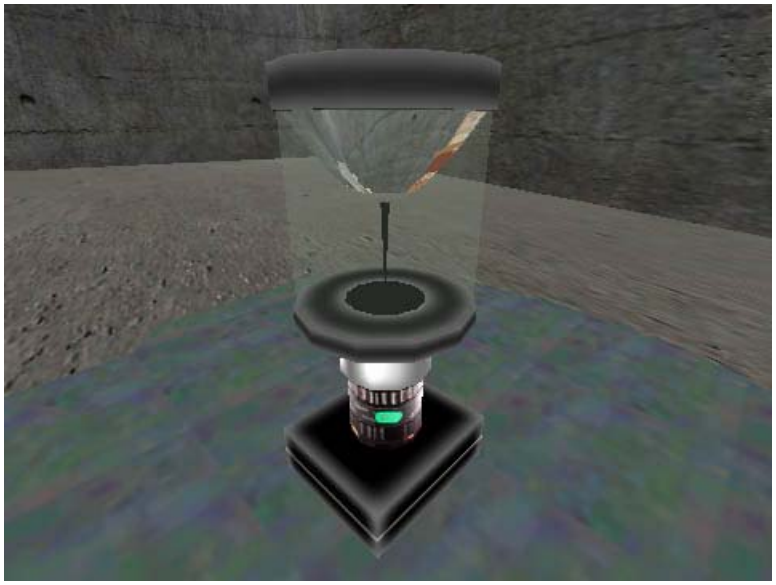
Where

Weight	The weight of the sensor in kg.
CameraDefFov	The camera's default FOV in radians.
CameraMinFov	The minimum FOV in radians.
CameraMaxFov	The maximum FOV in radians. If the CameraMaxFov is equal to the CameraMinFov, the camera is a fixed FOV camera that can't zoom in or out.

10.15 Omnidirectional Camera

10.15.1 How the sensor works

The omnidirectional Camera is a normal UsarSim camera, looking upwards towards a parabolic convex mirror. The camera seems to capture the reflection in the mirror, depicting the complete 360° surroundings with the perspective distortion that is typical for this type of omnidirectional cameras.



The creation of a parabolic convex mirror in UsarSim is not possible in a direct way, because the underlying Unreal engine only supports planar reflecting surfaces. The 'security camera trick' is used to simulate a non-planar mirroring surface 0. The camera looks at a parabolic shaped texture, which is a monitor connected to a security camera hidden behind the texture. The security camera is located at the center of projection of the parabolic texture, and looks through the texture in front of it. The UsarSim camera looks at the other site of the texture, and captures the image displayed on the texture as if it is a reflection. Care has been taken that the image of the security camera correctly distorted on the parabolic texture to get the perspective of an omnidirectional mirror.

10.15.2 How to mount it

To mount an omnidirectional camera you need three elements:

```
MisPkgs=(PkgName="OmniCamera",Location=(Y=0.0,X=0.0,Z=0.0), ...
        ... PkgClass=Class'USARMisPkg.OmniCamera)
Cameras=(ItemClass=class'USARBot.RobotCamera',ItemName="Camera", ...
        ... Parent="OmniCamera_Link1",Position=(Y=0.0,X=0.0,Z=0.09), ...
        ... Direction=(Y=1.5707964,Z=-3.1415927,X=0.0))
CamTexActors=(ItemClass=class'USARBot.USAREmitter',ItemName="OmniCamEmitt
        er",
        ... Parent="OmniCamera_Link1",Position=(Y=0.0,X=0.0,Z=0.0), ...
```

... Direction=(Y=0.0,Z=0.0,X=0.0))

The UsarEmitter class is used for the security camera. The OmniCamera Mission Package consists of the OmniCamBase and the OmniCamMirror models (fixed to each other).
The camera can be mounted on a pillar to give it more height. Use the following configuration to mount the camera on a pillar:

```
MisPkgs=(PkgName="OmniCamPillar",Location=(Y=0.0,X=0.0,Z=0.0), ...
        ... PkgClass=Class'USARMisPkg.OmniCamPillar')
Cameras=(ItemClass=class'USARBot.RobotCamera',ItemName="Camera", ...
        ... Parent="OmniCamPillar_Link4",Position=(Y=0.0,X=0.0,Z=0.09), ...
        ... Direction=(Y=1.5707964,Z=-3.1415927,X=0.0))
CamTexActors=(ItemClass=class'USARBot.USAREmitter',ItemName="OmniCamEmitt
        er",
        ... Parent="OmniCamPillar_Link4",Position=(Y=0.0,X=0.0,Z=0.0), ...
        ... Direction=(Y=0.0,Z=0.0,X=0.0))
```

The pillar consist of a bottom link, multiple center links and a top link. To adjust pillar height, center links can be added or removed in the USARMisPkg.ini file.

10.15.3 How to configure it

The configuration of a standard robot camera in the USARBot.ini is used:

```
[USARBot.RobotCamera]
CameraDefFov=0.7854
CameraMinFov=0.3491
CameraMaxFov=2.0944
```

T. Schmits and A. Visser (2008). An Omnidirectional Camera Simulation for the USARSim World. *Proceedings CD of the 12th RoboCup International Symposium*. Suzhou, China. To be published in the Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin / Heidelberg, Germany.

11 Effecters

11.1 Gripper

11.1.1 How the effecter works

A gripper constructed by a gripper base, a left finger and a right finger. The gripper base is the part where the effect is mounted. The left and right finger can be explicitly defined by the user or automatically found by USARSim through iterating all the parts connected to the gripper base. This effecter controls the left and right finger's joints to open or close the gripper to pick up or drop an object. It currently only manipulates the KNActors, the modified KActors that support network. Gripper

uses the SET command to open or close itself. The Opcode, Params and returned response Status are listed below.

Table 6 Gripper control command

Opcode	Params	Status
Open	The angle between left and right fingers in radians.	"OK": successfully opened the gripper "Failed": can't open the gripper. It may be caused by an invalid command.

11.1.2 How to configure it

The Gripper's configuration in the USARBot.ini file looks like:

```
[USARBot.Gripper]
MaxAngle=1.57
MinAngle=0.0
LeftFinger=LeftFinger
RightFinger=RightFinger
```

Where

MaxAngle	The maximum angle between left and right fingers in radians.
MinAngle	The minimum angle between left and right fingers in radians.
LeftFinger	The left finger's name. If we leave this parameter undefined, USARSim will iterate all the parts connected to the gripper base. The last found left part will be used as the left finger.
RightFinger	The right finger's name. If we leave this parameter undefined, USARSim will iterate all the parts connected to the gripper base. The last found right part will be used as the right finger.

An effecter is very similar to a sensor. We can mount it on the robot and use the SET command to control it. An effecter is more like a dumb sensor that can't send any data out. We introduce all the effecters in this section.

11.2 RFID Releaser

11.2.1 How the effecter works

The RFID releaser drops a RFID tag in the virtual world. The tag's ID is automatically assigned by the releaser in the range of 0-1000 to make sure that all of the IDs are unique. The IDs greater than 1000 are reserved by the system. You can use these reserved IDs for special purposes (preplaced in world maps). The releaser uses the SET command to drop a RFID tag. The Opcode, Params and returned response Status are listed below:

Table 7 RFID releaser control command

Opcode	Params	Status
Release	None	"OK": successfully dropped a tag "Failed": can't drop tags. It may be caused by an invalid command or lack of tags.
TagsRemaining	None	"*int*": number of tags left to dispense.

11.2.2 How to configure it

The RFID releaser's configuration in the USARBot.ini file looks like:

```
[USARBot.RFIDReleaser]
Weight=0,4
NumTags=10
```

Where

NumTags The number of tags that the releaser carries.
Weight The weight of the effecter in kg.

11.3 Roller Table

11.3.1 How the effecter works

A roller table is an effecter that is constructed of a roller table base with a series of controllable roller mounted on the surface of the table. This effecter enables items to be moved back and forth on the table, and is commonly used in manufacturing applications to load and unload packages on and off the table. The speeds of the rollers are synchronously controlled by the effecter, which enables the user to adjust the speed of the rollers to move an item along the surface of the table. The directions of the rollers are determined by the sign of the parameter; i.e. a positive value will spin the rollers the positive direction (along the positive X-axis) and a negative value will cause the rollers to spin in the opposite direction

Table 8 Roller Table opcodes

Opcodes	Parameters	Status
Animate	Float that defines the speed and direction that the rollers will move in radians per second.	"OK": successful "Failed": to set the speed of rollers. Error may be caused by improper arguments.

11.3.2 How to configure it

The Roller Table can be configured in the USARBot.ini file. The Roller Table section of this initialization file looks like:

```
[USARBot.RollerTable]
MaxSpeed=6.0
```

Where

MaxSpeed	Determines the maximum angular velocity that the rollers can spin in either direction.

11.4

11.5 Headlight

The headlight should also be an effecter. Right now, it's NOT an effecter because we simulate it by extending it from an Unreal class, DynamicProjector. In the future, we should try to fix this problem.

12 Robots

All robots in USARSim have a chassis, multiple wheels, sensors and effecters. The robots are configurable. You can specify which sensors/effecters are used and where they are mounted. You also can configure the properties of the robots, such as the battery life and the frequency of data transmission etc. The robots are based on the real robots and they have different capabilities. This section will introduce the robots one by one and explain how to configure them.

We control the robot mobility with one of the many DRIVE commands. For details about the different DRIVE commands, please go back to section 9.4.

12.1 P2AT

12.1.1 Introduction

The P2AT is a 4-wheel drive all-terrain pioneer robot from ActivMedia Robotics, LLC. For more information, visit the ActivMedia's website: http://www.activrobots.com.

In USARSim, we use classname USARBot.P2AT to represent the P2AT.

a) Real P2AT b) Simulated P2AT

Figure 26: P2AT robot

In summary, the P2AT has:

- Four wheels

- Skid-steer

In our simulation, it is equipped with:

- PTZ camera
- Front sonar ring
- Rear sonar ring
- Sick Laser Scanner LMS200
- INS
- Odometry sensor
- RFID sensor

The P2DX specification is as follows:

- Dimensions: Length x Width x Height = 50 cm x 49 cm x 26 cm
- Wheel: Diameter x Width = 22 cm x 7.5 cm
- Weight: 14 kg
- Payload: 40 kg
- Maximum Translate Speed: 700 mm/s
- Maximum Rotating Speed: 140 deg/s

Sonars' positions are:

```
{ X(mm), Y(mm), Theta(deg) } =  {  145,  -130,   -90 },
                                {  185,  -115,   -50 },
                                {  220,   -80,   -30 },
                                {  240,   -25,   -10 },
                                {  240,    25,    10 },
                                {  220,    80,    30 },
                                {  185,   115,    50 },
                                {  145,   130,    90 },
                                { -145,   130,    90 }
                                { -185,   115,   130 },
                                { -220,    80,   150 },
                                { -240,    25,   170 },
                                { -240,   -25,  -170 },
                                { -220,   -80,  -150 },
                                { -185,  -115,  -130 },
                                { -145,  -130,   -90 },
```

12.1.2 Configure it

The whole P2AT robot configuration can be found in the section [USARBot.P2AT] of USARBot.ini file. The following lists the parameters you may want to change. For other parameters please refer section 14.4.2.

```
[USARBot.P2AT]
msgTimer=0.200000
bAbsoluteCamera=true
bMountByUU=True
```

Weight=14
Payload=40
batteryLife=3600
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F1",Position=(X=7.6125,Y=-6.825,Z=0),Direction=(Y=0,Z=-16384,X=0))
...
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F8",Position=(X=7.6125,Y=6.825,Z=0),Direction=(Y=0,Z=16384,X=0))
Cameras=(ItemClass=class'USARBot.RobotCamera',ItemName="Camera",Parent="CameraTilt",Position=(Y=0,X=4.2,Z=-3.36),Direction=(Y=0,Z=0,X=0))

Where

msgTimer	The time interval between sending two consecutive messages.
bAbsoluteCamera	Indicates whether the camera control command uses an absolute value or not.
bMountByUU	Indicates whether we use unreal units in mounting sensors. If it's true, all the sensor/effecter position and direction parameters are in Unreal Unit.
Weight	The weight of the chassis in kg. Similar to sensor's weight, it's just an attribute for description purposes.
Payload	The robot's payload capability in kg.
batteryLife	The life of the battery in seconds.
Sensors	The sensors mounted on the robot. The structure of sensor mounting is:
ItemClass	The sensor class or the type of the sensor.
ItemName	The name assigned to the sensor
Parent	The part the sensor will mount on.
Position	The mounting position relative to parent's geometric center.
Direction	The direction the sensor is facing relative to its parent.
Cameras	The cameras mounted on the robot. It uses the same structure of the sensor. The first camera is the main camera. And you use The CAMERA command to control it. For other cameras, please use SET and MISPKG command to control its FOV and direction.

Note: When you control the camera, make sure bAbsoluteCamera is set to the correct value.

12.2 StereoP2AT

12.2.1 Introduction

The StereoP2AT is a modification of the P2AT. The specifications are the same as that of the P2AT except that the StereoP2AT adds stereo vision.

In USARSim, we use classname USARBot.StereoP2AT to represent this robot.

12.2.2 Configure it

It's the same as P2AT.

12.3 P2DX

12.3.1 Introduction

The P2DX is the 2-wheel drive pioneer robot from ActivMedia Robotics, LLC. For more information please visit ActivMedia's website: http://www.activrobots.com.

In USARSim, we use classname USARBot.P2DX to represent the P2DX.

a) Real P2DX

b) Simulated P2DX

Figure 27: P2DX robot

In summary, the P2DX has:

- Two wheels
- Differential steering

In our simulation, it is equipped with:

- PTZ camera
- Front sonar ring
- Sick Laser Scanner LMS200
- IMU sensor
- Odometry sensor
- Encoders

The P2DX specification is as follows:

- Dimension: Length x Width x Height = 44 cm x 38 cm x 22 cm
- Wheel: Diameter x Width = 16.5 cm x 3.7 cm
- Weight: 9kg
- Payload: 20 kg
- Maximum translate speed: 1800 mm/s
- Maximum rotating speed: 300 deg/s

Sonars' positions are:

{ X(mm), Y(mm), Theta(deg) } = { 155, -115, -50 },
{ 155, -115, -50 },
{ 190, -80, -30 },
{ 210, -25, -10 },
{ 210, 25, 10 },
{ 190, 80, 30 },
{ 155, 115, 50 },
{ 115, 130, 90 }

12.3.2 Configure it

It's the same as P2AT.

12.4 ATRVJr

12.4.1 Introduction

The ATRV-Jr is a 4-wheel drive outdoor all terrain robot vehicle developed by iRobot.

In USARSim, we use classname USARBot.ATRVJr to represent the ATRV-Jr.

a) Real ATRVJr b) Simulated ATRVJr

Figure 28: ATRV-Jr robot

In summary, the ATRV-Jr has:

- Four wheels
- Differential steering

In our simulation, it is equipped with
- PTZ camera
- 17 Sonars (5 front, 10 side, 2 rear)
- Sick Laser Scanner LMS200

The ATRV-Jr specification is as follows:
- Dimension: Length x Width x Height = 77.5 cm x 62.2 cm x 55 cm
- Wheel: Diameter x Width = 33 cm x 10 cm (guessed data)
- Weight: 50 kg
- Payload: 25 kg
- Maximum translate speed: 1000 mm/s
- Maximum rotating Speed: 120 deg/s

Sonars' position are:

```
{   X(mm),   Y(mm), Theta(deg) } = {  334.95, -104.39,  -30   },
                                   {  340.41,  -49.91,  -15   },
                                   {  347.06,       0,    0   },
                                   {  340.41,   49.91,   15   },
                                   {  334.95,  104.39,   30   },
                                   {  230.23,     175,   45   },
                                   {  172.49,   178.6,   60   },
                                   {   117.2,   181.1,   75   },
                                   {   72.26,   181.1,   90   },
                                   { -295.17,   181.1,   90   }, (guessed data)
                                   { -347.06,  150.36,  180   }, (guessed data)
                                   { -347.06, -150.36,  180   }, (guessed data)
                                   { -295.17,  -181.1,  -90   }, (guessed data)
                                   {   72.26,  -181.1,  -90   },
                                   {   117.2,  -181.1,  -75   },
                                   {  172.49,  -178.6,  -60   },
                                   {  230.23,    -175,  -45   }
```

12.4.2 Configure it

It's the same as P2AT.

12.5 HMMWV (Hummer)

12.5.1 Introduction

The HMMWV is a High Mobility Multipurpose Wheeled Vehicle built by the National Institute of Standards and Technology. The HMMWV is a test bed vehicle used to test, evaluate, and demonstrate advanced mobility technology at test facilities and on real roads while performing transportation specific operations.

In USARSim, we use classname USARBot.Hummer to represent the HMMWV.

a) Real HMMWV b) Simulated HMMWV

Figure 29: HMMWV Vehicle

In summary, the HMMWV has:

- Four drive wheels.
- Single Ackerman steering (front two wheels are Ackerman steered).

In our simulation, it is equipped with:

- A pan-tilt camera that can take a 360 degree panorama
- A Sick Laser Scanner LMS200

The HMMWV specification is as follows:

- Dimension: Length x Width x Height = 3.686m x 1.799m x 2.059m
- Wheel Radius = 0.3727m
- Maximum translate speed: 134.172 kph, 83.37mp
- Maximum wheel spin speed: 100 rad/s

12.5.2 Configure it

It's the same as P2AT.

12.6 SnowStorm

12.6.1 Introduction

SnowStorm is the vehicle used by the University of British Columbia (http://www.ubcthunderbird.com) for the DARPA Grand Challenge. It is a 1991 Jeep Cherokee Laredo modified for autonomous control.

In USARSim, we use classname USARBot.Snowstorm to represent this vehicle.

a) Real SnowStorm b) Simulated SnowStorm

Figure 30: SnowStorm

In summary, SnowSorm has:

- Two wheel or four wheel drive.
- Single Ackerman steering (front two wheels are Ackerman steered).
- GPS
- Bumblebee stereo camera
- A 3D Sick Laser Scanner

The SnowStorm specification is as follows:

- Dimension: Length x Width x Height = 4.239m x 1.720m x 1.621m

12.6.2 Configure it

It's the same as P2AT or Hummer.

12.7 Sedan

12.7.1 Introduction

The Sedan is an Ackerman steered vehicle based on the Nissan Primera/ Infinity G20. It was added to USARSim as an additional vehicle for outdoor scenes.

In USARSim, we use classname USARBot.Sedan to represent this vehicle.

a) Real Sedan

b) Simulated Sedan

Figure 31: Sedan Vehicle

In summary, the Sedan has:

- Four wheels. The two front wheels are powered.
- Single Ackerman steering (front two wheels are Ackerman steered).

The Sedan specification is as follows:

- Dimension: Length x Width x Height = 4.2779m x 1.7883m x 1.3238m
- Wheel: Radius = 0.2820
- Maximum translate speed: 134.172 kph, 83.37mph
- Maximum wheel spin speed: 100 rad/s

12.7.2 Configure it

It's the same as P2AT.

12.8 Cooper

12.8.1 Introduction

The Cooper is an Ackerman steered vehicle based on the Mini Cooper. It was added to USARSim as an additional vehicle for outdoor scenes.

In USARSim, we use classname USARBot.Cooper to represent this vehicle.

a) Real Cooper

b) Simulated Cooper

Figure 32: Cooper Vehicle

In summary, the Cooper has:

- Four wheels. The two front wheels are powered.
- Single Ackerman steering (front two wheels are Ackerman steered).

The Cooper specification is as follows:

- Maximum translate speed: 134.172 kph, 83.37mph
- Maximum wheel spin speed: 100 rad/s

12.8.2 Configure it

It's the same as P2AT.

12.9 Submarine

12.9.1 Introduction

The submarine is an underwater robot. It is a standard submarine, which was not based on any particular real models.

In USARSim, we use classname USARBot.Submarine to represent this vehicle.

Figure 33: Submarine in Simulation

In summary, the submarine has:

- A propeller, a rudder, and a stern plane

In our simulation, the submarine has:

- A pan-tilt camera that can take a 360 degree panorama
- A sonar sensor, located on the belly of the submarine, pointing down

The submarine specification is as follows:

- Dimension: Length x Width x Height = 6.5364m x 1.1428m x 1.9929m
- Rudder Area: 0.3378m^2
- Maximum Rudder Angle: 0.4363 radians
- Stern Plane Area: 0.2152m^2
- Maximum Stern Plane Angle: 0.4363 radians
- Propeller's Pitch: 0.3048m
- Maximum Propeller Spin Speed: 6.28 rad/s

12.9.2 Configure it

It's the same as P2AT.

12.10 Tarantula

12.10.1 Introduction

The Tarantula is a toy-based robot which was first turned into a robot platform named "Lurker" by the team "Rescue Robots Freiburg". They used the modified version in the Rescue Robot League during the RoboCup 2005 competition. The Tarantula model, which is now part of the USARSim package, was originally developed at the University of Freiburg and has been further improved and merged into USARSim by the University of Pittsburgh.

In USARSim, we use classname USARBot.Tarantula to represent this robot.

a) Real Tarantula b) Simulated Tarantula

Figure 34 : Tarantula Model in USARSim

In summary, the tarantula has:

- Four flippers, each of which can be controlled independently

The tarantula specification is as follows:

- Dimension: Length x Width x Height = 0.6778m x 0.444m x 0.1406m
- Maximum wheel spin speed: 3.1416 rad/sec

The most significant difference of the Tarantula as compared to conventional robot platforms is the flippers, which endow the robot with considerable mobility. The flippers' control commands are

- Front flipper: MULTIDRIVE {FRFlipper *float*} {FLflipper *float*}
- Rear flipper: MULTIDRIVE {RRFlipper *float*} {RLFlipper *float*}

For example, "MULTIDRIVE {FRFlipper 1.57} {FLFlipper 1.57}" will flip the front flippers towards 90 degree.

Sensors, cameras, and effectors can be attached to the Tarantula model in the same way as to any other robot, as for example a P2AT.

12.10.2Configuration

In addition to standard configuration settings, one can define the accuracy of the track-simulation of the Tarantula model. The tracks of the Tarantula model are simulated by a series of tires, which is computationally complex on the Unreal engine. When the number of tires is large, the simulation might become unstable due to the computational complexity. Hence, the model enables one to adjust the number of tires utilized for the simulation of one track according to your specific needs. Note that with increasing number of tires, the quality of the simulation increases, however, the computational complexity increases as well. The adjustment can be done in the *USARBot.ini* file, in the sections *USARBot.TarantulaFrontTrack* and *USARBot.TarantulaRearTrack*, respectively.

12.11 Zerg

12.11.1 Introduction

The Zerg robot is a 4WD (4-wheeled robot), which has been developed and deployed by the team “Rescue Robots Freiburg” during RobotCup05. The simulation model was developed at University of Freiburg and has been further improved and merged into USARSim by the University of Pittsburgh. The character of this model is “simple”.

In USARSim, we use classname USARBot.Zerg to represent this robot.

a) Real Zerg b) Simulated Zerg

Figure 35: Zerg model

In summary, the Zerg has:

- Four wheels
- Differential steering

In our simulation, it is equipped with

- PTZ camera

The Zerg specification is as follows:

- Dimension: Length x Width x Height = 31.12cm x 41.54cm x 12.11cm
- Wheel radius = 6.06cm

- Maximum wheel spin speed: 6.2832 rad/sec

12.11.2 Configuration

See P2AT.

12.12 Talon

12.12.1 Introduction

The Talon is a lightweight tracked vehicle built by Foster-Miller (www.foster-miller.com) for missions ranging from reconnaissance and weapons delivery to rescue.

In USARSim, we use classname USARBot.Talon to represent this robot.

a) Real Talon b) Simulated Talon

Figure 31: Talon Robot

In summary, a Talon has:

- Two tracks
- One arm with two joints
- Weight: 34 kg
- Payload: 45 kg

In USARSim, the Talon is equipped with

- 4 fixed color cameras
- One gripper with two fingers
- One odometry sensor
- One INU sensor

The Talon specification is as follows:

- Dimension: Length x Width x Height = 91.17cm x 59.03cm x 36.54cm
- Wheel radius = 0.28cm
- Maximum wheel spin speed: 6.43 rad/sec

12.12.2 Configure it

It's the same as P2AT.

12.13 QRIO

12.13.1 Introduction

The QRIO is an artificially intelligent humanoid designed and manufactured by Sony. Although the QRIO was to be marketed and sold by Sony as an "entertainement robot", development stopped in January 2006.

In USARSim, we use classname USARBot.QRIO to represent this robot.

a) Real QRIO b) Simulated QRIO

Figure 36: QRIO Robot

In summary, the QRIO has:

- Two Legs, each of which has 6 joints
- Two Arms, each of which has 3 joints
- Head made of 2 Joints (Pan and Tilt)
- Rotation Body made of two joints (Pan and Tilt)
- Head Camera

12.13.2 Configure it

It's the same as the P2AT.

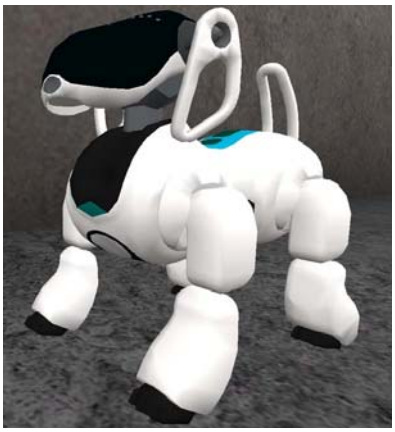
12.14 ERS

12.14.1 Introduction

The ERS (also known as AIBO) is an artificially intelligent robotic pet designed and manufactured by Sony. The ERS is able to walk, see its environment via camera, and recognize spoken commands. ERS robots are used in the Four-Legged League of RoboCup soccer.

In USARSim, we use classname USARBot.ERS to represent this robot.

a) Real ERS



b) Simulated ERS

Figure 37: ERS Robot

In summary, the ERS has:

- Four legs, each of which has three joints
- Paw sensors in each Leg
- Head made of three joints
- Head Camera
- IR Distance Sensors (Near and Far)
- Weight: 1.6 kg

The ERS specification is as follows:

- Dimensions: 31.9 cm x 18.0 cm x 27.8 cm

12.14.2 Configure it

It's the same as the P2AT.

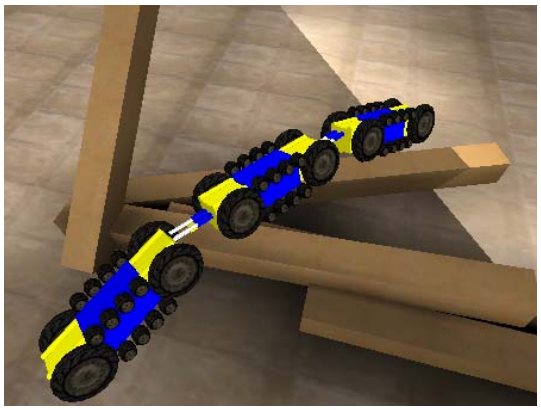
12.15 Soryu

12.15.1 Introduction

The IRS Soryu is a serpentine robot used for search and rescue developed by Tokyo Institute of technology and International Rescue System Institute. It has the ability to easily intrude narrow spaces as well as traverse rough terrain.

In USARSim, we use classname USARBot.Soryu to represent this vehicle.

a) Real Soryu



b) Simulated Soryu

Figure 38: Soryu Robot

In summary, the Soryu has:

- Three cars, each of which has two tracks
- Two joints between each car (pan and tilt)
- CCD and IR cameras
- Ability to recover from lying on its side
- Weight: 10 kg

The ERS specification is as follows:

- Dimensions: 121 cm x 12.2 cm x 14.5 cm
- Maximum translate speed: 370 mm/s
- Maximum step height: 483 mm

12.15.2 Configure it

It's the same as the P2AT.

12.16 Kurt2D

12.16.1 Introduction

The Kurt2D is a high speed indoor robot.

In USARSim, we use classname USARBot.Kurt2D to represent this robot.

a) Real Kurt2D b) Simulated Kurt2D

Figure 395: Kurt2D Robot.

In summary, the Kurt2D has:

- Six Wheels
- Differential steering
- Weight: 35 kg

In USARSim, the Kurt2D is equipped with

- One fixed color camera
- One 2D SICK laser scanner
- Two encoder sensors
- One sonar sensor
- Seven IR sensors

The Kurt2D specification is as follows:

- Dimensions: 45cm x 29cm x 35cm
- Maximum translate speed: 4 m/s

12.16.2 Configure it

It's the same as the P2AT.

12.17 Kurt3D

12.17.1 Introduction

KURT3D is a mobile robot of type KURT2D that is equipped with a 3D laser scanner. Given the right control and sensor data processing software, it is generally able to build 3D models (data point clouds) of its working environment autonomously.

In USARSim, we use classname USARBot.Kurt3D to represent this robot.

a) Real Kurt3D b) Simulated Kurt3D

Figure 406: Kurt3D Robot.

In USARSim, the Kurt2D is equipped with

- Two color cameras
- One 2D SICK laser scanner
- Two encoder sensors
- One sonar sensor
- Seven IR sensors

12.17.2 Configure it

It's the same as the P2AT.
To perform a 3D scan you have to manually set a rotation command to the 'ScannerSides'.

```
DRIVE {Name ScannerSides} {Value ROTATION_VALUE}
```

If you send this command often enough to GameBots (depends on the resolution of your 3D scan) you get a 3D scan of the environment. This is not as easy as the USARSim RangeScanner3D, but using this procedure we can treat the simulated scanner like the real scanner.

12.18 Lisa

12.18.1 Introduction

The LISA robot is developed in a BMBF funded project during the next 2 years. It is an assistance system in a life science environment. The physical robot is not built yet. We use USARSim to explore the Omni drive behaviour of the robot. The robot has two big Omni drive steered wheels on two opposite corners and two small passive wheels on the other corners. It is equipped with encoder sensors on the steered wheels.

In USARSim, we use classname USARBot.Lisa to represent this robot.

a) Real LISA robot b) Simulated LISA robot

Figure 417: Lisa Robot.

12.18.2 Configure it

It's the same as the P2AT.

The LISA script file looks like this. Using the Omni drive DRIVE command we are able to drive wheel 0 and 1 in every direction with every speed (taking the MaxSteerAngle and MaxSpeed limits into account).

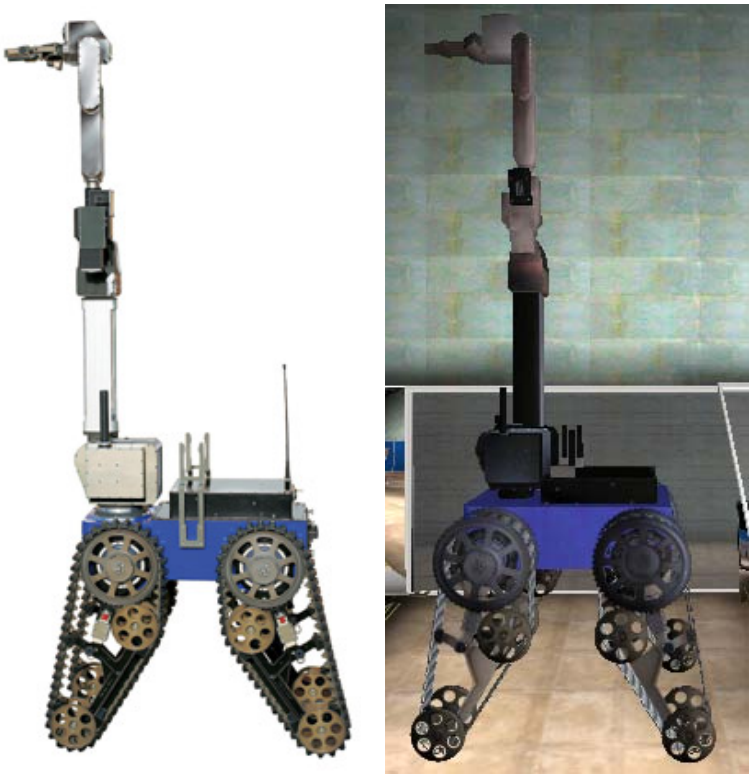
```
Wheels(0)=(Number=0,PowerType=Holo_Powered,SteerType=Holo_Steered,MaxSteerAngle=3.14);
Wheels(1)=(Number=1,PowerType=Holo_Powered,SteerType=Holo_Steered,MaxSteerAngle=3.14);
Wheels(2)=(Number=2,PowerType=Not_Powered);
Wheels(3)=(Number=3,PowerType=Not_Powered);
```

12.19 TeleMax

12.19.1 Introduction

The TeleMax is an EOD (explosive ordnance disposal) robot built by Rheinmetall Defence (http://www.rheinmetall-defence.com) that intends to work in narrow spaces.

In USARSim, we use classname USARBot.TeleMax to represent this robot.



a) Real TeleMax b) Simulated Telemax

Figure 428: Telemax Robot

In summary, a TeleMax has:

- Four tracks
- One 7-axis manipulator with turret and linear axis

The TeleMax specification is as follows:

- Wheel: Diameter x Width = 38.1 cm x 9.5 cm
- Size: Length x Width x Height = 80 cm x 40 cm x 75 cm (Stowed Position)
- Vertical reach: 235 cm
- Horizontal reach front: 120 cm
- Maximum Speed: 1.3 m/s
- Operation Time: 2 hours

In our simulation, it is equipped with

- Fore fixed color camera
- One gripper with two fingers
- One Odometry sensor
- One INU sensor

To control the arm in USARSim, we use the following commands:
Arm control:

```
MISPKG {Name TeleMaxArm} {Link 1} {Value x} {Link 2} {Value y} {Link
3} {Value z} {Link 4} {Value r} {Link s} {Value t} {Link 6} {Value
u} {Link 7} {Value v}
```

where *x*, *y*, *z*, *r*, *s*, *t*, *u*, *v* are either the rotation angles in radians (for revolute joints) or translation distance in meters (for prismatic joints). From the turret to the arm terminal (the gripper), Link 1-7 correspond to the turret pan axis, upper arm tilt axis, telescope, lower arm tilt axis, lower arm turn axis, gripper tilt axis, and gripper turn axis, respectively. For example, to set the arm in initial pose, we use:
MISPKG {Name TeleMaxArm} {Link 1} {Value 0} {Link 2} {Value 2} {Link 3} {Value 0} {Link 4} {Value -2} {Link 5} {Value 0} {Link 6} {Value 0} {Link 7} {Value 0}

Flipper control:
`MULTIDRIVE {FRFlipper x} {FLFlipper y} {RRFlipper z} {RLFlipper r}`

where *x*, *y*, *z*, *r* are the flipper angles in radians. The four parameters control the front left, front right, rear left and rear right track separately. For example, to stow the tracks to lift or lower the chassis, we use
MULTIDRIVE {FRFlipper -1.5} {FLFlipper -1.5} {RRFlipper 1.5} {RLFlipper 1.5}
or.
MULTIDRIVE {FRFlipper 1.5} {FLFlipper 1.5} {RRFlipper -1.5} {RLFlipper -1.5}

Gripper control:
`SET {Type Gripper} {Name Gripper} {Opcode Open} {Params x}`

where *x* is the desired open angle in radians. For example to open and close the gripper, we use
SET {Type Gripper} {Name Gripper} {Opcode Open} {Params 1}
and
SET {Type Gripper} {Name Gripper} {Opcode Open} {Params 0}

12.19.2 Configure it

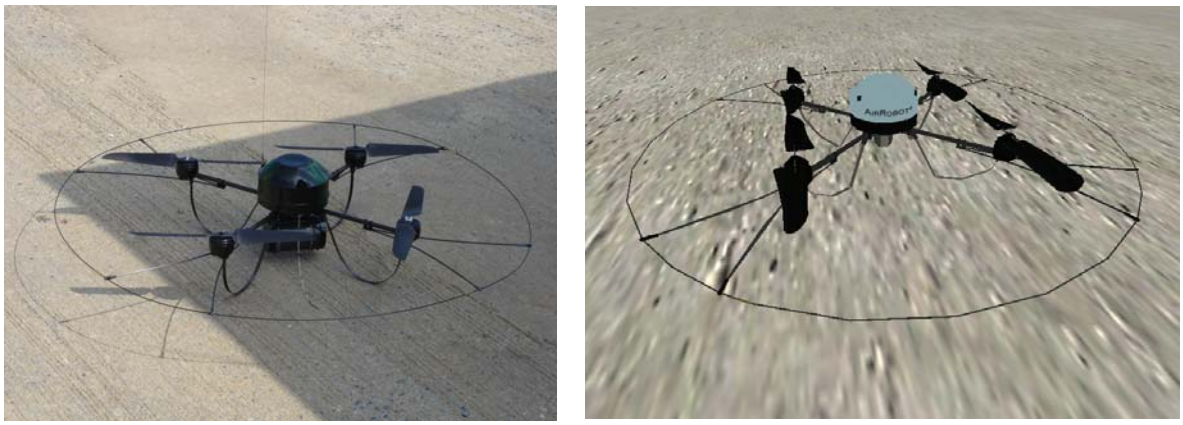
It's the same as P2AT.

12.20 AirRobot

12.20.1 Introduction

The AirRobot is a four-rotor electrical helicopter with flight stabilization control produced by AirRobot Co. (http://www.airrobot.com/englisch/index.php). The AirRobot serves many purposes including, but not limited to, exploration, observation, documentation, and measurement.

In USARSim, we use classname USARBot.AirRobot to represent this robot.



a) Real AirRobot b) Simulated AirRobot

Figure 439: AirRobot

In summary, an AirRobot has:

- Four propellers
- One color camera that can tilt
- Weight: 1 kg
- Payload: 200 g

In USARSim, the AirRobot is equipped with

- One “tilt-only” color camera

The AirRobot specification is as follows:

- Dimension: Length x Width x Height = 0.999m x 0.999m x 0.194m
- Maximum altitude velocity: 5 m/s
- Maximum linear velocity: 5 m/s
- Maximum lateral velocity: 5 m/s
- Maximum rotational velocity: 1.5708 rad/s

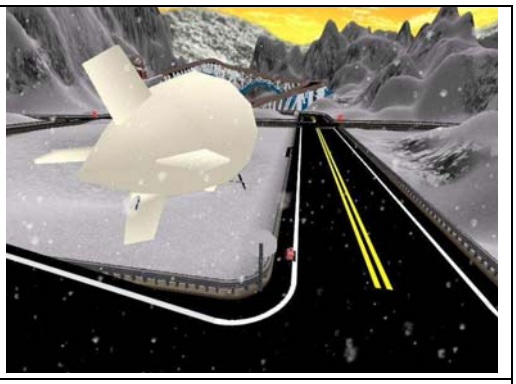
12.20.2 Configure it

It’s the same as P2AT.

12.21 Passarola

12.21.1 Introduction

Passarola (the aerial blimp/zeppelin robot), Autonomous blimp for rescue missions (http://rescue.isr.ist.utl.pt/), from IST - Instituto Superior Técnico (Higher Technical Institute) – Portugal

	
a) Real Passarola	b) Simulated Passarola

In USARSim, we use classname USARBot.Passarola to represent this robot.
In summary, Passarola has:
• Three propellers (two for the linear movement, vectoring 180 degrees (-90º to 90º), and one on the tail for angular movements)
• One color camera that can tilt
• Weight: 3 kg
• Payload: 2 Kg

In USARSim, the Passarola is equipped with
• One “tilt-only” color camera

The Passarola specification is as follows:
• Dimension: Length x Width x Height = 4.0m x 2.0m x 2.0m
• Maximum altitude velocity: 2 m/s
• Maximum linear velocity: 2 m/s
• Maximum rotational velocity: 1 rad/s

12.21.2 Configure it

It’s the same as P2AT.

12.21.3 Extended USARSim command for Passarola robot

DRIVE {XZAngle *float*}{ThrustPropeller *float* }{TailPropeller *float* }
Where:
{ XZAngle *float*}
‘*float*’ is the rotation angle of the support thrust motors bars, that make possible change the altitude of the robot (i.e up/down). If we use normalized values, the value range is -100 to 100 and corresponds to the bar’s minimum and maximum rotation angle, respectively. Otherwise, the value is the absolute rotation angle, in radians per second.

{ ThrustPropeller *float*}
‘*float*’ is the module of the velocity vector to be applied by the front thrusters, to move the robot in the X0Z plane (i.e forward/backward and up/down as the value of XZAngle). If we use normalized values, the value range is -100 to 100 and corresponds to the robot’s minimum and maximum velocity, respectively. Otherwise, the value is the absolute linear velocity, in meters per second.

{ TailPropeller *float*}
‘*float*’ is the rotational velocity (i.e left/right). If we use normalized values, the value range is -100 to 100 and corresponds to the robot’s minimum and maximum rotational velocity, respectively. Otherwise, the value is the absolute rotational velocity, in meters per second.

{Normalized *bool*}
Indicates whether we are using normalized values or not. The default value is ‘False’ which means absolute values are used.

Example: DRIVE { XZAngle 0.5} will rotate the support motor bars 0.5 radians
DRIVE { ThrustPropeller 1} will thrust the robot at a rate of 1 meters per second.
DRIVE { TailPropeller -0.3} will rotate the robot to the right at a rate of 0.3 radians per second.

12.22 Rugbot

The Rugbot Robot is a compact, lightweight, rugged vehicle developed at Jacobs University Bremen. It is ideal for both autonomous and teleoperated mode of operation.

In USARSim, we use classname USARBot.Rugbot to represent this robot.

a) Real Rugbot b) Simulated Rugbot

Figure : Rugbot Robot

In summary, a Rugbot has:

- Two tracks
- Weight: 34 kg

• Payload: 45 kg

In USARSim, the Rugbot is equipped with

• One Panasonic Pan/Tilt camera
• One SICK LMS sensor
• One odometry sensor
• One INU sensor

The Rugbot specification is as follows:

• Dimension: Length x Width x Height = 38.5cm x 25.5cm x 27.0cm
• Wheel radius = 0.125cm
• Maximum wheel spin speed: 6.0 rad/sec

Configure it

It’s the same as P2AT

12.23 Kenaf

12.23.1 Introduction

The Kenaf is a 6-track mobile robot platform which is designed for drastic performance gain of uneven terrain mobility. In fact, the Kenaf wins championship in RoboCup Rescue 2007 Mobility Challenge. Each track of the robot grasps the environment firmly and makes the robot to be grounded to the environment, and the robot gets over the unknown rough terrains. This robot will be commercially available.

In USARSim, we use classname, "USARBot.kenaf," to represent this robot.

In summary, the Kenaf has:
. Two full body tracks to avoid stacking on a pole in step field
. Four flippers, each of which can be controlled independently
. Powered by 6 x 50W brush-less DC motors

The Kenaf's specification is as follows:
. Dimensions :
  Minimum Length = 575 [mm]
  Maximum Length = 937 [mm] (include flipper length)
  Width = 429 [mm]
  Height = 259 [mm] (without sensors and cameras)
. Weight approx. 20 [kg]

In our simulation, it is equipped with
. One PTZ camera
. One Bird's-eye view camera
. One 2D Hokuyo laser scanner
. Six encoder sensors

Sensors and effectors can be attached to the Kenaf model in the same way as to any other robot, as for example a P2AT.

To control the arm in USARSim, we use the following commands:

Flipper control:
MULTIDRIVE {FRFlipper x} {FLFlipper y} {RRFlipper z} {RLFlipper r}
where x, y, z, r are the flipper angles in radians. The four parameters control the front left, front right, rear left and rear right track separately. For example, to stow the tracks to lift or lower the chassis, we use
MULTIDRIVE {FRFlipper -1.5} {FLFlipper -1.5} {RRFlipper 1.5} {RLFlipper 1.5}
or.
MULTIDRIVE {FRFlipper 1.5} {FLFlipper 1.5} {RRFlipper -1.5} {RLFlipper -1.5}

Camera control:
MULTIDRIVE {kenafCameraPan x}{kenafCameraTilt y}
where x, y are the pan, tilt angles in radians. For example, to pan the camera to left or right side 1.0 radians.
MULTIDRIVE {kenafCameraPan -1.0}{kenafCameraTilt 0.0}
or.
MULTIDRIVE {kenafCameraPan 1.0}{kenafCameraTilt 0.0}

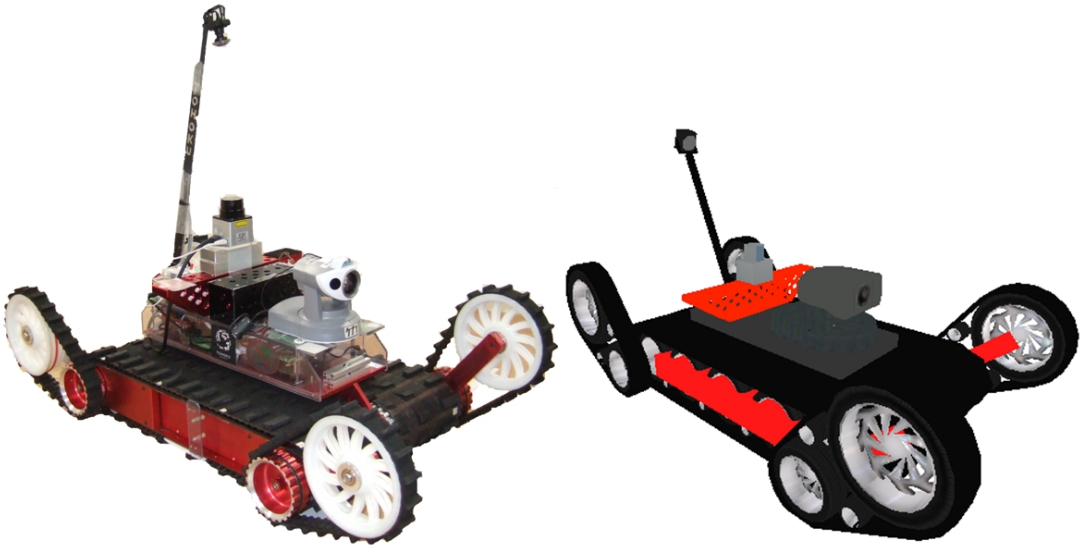


Figure 44: Real and simulated Kenaf

12.23.2 Configure it

It's the same as the P2AT.

13 Controller

13.1 MOAST

A description of the low-level connection (the architectural servo and prim levels) from MOAST to USARSim is provided here. For a more complete description of the MOAST system, please refer to the MOAST manual (http://moast.sourceforge.net/).

A description of how to install and bring a robot into the environment is presented in Section 5.2.2. It is assumed that you have successfully installed MOAST, started the Unreal Server, and have run the "run" script (located in the bin directory) with SECT, VEH, and AM set to 'no' and PRIM and USARSIM set to 'yes'.

NOTE: You must set the HOST_NAME in the file moast.ini to point to the machine that is running the Unreal Server.

You should now see the specified robot type at the specified start location in your Unreal Client window (if you are running the client!). The type and location are specified under the block in the moast.ini file that pertains to the arena currently under play by the Unreal Server that was connected to.

Under the MOAST framework, all of the sensor data and robot commands are delivered over Neutral Messaging Language (NML) buffers. There are three general techniques for a user or program to interface to these buffers. The first is to directly connect to the appropriate NML buffer (please see the NML tutorial located at http://www.isd.mel.nist.gov/projects/rcslib/), the second is to utilize one of the provided shells, and the third is through the RCS Diagnostics tool.

An example of directly connecting to NML buffers is provided by the nmlPrint program located in the *moastBaseDir/devel/src/tools* directory. This program prints out the content of a selected buffer to the screen. By piping its output to other programs (such as gnuplot), sensor displays and graphs are possible. Figure shows the result of running the command spPlot | gnuplot where spplot is a shell script located in MOAST's bin directory that runs nmlPrint on the buffer servoSPLinescan1. This buffer contains the data received from the Sick LMS sensor on robot 1 and (like all NML buffers) is available to any computer that has a network connection to the system running the MOAST/USARSim middle ware. The actual location of the buffer is invisible to the application that is connecting to the buffer.

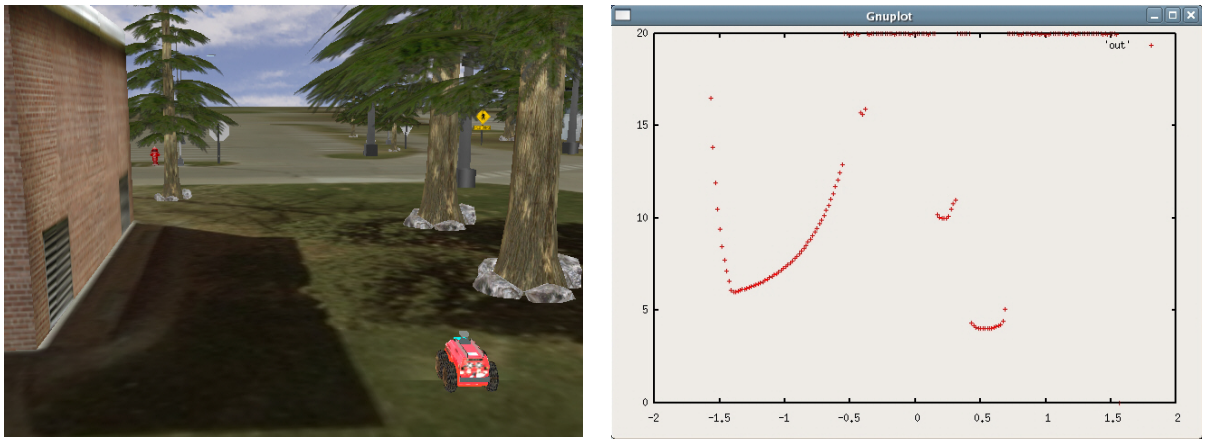


Figure 40: Image from the unreal Client and resulting graphical display of Sick LMS readings provided by nmlPrint and gnuplot. The building and trees are clearly visible in the plot.

The second technique is to utilize the command shell that is started by the run script. A command shell exists for each level of the hierarchy, and the run script automatically starts the highest level command shell that is appropriate. In our case, the prim shell will be started and the window where the run script was started should now be displaying a '>' prompt that lets you know you are in the prim shell. Other command shells may be opened by hand in additional windows. Pressing a carriage return (<CR>) will print the robot's current status. Entering ?<CR> will provide a list of available commands. The information provided by the status message is as follows:

command_type:	The name of the executing command.
echo_serial_number:	The serial number of the executing command.
status:	The system status.
state:	Most RCS controllers run state machines. This is the id of the current state.
line:	The line in the source code of the state table that is being executed.
source_line:	The location of the beginning of the current state in the source file.
source_file:	The name of the source code file.
heartbeat:	A constantly increasing number that allows you to know the system is functioning.
pathIndex:	If the system is following a path, this indicates which point in the path is being servoed to.
tranAbs:	The x, y, z location in meters of the vehicle in absolute coordinates
rpyAbs:	The roll, pitch, and yaw in radians of the vehicle in absolute coordinates
tranRel:	The x, y, z location in meters of the vehicle in vehicle relative coordinates.
rpyRel:	The roll, pitch, and yaw in radians of the vehicle in relative coordinates.

The available commands for the robot at this level of control are:

init:	Initialize the system.
halt:	Provide an orderly, safe, and recoverable halt of all systems.
abort:	Provide an immediate and safe halt of all systems. Some systems may need to be reset to recover from an abort.
shutdown:	Turn off (power down) the systems.
arc <file>:	Drive the arcs given in the file <file>.
wp <file>:	Drive straight line segments between points given in the file <file>.
rotate <theta>:	Rotate the robot to angle theta.
vel <v> <w>:	Drive the vehicle at velocity v with rotational velocity w.
ct <secs>:	Set the system cycle time to <secs>
pars <5, vmax…>:	Configure the vmax, amax, wmax, alphamax, and cut parameters that are used for path following.
debug:	Set the debug output level of the software at this level.

More information on the commands available at this level of the hierarchy as well as the other levels may be found at the MOAST website. It should be noted that the shell programs are provided to demonstrate how to connect to the robot at various levels of control and to provide some simple user debugging.

One of the features of MOAST is that all of the control interfaces are brought out over standardized interfaces with NML. Complete documentation on the available buffers is available on the MOAST website. Programs running on Windows (compiled with Microsoft Visual C++), under cygwin (compiled with GNU tools), and under Linux may all connect simultaneously to these buffers. The contents of all of these buffers may be examined by running the RCS-Diagnostics tool (the third technique). This may be run from the *moast_base_dir*/devel/src/nml directory as either ./moastDiag.csh or ./moastDiag.cygwin.csh depending on your operating system.

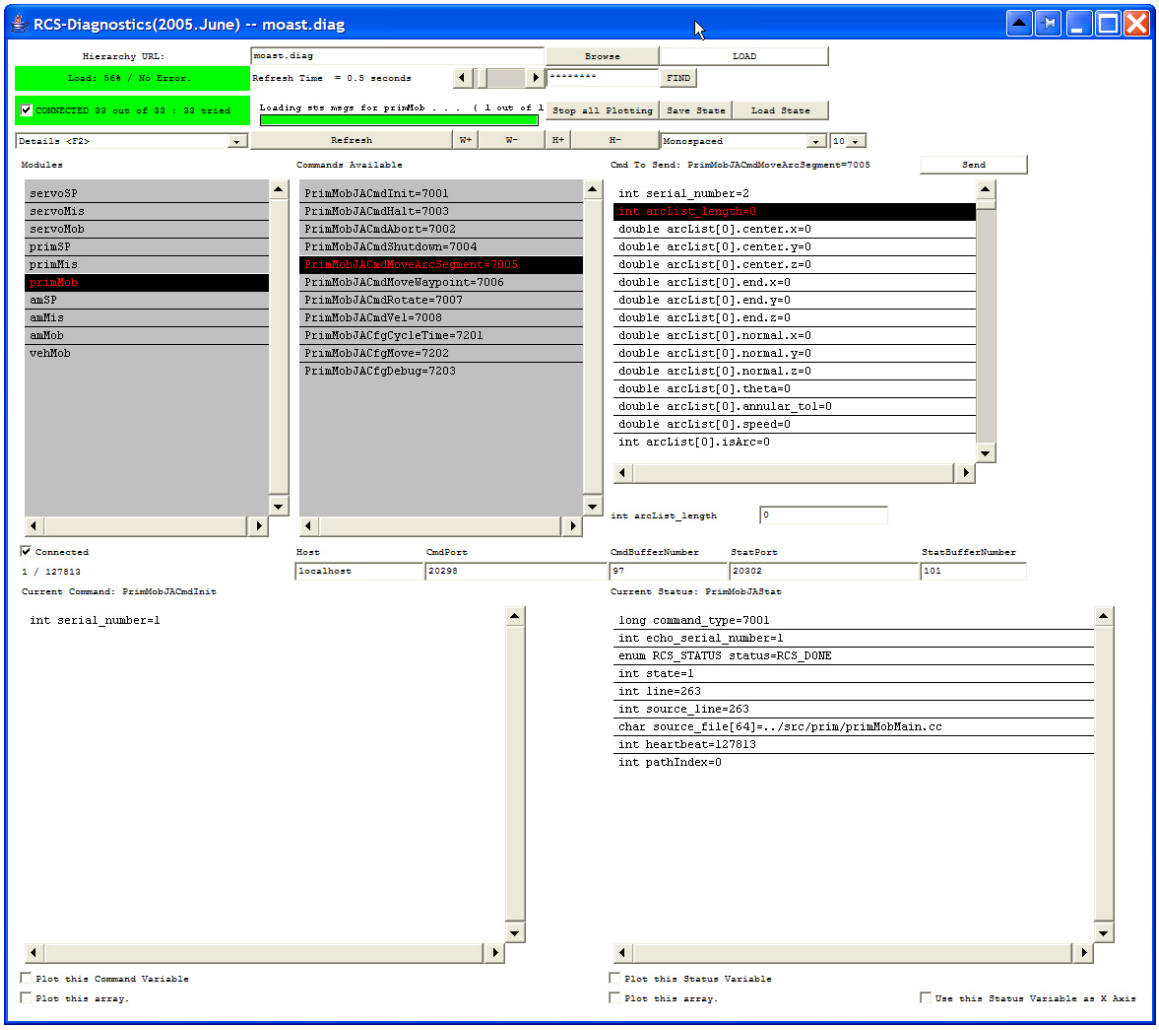


Figure 45: MOAST Diag display

Figure 45 displays the details page of the diagnostic tool. This tool allows you to see the details of the currently executing command for any module and the module's status. In addition, any command may be sent to any module from this interface. This allows for complete unit testing of control code. Once again, much more information is available on the MOAST website.

13.2 Pyro

A complete description of Pyro can be found on the Pyro website: http://pyrorobotics.org/pyro/?page=PyroModulesContents. In this section we only explain the elements that are involved in USARSim.

13.2.1 Simulator and world

The USARSim simulator loader is put into the pyro\plugins\simulators directory. The loader USARSim.py is a Python program that can load the Unreal server and client for the user. It reads the world file to figure out which arena (map) you want. Then, it will start the Unreal server using the appropriate arena (map) in the

Unreal world. After a wait of 5 seconds to load the server, it will launch the Unreal client.

The world files for USARSim are stored in the plugins\worlds\USARSim.py directory (NOTE: here USARSim.py is not a file. It's a directory.). The file follows the INI file format. A world file looks like:

```
[Server]
Path=c:\ut2004
App=ut2004.exe
LoadServer=true
IP=127.0.0.1
Port=3000
Map=DM-USAR_yellow
Location=4.5,1.9,1.8
```

Where:

Path	The install path to UT2004.
App	The application used to load Unreal Client. For UT2004, it's UT2004.exe.
LoadServer	A Boolean variable indicating whether the loader needs to start the Unreal server. If you already started Unreal server or you want to run the Unreal server on another machine, you need to set LoadServer to false. Default value is true.
IP	The IP address of the Unreal server. Default value is 127.0.0.1
Port	The port number of the Gamebots. Default value is 3000. The port number should be the same as the "ListenPort" in the BotAPI.ini file in the Unreal system directory (more details see section 9.1).
Map	The Unreal map you want to load. For yellow, orange and red arenas, they are DM-USAR_yellow, DM-USAR_orange and DM-USAR_red.
Location	The initial position where the robot will be spawned. Please refer Table 2 or the map location files that are bundled with the maps to decide the values you want.

13.2.2 Robots

USARSim robot drivers are written for Pyro. In summary, there are three levels of control provided by the drivers.

The lowest level driver is robots\driver\utbot.py. It communicates with the Unreal server through a TCP/IP socket. The main functions in the driver are

1) Creating a connection with the Unreal server
2) Sending commands to the Unreal sever.
3) Listening and parsing messages from the Unreal server.

In the robots\USARBot directory are the low level drivers. __init__.py is the basic driver that provides the Pyro interface. It lets the Pyro commands and data be understood by USARSim. The P2AT.py, P2DX.py. PER.py etc are the drivers extended from the basic driver. These drivers configure the basic driver according to the individual robot. For example, it configures which sensor is mounted on the robot.

At last, you will find several files in the plugins\robots\USARBot directory. These files are the wrapper to the robot drive. You can directly load these files from the Pyro GUI to add a robot into the USARSim virtual environment.

13.2.3 Services

To help the user to understand the data being reported by the sensors, some services are added to visualize the sensor data. These sensor visualizations are modified from the visualization module of PyPlayer (http://robotics.usc.edu/~boyoon/pyplayer/). To load the services, from the 'Load' menu select 'Services …'. Then go to plugins\services\USARBot directory you can found all the services. The real code for these services is in the robots\USARBot__init__.py file. The supported sensors are:

- Sonar

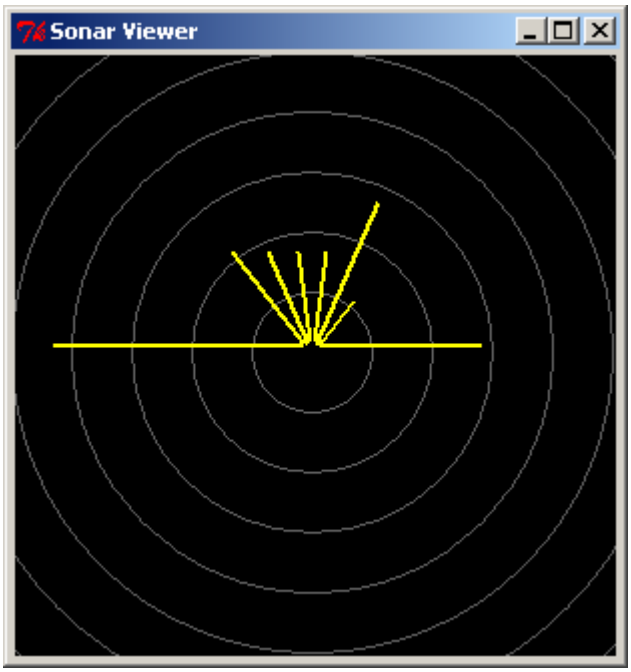


Figure 46: Sonar visualization

- Laser

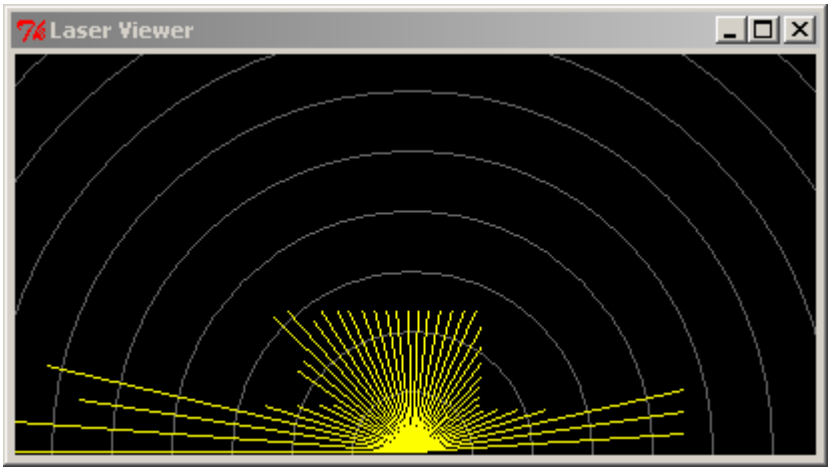


Figure 473: Laser visualization

- PTZ Camera

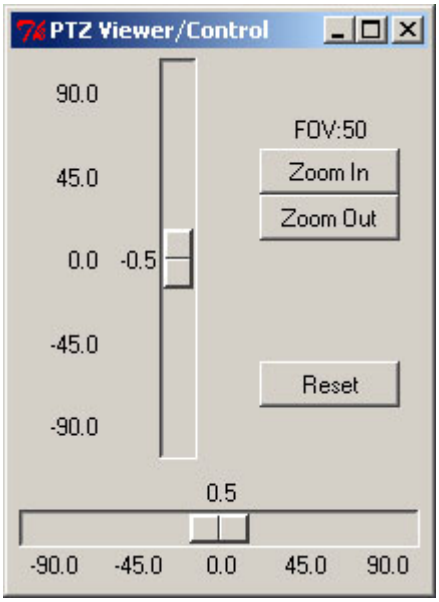


Figure 484: PTZ Camera viewer and controller

13.2.4 Brains

Pyro refers to control programs as “Brains”. Since the USARSim API follows the Pyro interface, the brains of Pyro will work for USAR robots. The tested working brains include Slider.py, Joystick.py, and BBWander.py.

13.3 Player

Player provides a client-server based hardware abstraction layer. Every sensor supports one or multiple sensor interfaces. Therefore it’s not important, if used for example a SICK or a Hokuyo or a simulated laser scanner. This provides the possibility to create high level abstract drivers like localization or path planning

drivers. The following figure shows the UT2004 window and the standard player GUIs playerv and playernav for path planning and sensor data visualisation.

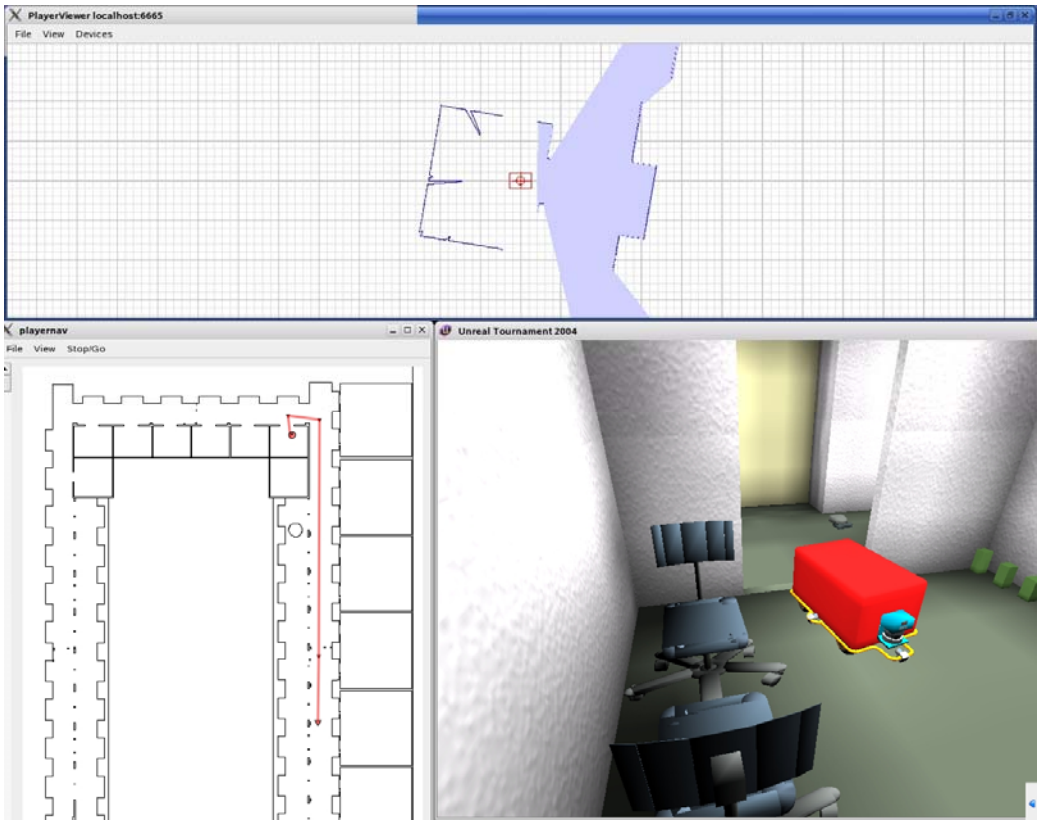


Figure 495: playerv sensor data visualization tool (top), playernav player navigation and path planning tool,(left), the simulated robot in USARSim (right).

In this section, we introduce how to control USARSim robots through Player. At first, we explain how to plug USARSim into Player. And then we explain all the Player drivers added into USARSim. For additional information about Player, please read the Player Wiki: http://playerstage.sourceforge.net/wiki/Main_Page.

13.3.1 Simulation and device configuration

In Player, all the USARSim actuators and sensors are treated the same as the physical devices. The only difference between our USARSim device driver and the physical device driver is that our driver exchanges data with the Unreal server while physical device drivers exchange data with physical devices. Like all the other Player devices, to use USARSim, we need to define the Player configuration file. In the Player configuration file, we need to:

1) Define the USARSim robot

Before we can control a robot in USARSim, we need to spawn it in the virtual world. The USARSim robot definition defines how the robot is added into USARSim. In detail, we define where the Unreal server is, which type of robot is added to the simulation, and where it is spawned. The complete configuration options can be found in section 13.3.2.1.

2) Define USARSim devices

In the definitions, we define the parameters that help Player figure out where and how the devices are connected. The definition is very similar to that of the physical devices. The only exception is that instead of defining the device's connection port, we define the USARSim robot where the device is located. The details about how to configure the USARSim devices is explained in section 13.3.2.2 to section 13.3.2.7.

The following is an example player configuration file:

```
driver
(
 name "us_bot"
 provides ["simulation:0"]
 port 3000
 host "127.0.0.1"
 pos [-5 27 0.5]
 rot [0 0 0]
 bot "USARBot.P2AT"
 botname "robot1"
)

driver
(
  name "us_laser"
  provides ["laser:0"]
  requires ["simulation:0"]
  laser_name "Scanner1"
)
driver
(
  name "us_position"
  provides ["odometry:::position2d:0" ]
  requires ["simulation:0"]
)
driver
(
  name "us_sonar"
  provides ["sonar:0" ]
  requires ["simulation:0"]
  sonar_name "F"
)
driver
(
  name "us_sonar"
  provides ["sonar:1" ]
  requires ["simulation:0"]
  sonar_name "R"
)
```

The USARSim robot is defined in 'simulation:0'. It's a P2AT robot. The 'odometry:::position:0' and 'laser:0' are USARSim devices. The 'requires

[simulation:0]' specifies that the devices are located on 'simulation:0' that is the P2AT robot.

If you want to control multiple robots with player, you can insert multiple simulation devices into the configuration file (provides ["simulation:0"],…, provides ["simulation:1"],…, provides ["simulation:2"],…) and connect the sensor to a simulation device using requires ["simulation:0"] or requires ["simulation:1"] . But it is easier to start a player server for each simulated robot. (If you want to use the player GUIs playerv and playernav you have to do this.)

If you want to start player you have to do the following steps:

1. start the USARSim server
2. adjust the host and port in your player configuration file to the USARSim server.
3. execute the player server: player p2at.cfg
4. execute playerv to see your sensor data.

13.3.2 Device Drivers

In this section, we explain how the USARSim devices work and how to configure them.

13.3.2.1 us_bot

Synopsis:

The us_bot driver is the bridge between Gamebots and the USARSim devices. It takes care the following tasks:

1) Connect to the Unreal server specified by 'host' and 'port'.
2) Spawn a robot of type 'bot' at location 'pos'.
3) Collect and parse the robot's data from Gamebots.
4) Provide the collected data to other USARSim devices and transfer these devices' commands to Gamebots.

Interfaces:

Supported interfaces:

- simulation

Required devices:

- None.

Configuration file options:

Name	Type	Default	Meaning
host	string	127.0.0.1	The Unreal server name
port	int	3000	The Gamebots port number
pos	int,int,int	0,0,0	The initial spawn position

rot	int,int,int	0,0,0	The initial spawn orientation
bot	string	P2AT	The robot type
botname	string	-	The robot name

13.3.2.2 us_position

Synopsis:

The us_position driver is used to control the robot's movement. It gets the robot's steering type from USARSim and interprets the player driving commands according to this steering type.

Interfaces:

Supported interfaces:

- position2d

Required devices:

- simulation.

Configuration file options:

Name	Type	Default	Meaning
simulation	int	-1	The simulation id that specifies the USARSim robot where this device is located.
Odo_name	String	Odometry	The name of the USARSim odometry sensor.

13.3.2.3 us_position3d

Synopsis:

The us_position3d driver is the same as us_position except that it uses the position3d interface.

Interfaces:

Supported interfaces:

- position3d

Required devices:

- simulation.

Configuration file options:

Name	Type	Default	Meaning
simulation	int	-1	The simulation id that specifies the USARSim robot where this device is located.

13.3.2.4 us_sonar

Synopsis:

The us_sonar driver is used to access the robot's sonar arrays.

Interfaces:

Supported interfaces:

- sonar

Required devices:

- simulation.

Configuration file options:

Name	Type	Default	Meaning
simulation	int	-1	The simulation id that specifies the USARSim robot where this device is located.
sonar_name	string	-	The name of your sonar array if you have a sonar array named "F1" – "F8" your sonar name must be "F"

13.3.2.5 us_laser

Synopsis:

The us_laser driver is used to access the robot's laser sensor. It only accesses the laser whose name is the same as the 'name' specified in the configuration file.

Interfaces:

Supported interfaces:

- laser

Required devices:

- simulation.

Configuration file options:

Name	Type	Default	Meaning
simulation	int	-1	The simulation id that specifies the USARSim robot where this device is located.
Laser_name	string	-	The name of the laser.

13.3.2.6 us_fakelocalize

Synopsis:

The us_fakelocalize driver is used to provide a ground truth localization in player using the USARSim STA message.

Interfaces:

Supported interfaces:

- position2d

Required devices:

- simulation.

Configuration file options:

Name	Type	Default	Meaning
simulation	int	-1	The simulation id that specifies the USARSim robot where this device is located.
origin	int,int,int	0,0,0	This value is added to the USARSim position to adjust the USARSim coordinate system to the player coordinate system. Only x and y are used.

13.3.2.7 us_ptz

Synopsis:

The us_ptz driver is used to control the robot's ptz camera. But you can't get the current orientation and camera velocity yet. The camera should use absolute pose control.

Interfaces:

Supported interfaces:

- ptz

Required devices:

- simulation.

Configuration file options:

Name	Type	Default	Meaning
Simulation	int	-1	The simulation id that specifies the USARSim robot where this device is located.

NOTE: The camera should use absolute pose control.

13.3.2.8 Known bugs

1. There is frequently the "Read Timeout while trying to read from server" error in us_bot.cc. But it seems like this doesn't affect anything.
2. Sometimes the us_position device is not able to get the robot configuration from USARSim. If that's the fact you can't control the robot from player or you even get a "can't subscribe to position2d device" error and player shuts down.
3. If you subscribe to a laser or a sonar device and the robot is not in 0,0,0 orientation the sensors point in the wrong direction. This is because

USARSim GEO message supports the sensor orientation in a global coordinate system and not in a robot centered coordinate system.

14 Advanced User

This section is for advanced users who want to build their own additions to the simulator. We assume the user already has programming experience or 3D modeling experience and robot background.

Before we start this section, we need to change the ut2004.ini file found in the Unreal system directory. Adding the following lines to the corresponding sections in ut2004.ini will let the Unreal engine recognize our own model. With this modification, we can compile and use our models in Unreal Editor.

```
[Engine.GameEngine]
ServerPackages=USARBot
ServerPackages=USARBotAPI
ServerPackages=USARMisPkg
ServerPackages=USARModels
ServerPackages=USARVictims

[Editor.EditorEngine]
EditPackages=USARBot
EditPackages=USARBotAPI
EditPackages=USARMisPkg
EditPackages=USARModels
EditPackages=USARVictims

[UnrealEd.UnrealEdEngine]
  ; Use this section only if you want USARSim packages to
  ; automatically
  ; load up when you start UnrealEd.
  EditPackages=USARBot
EditPackages=USARBotAPI
EditPackages=USARMisPkg
EditPackages=USARModels
EditPackages=USARVictims
```

NOTE: You need to modify ut2004.ini before you build your own models.

14.1 Build your arena

An arena is an Unreal map. It includes geometric models and objects in the environment. The objects can be obstacles such as bricks or victims that can move their bodies. Before building your arena, we must keep in mind that all the meshes must be static meshes. Karma objects only works well with static meshes. In addition, static meshes can accelerate 3D graphic rendering.

NOTE: All the meshes must be *static mesh*. The Karma engine only works well with static meshes.

When you build a new arena, there are three things you may need to do: 1) build the geometric model, 2) simulate some special effects, and 3) add objects such as obstacles and victims into the arena. The three things are explained in the following sections.

14.1.1 Geometric model

We have two options for building a geometric model. One is to import an existing model into Unreal. The other is to build the model by hand in Unreal. After building the model, we need to transfer it into a static mesh.

To facilitate users building their own arenas, we modeled all the parts used for building the NIST arenas. The model packages are located in the file ut2004\StaticMeshes\NIST.usx and ut2004\StaticMeshes\USAR_Meshes.usx.

Figure 50: Some NIST facilities

14.1.1.1 Import an existing model

The basic idea of importing a model is to convert your model into a format that Unreal Editor can read in. The file formats that are supported by the Unreal engine are:

- ASC: A 3D graphics file created from 3D Studio Max.
- ASE: Short for ASCII Scene Exporter.
- DXF: 3D graphic image file originally created by AutoDesk which stores 3D scenes and models.
- LWO: Is from LightWave model program.
- T3D: Is a text file that holds a text list of Unreal map objects.

Details about how to import a 3D model are described in the document:

UDN: Converting CAD data into Unreal
(http://udn.epicgames.com/Two/CADtoUnreal).

14.1.1.2 Build it with Unreal Editor

Unreal Editor is a nice 3D authoring tool. There are two websites you may need to visit if you want to learn how to build a map with Unreal Editor.

UDN (Unreal Developer Network): http://udn.epicgames.com

Unreal Wiki: http://wiki.beyondunreal.com/wiki/

The 'Basics' category in UDN contains documents with all of the details of modeling with the Unreal Editor. And the 'Topics On Mapping' under Unreal Wiki (http://wiki.beyondunreal.com/wiki/Topics_On_Mapping) lists all the topics involved in mapping.

14.1.2 Special effects

Most of the special effects are obtained by applying special materials. Please read the UDN: Material Tutorial (http://udn.epicgames.com/Two/MaterialTutorial) to have a sense of what an Unreal material is.

The mask effect (parts of material are either opaque or transparent) is achieved by using textures with an alpha-channel. The gray level in the alpha-channel indicates how transparent the corresponding pixel will be. Alpha-channel with grid bitmap will bring us the grid fender effect.

The glass effect is simulated by semi-transparent material. A texture with a gray alpha-channel will give us a semi-transparent effect. Using shaded material, we can get higher fidelity effects.

The mirror effect is obtained by using scripted texture. The basic idea is to put a camera in the place you want to put the mirror and then render the picture from the camera, into the place where the mirror is. The idea comes from Angel Mapper's reflection tutorial (http://angelmapper.com/tutorials/reflections.htm). The details about how to add a mirror can be found at the Security Camera Tutorial that is located at http://angelmapper.com/tutorials/securitycamera.htm. According to the author, this approach doesn't work online. To fix this shortcoming, a customized CameraTextureClient named myCameraTextureClient is created in USARSim. Replacing all the CameraTextureClient by myCameraTextureClient in the tutorial, will give us a mirror effect that works online. To add myCameraTextureClient, go to the 'Actor Classes' browser in Unreal Editor, select myCameraTextureClient from the path:

Actor\Info\CameraTextureClient\myCameraTextureClient

14.1.3 Obstacles and Victims

To get a high fidelity simulation, we recommend using Karma objects as the obstacles. An example of adding Karma objects in a map can be found at UDN: Karma Colosseum (http://udn.epicgames.com/Two/ExampleMapsKarmaColosseum).

There is a known bug in UT2004 that the KActor doesn't support networks well. The KNActor included in USARSim is the substitute that fixes this bug.

Victims are another type of objects we may need to put into the map. Victims are special objects that can implement some actions. The victim model built in USARSim can be loaded from the Unreal Editor. To load it, please open the 'Actor Classes' browser and select the USARVictim from the following path:

Actor\Pawn\UnrealPawn\xIntroPawn\USARVictim

After you put it on the map, you can

1) Set the mesh

The default mesh is 'Intro_gorgefan.Intro_gorgefan'. To change the mesh, double click the victim to pop up the 'USARVictim Properties'. Then, open the 'Display' category. Changing the 'Mesh' item in this category will set the victim's mesh.

2) Specify the actions

In the 'USARVictim Properties', under the 'Victim' category are the parameters that specify the victim's actions. These parameters are:

AnimTimer	Sets how quickly the victim moves. Low value means a slow action.
HelpSound	Sets the sound the victim can play
Segments	Specifies how the body segment moves. You can set at most 8 segments. For every segment, you can define an action. The segment will move from the initial pose to the final pose with the specified move rate. The action definition parameters are:

InitRotation	The initial rotation (pitch, yaw, and roll in integer. 65535 means 360 degrees) of the segment.
FinalRotation	The final rotation (pitch, yaw and roll in integer. 65535 means 360 degrees) of the segment.
PitchRate	The move amount from current pitch angle to the next pitch angle. Large PitchRate means tilt quickly.
YawRate	It's the same as PitchRate except that it defines the yaw angle.
RollRate	It's the same as PitchRate except that it defines the roll angle.
Scale	The scale of this segment. '0' will hide this segment. Since there is hierarchical relationship in the skeletal system, this scale value will affect other segments

	under it. For example, hips will affect thigh, shine and foot.
SegName	The name of the segment. Different skeletal meshes may have different names. You can use the ‘Animations’ browser to view the bone name. An example in showed in Figure 51.

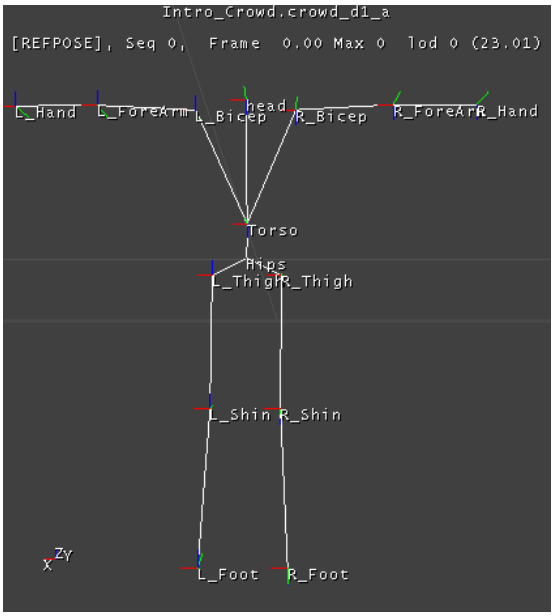


Figure 517: Skeletal bones name

For more details about skeletal mesh, please visit:

UDN: AnimBrowserReference (http://udn.epicgames.com/Two/AnimBrowserReference)

UDN: UWSkelAnim2 (http://udn.epicgames.com/Two/SkelAnim2)

After you set the actions, the victim will not move immediately. In Unreal Editor, everything is static. To let them to be active, you need to play the map.

As we know, there is a bug in the Unreal engine. Some meshes may play their default animations when your viewpoint is far away from the victim.

NOTE: There is hierarchical relationship in the skeletal system. Changing one scale value may affect other segments under it. For example, hips will affect thighs, shins and feet.

14.2 Build your sensor

Before you build your sensors, you need to understand Unreal Script and the client/server architecture of the Unreal engine. The following resources may be helpful to you:

UDN: UnrealScriptReference (http://udn.epicgames.com/Two/UnrealScriptReference)

UnrealWiki: UnrealScript Topics (http://wiki.beyondunreal.com/wiki/UnrealScript)

Unreal Networking Architecture (http://unreal.epicgames.com/Network.htm)

14.2.1 Overview

In USARSim, all sensors are inherited from the Sensor class. The Sensor class defines the interfaces that the robot model can interact with. We use a hierarchical architecture to build the sensors. The hierarchy chart is shown below.

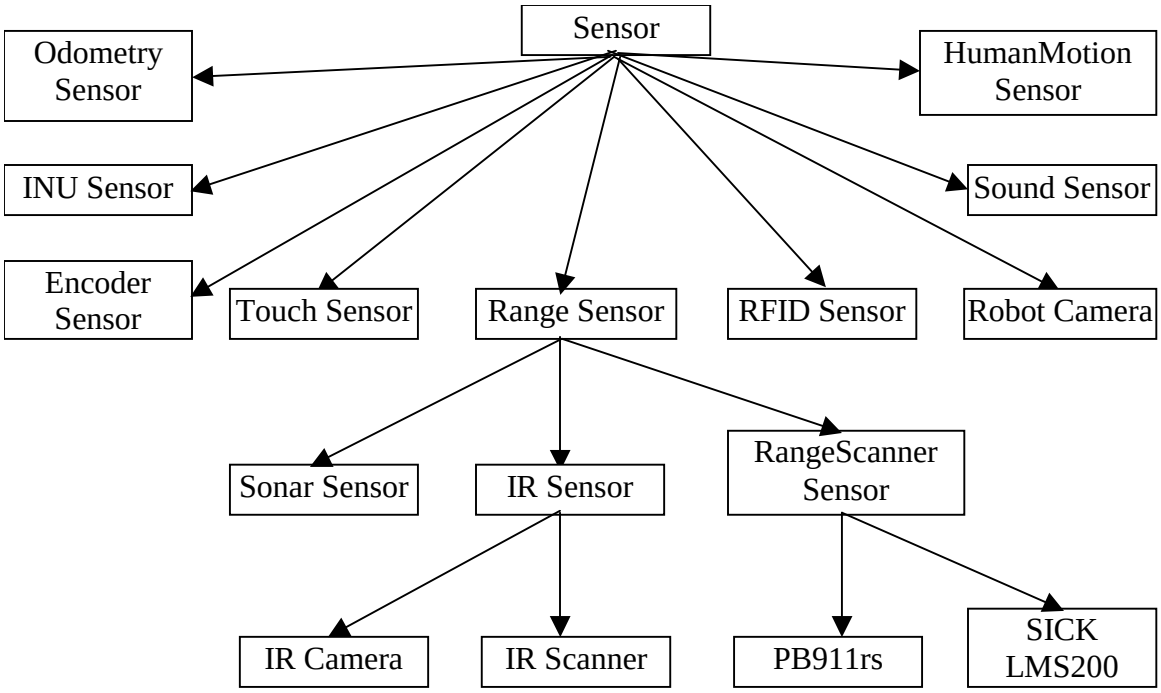


Figure 528: Sensor Hierarchy Chart

14.2.2 Sensor Class

The Sensor class is the ancestor of all the sensor classes. It extends from the Item class which is the base class for all the items that can be mounted on the robot. The Item class takes care of creating the item, mounting itself on the robot, providing a command response interface, and preparing some information that will be helpful to the user. The sensor class provides the basic interaction interface to send out data. The details about Item classes are explained below:

Attributes:

```
var string ItemName; // the item's name
var string ItemType; // the item's type
var string ItemMount; // the  mount base
var vector myPosition; // the mounting position
```

var rotator myDirection; // the mounting direction
var USARConverter converter; // the converter object used by USARSim to do unit and cooridniate conversion
var KVehicle Platform; // the item's robot platform

Methods:

function SetName(String iName) // set the item's name
function Init(String SName, Actor parent, vector position, rotator direction, KVehicle veh, name mount) // mount the item
function ConvertParam(USARConverter converter) // transfer the item's parameters' units and coordinates to Unreal units and coordinates.
function string Set(String opcode, String args) // the interface of the SET command
function bool isType(String type) // return whether the item's type matches the specified type
function bool isName(String name) // return whether the item's name matches the specified name

The new variables and functions introduced in the Sensor class are:

Attributes:

var config bool HiddenSensor; // variable that indicates whether to show the
// sensor in Unreal
var config InterpCurve OutputCurve; // the distortion curve
var config float Noise; // the random noise amount

Methods:

function String GetHead() // the interface that sends sensor data HEAD to the robot. It's usually something like "SEN {Type xxx}"
function String GetData() // the interface that sends sensor data to the robot. For example, it can be "{Name xxx} {Pose x,y,theta}"
function String GetGeoHead() // the interface that sends the sensor's geometric information HEAD to the robot
function String GetGeoData() // the interface that sends the sensor's geometric data to the robot
function String GetConfHead() // the interface that sends the sensor's configuration information HEAD to the robot
function String GetConfData() // the interface that sends the sensor's configuration data to the robot

14.2.3 Writing your own sensor

Your sensor should extend from the Sensor class. You may add your own sensor parameters, and these parameters should be in the API interface's units and coordinates. In the function ConvertParam, you transfer them to Unreal's units and coordinates. Then you may override the Getxxxx methods to return your own data in a string. When you generate this data string, you need to convert the units and coordinates back to the API interface's units and coordinates.

The robot model will call the isType and isName functions to find out whether the sensor is the current sensor it needs to process. By default, these two functions do simple string matching. You can override them to do some advance things like considering super type and sub type. Although there are Noise and OutputCurve parameters in Sensor class, it does nothing about the noise data simulation and data distortion simulation. It's your responsibility to simulate them in the GetData method.

NOTE: You MUST use the "converter" object to do all the unit and coordinate conversion for flexibility and consistency reasons. Always use isType and isName function in your code to do the type and name matching.

14.3 Build your effecter

This section of the manual will provide you an overview of how to implement a new effecter in USARSim. In the first subsection, an overview of the functions and structures in the Effecter base class will be presented. The second subsection will show how to implement on opcode in a new effecter class.

14.3.1 Overview of the Effecter.uc class

In USARSim, all effecters are inherited from the Effecter class. The Effecter base class defines the valid opcodes and the means to query/format information from the various extensions of effecters. As mention in Section 8, effecters in USARSim contain a restricted vocabulary of operational codes. These opcodes are defined at the top of the Effecter.uc file, as shown below

```
enum EFFECTOR_OPCODE_TYPE
{
   EFFECTOR_OPCODE_ACTIVATE_TYPE,
   EFFECTOR_OPCODE_ANIMATE_TYPE,
   EFFECTOR_OPCODE_FIRE_TYPE,
   EFFECTOR_OPCODE_RELEASE_TYPE,
   EFFECTOR_OPCODE_RESET_TYPE,
   EFFECTOR_OPCODE_NOP_TYPE
};
```

The base effecter class implements several virtual functions that will enable the base class to query and send commands to its children. There are two basic categories of virtual functions that are implemented in the effecter class; the "Do<*Opcode Name*>" functions and the "Get<*Opcode Name*>Conf" functions. Therefore, the developer must implement both functions for each opcode in order to properly implement the opcode. These functions will overwrite the virtual functions, letting the effecter base class know that these are valid opcodes that can be used for a given effecter.

The virtual "Do<*Opcode Name*>" functions defined in Effecter.uc. These function are used to perform opcodes in the effecter. The value specifies the value that is associated with the opcode and the returns a Boolean to indicate whether the command was successfully implemented or not.

```
function Bool DoActivate(float val)
function Bool DoAnimate(float val)
function Bool DoFire(float val)
function Bool DoRelease(float val)
function Bool DoReset(float val)
function Bool DoNOP(float val)
```

The virtual "Get<*Opcode Name*>Conf" functions defined in Effecter.uc. This function enables the base class to retrieve configuration information from a referenced argument, EffecterOpcodeConfig structure. It returns a Boolean to indicate whether the opcode is implemented or not.

```
function Bool GetActivateConf(out EffectorOpcodeConfig opcodeConf)
function Bool GetAnimateConf(out EffectorOpcodeConfig opcodeConf)
function Bool GetFireConf(out EffectorOpcodeConfig opcodeConf)
function Bool GetReleaseConf(out EffectorOpcodeConfig opcodeConf)
function Bool GetResetConf(out EffectorOpcodeConfig opcodeConf)
function Bool GetNOPConf(out EffectorOpcodeConfig opcodeConf)
```

The Effector Opcode Configuration Structure is a structure is used by the FormatOpcodeConf to produce the appropriate CONF message for a given opcode. Therefore, this structure should be used by derived classes to hold configuration information and will enable derived classes to maintain appropriate information for each opcode.

```
function String FormatOpcodeConf(EffectorOpcodeConfig opcodeConf)
struct EffectorOpcodeConfig
{
  var () EFFECTOR_OPCODE_TYPE opcode;
  var () float maxVal;
  var () float minVal;
};
```

The effecter class also defines some utility functions. There are two lookup functions that convert the name of the opcode to the opcode and vice versa.

```
function EFFECTOR_OPCODE_TYPE GetOpcodeType(string opStr)
function String GetOpcodeName(EFFECTOR_OPCODE_TYPE eType)
```

14.3.2 Writing your own effecter

Same as section 14.2.3

Here is an example of how to implement the "Animate" operational code in a new effecter class.

> NOTE: You do not have to implement the "Get<*Opcode Name*>Conf" and the "Do<*Opcode Name*>" for every opcode. You only need to implement the appropriate functions for the opcode you are attempting to implement.

First, overwrite the 'GetAnimateConf' function in the new effecter class.

```
function Bool getAnimateConf(out EffectorOpcodeConfig opcodeConf) {
  opcodeConf.opcode =EFFECTOR_OPCODE_ANIMATE_TYPE
  opcodeConf.maxVal=1.57;
  opcodeConf.maxVal=0;
  return true;    //indicates that this is a valid opcode.
}
```

Next, overwrite the "DoAnimate" function in the new effecter class.

```
function Bool DoAnimate(float s) {
        … code for doing the opcode …
        if( code was successful )
                return true;
        else
                return false;
}
```

14.4 Build your robot

Usually, building a robot involves a lot of programming, deeply understanding Unreal network architecture, and the background knowledge of mathematics and mechanics. It takes a lot of time in programming and debugging. To facilitate the robot building, we built a general robot model to help users build their own robot. In the robot model, every robot is constructed of:

- Chassis: the chassis of the robot.
- Parts: the mechanical parts, such as tires, linkages, camera frame etc., that are used to construct the robot.
- Joints: the constraints that connect two parts together. In the robot model, we use Car Wheel Joint.
- Attached Items: the auxiliary items, such as sensors, effecters etc., attached to the robot.

A chassis can connect to multiple parts through joints. However, each part can only have one joint. The attached items can be attached to either the chassis or a part. The chassis or part can have multiple attached items.

The working flow of building a robot is to first build a geometric model for all the objects used to construct the robot. Then create part/wheel classes for all the robot parts/wheels that extend from KDPart/USARTire, and a new robot class that extends from KRobot. In the robot class you set the physical attributes of the robot. And you also need to configure how the chassis, parts/wheels and auxiliary items are connected to each other. Lastly, if you want to add some new features not included in the robot model, you will do some programming work.

14.4.1 Step1: Build geometric model

Essentially, this step is the same as building your own arena. Please refer to section 14.1.1 to learn how to build a static mesh. One thing we want to emphasize

here is that the orientation of the geometric model is very important. You must let the X-axis of the model point to the head, and the Y-axis point to the right. An incorrect axis will bring you incorrect pitch, yaw and roll angles.

NOTE: Make sure the geometric model has the correct x-axis and y-axis. This will affect the attitude data.

14.4.2 Step2: Construct the robot

14.4.2.1 Create the part/wheel class

Here we create a wrapper class for our part or wheel geometry model. The part class looks like:

```
class part_class_name extends KDPart;
defaultproperties
{
  //properties
}
```

where part_class_name is the name of your part class. In defaultproperties, we point the StaticMesh to your part's geometry model; set the part's Weight, Mass and the Kparams (Karma parameters). For details, please refer the next section. For the wheel class, the only difference is that the class extends from USARTire not KDPart.

14.4.2.2 Create the robot class

First, you need to create a robot class that extends the KRobot. The class should look like:

```
class robot_class_name extends KRobot config(USAR);
defaultproperties
{
  //properties
}
```

where robot_class_name is the name of your class.

14.4.2.3 Prepare the attributes and objects used for your robot

In the defaultproperties block of the class, you can set the attributes of the robot. The attributes are:

MotorTorque	The default motor torque in Karma Units. Default value is 20.
MaxTorque	The maximum motor torque. Default value is 60. The control torque will be cut to this value if it's larger than MaxTorque.
MotorSpeed	The default motor speed. Default value is 0.1745 radians/second.
Weight	The weight of the chassis in kg. Please note, the value is

	used only for description purpose. The real value that affects the physical characteristic is ChassisMass.
ChassisMass	The mass of the chassis in Karma Units. Default value is 1.0.
StaticMesh	The static mesh for the chassis. The format looks like: StaticMesh'*your_mesh_name*'
DrawScale	The scale of the static mesh. Default is 0.3
DrawScale3D	The scale in X, Y and Z axes.
KParams	The Karma physical parameters of the chassis. It's a KarmaParams object. For details please read the UDN: KarmaReference (http://udn.epicgames.com/Two/KarmaReference).
ConverterClass	The class used by the robot for units and coordinates conversion. By default, it's "USARBot.USARConverter".

Besides these properties, you also can set the joints and tire parameters for the robot. These parameters will affect all the joints and tires. Usually you needn't change them. In case you want to change them, we list all the parameters below.

Name	Description	Default value
HingePropGap	The proportional gap used by a hinge joint.	364.0
SteerPropGap	The proportional gap used for steering speed control.	1000.0
SteerTorque	The torque applied to the steering.	1000.0
SteerSpeed	The steering speed.	15000.0
SuspStiffness	Stiffness of suspension springs.	150.0
SuspDamping	Damping of suspension.	15.0
SuspHighLimit	The highest offset from the suspension center in Karma scale, which is 1/50th of Unreal scale.	1.0
SuspLowLimit	The lowest offset from the suspension center in Karma scale, which is 1/50th of Unreal scale.	-1.0
TireRollFriction	Roll friction of the tire.	15.0
TireLateralFriction	Lateral friction of the tire.	15.0
TireRollSlip	Maximum first-order (force ~ velocity) slip in tire direction.	0.06
TireLateralSlip	Maximum first-order (force ~ velocity) slip in sideway direction.	0.06
TireMinSlip	The minimum slip in both directions.	0.001
TireSlipRate	The amount of slip per unit of velocity.	0.0005
TireSoftness	The softness of the tire.	0.0
TireAdhesion	The stickyness of the tire.	0.0
TireRestitution	The bouncyness of the tire.	0.0

TIPS: Low TireSlipRate and high friction give the tire high climbing capability.

14.4.2.4 Connect the parts/wheels

After we set up all the attributes and classes, we can use the part-joint pairs to connect the chassis and parts. In the part-joint pair we define the part and how it is connected to another part through a joint. Currently, we support two kinds of joints, the car-wheel joint that is used to connect a wheel to the robot, and the hinge joint that is used to link any parts together.

A car-wheel joint connects two parts by two axes. One is the spin axis (hinge axis in Figure 53) that the part can spin around. Another is the steering and suspension axis (Steering Axis in Figure 53) that the part can steer around and travel along. A hinge joint connects two parts by one axis.

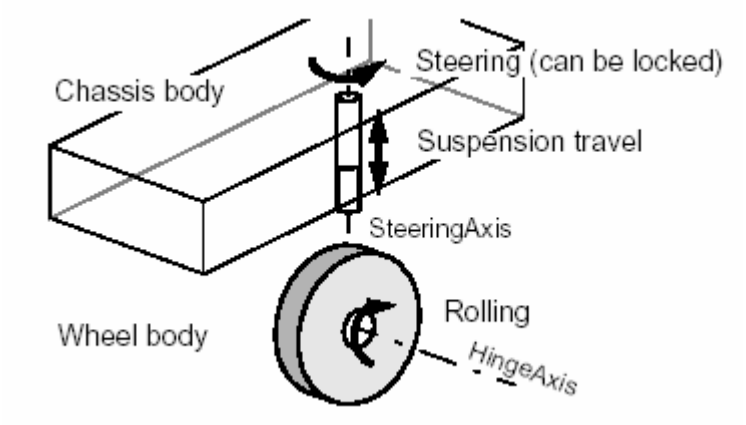


Figure 539: Car wheel joint

The part-joint pair is a structure defined below:

```
struct JointPart {
  // Part
  var() name                    PartName;
  var() class<KActor>           PartClass;
  var() vector                  DrawScale3D;
  // Joint
  var() class<KConstraint>      JointClass;
  var() bool                    bSteeringLocked;
  var() bool                    bSuspensionLocked;
  var() float                   BrakeTorque;
  var() name                    Parent;
  var() vector                  ParentPos;
  var() vector                  ParentAxis;
  var() vector                  ParentAxis2;
  var() vector                  SelfPos;
  var() vector                  SelfAxis;
  var() vector                  SelfAxis2;
```

};

where

PartName	The name of the part.
PartClass	The part's class name. It can be Class'USARBot.KDPart' or the tire's class name.
DrawScale3D	The scale along X, Y and Z axes of the static mesh. Please note, this only changes how the part looks look. In unreal engine, it still uses the original mesh for collision detection. Please use it carefully.
JointClass	The joint's class name. It should be: class'KCarWheelJoint' or class'KDHinge'.
bSteeringLocked	Indicates whether steering is locked if we are using a car-wheel joint.
bSuspensionLocked	Indicates whether suspension is locked if we are using a car-wheel joint.
BrakeTorque	The brake torque applied for braking the joint if we are using a car-wheel joint.
Parent	The part or chassis the part is connecting to. NOTE: the part must have already been defined.
ParentPos	The position where the joint connects to the parent.
ParentAxis	For a car-wheel joint, it's the steering axis relative to the parent. For a hinge joint, it's the spin axis.
ParentAxis2	For a car-wheel joint, it's the spin axis relative to the parent.
SelfPos	The position where the joint connects the part.
SelfAxis	For a car-wheel joint, it's the steering axis relative to the part. For a hinge joint, it's the spin axis.
SelfAxis2	For a car-wheel joint, it's the spin axis relative to the part

The order in which you define the part-joint pairs is important. Since the parent in the part-joint pair must already be defined, you need to define the parent before the part. You also can define these part-joint pairs in the USARBot.ini file (you may need to create the robot section by yourself). By using the USARBot.ini file, you needn't compile your class after you change something.

You may find that it's not easy to know the joint position relative to the parent and the part. One way to help you figure out these values is using the Unreal Editor. At first, you put all the chassis and parts in the map in the draw scale you want. Then you assemble them together in the map. Using some simple geometric objects to represent the joints, you can put them on the connection position you want. You also may need to assign a name to every object to help you distinguish them. After that, you can export the map as a t3d file. In the t3d file, you will find every object's position. By subtracting the parent or part's position from the joint position, you will get the accurate relative position.

TIP: Assembling the robot in Unreal Editor can help you calculate the relative position.

Like the real mechanical world, improper mechanical structure can cause the robot to be unstable. When you create the robot, make sure your geometric model is correct. You especially need to check if the model has the correct mass distribution. In some cases, you may need to specify the mass center offset in the KarmaParams. When your robot is unstable, try to add the parts one by one. This can help you figure out which part causes the problem.

TIP: Specifying the mass center offset in the KarmaParams can help you simulate the mass distribution.

14.4.2.5 Mount the auxiliary items

After you created the robot, you can mount other items on it. To mount an item, please use the following data structure:

```
struct sItem {
 var class<Actor>      ItemClass;
 var name              Parent;
 var string            ItemName;
 var vector            Position;
 var vector            Direction;
 var rotator           uuDirection;
};
```

where

ItemClass	The class used to create the item.
Parent	The object on which the item will mount.
ItemName	The name assigned to this item.
Position	The mounting position relative to its parent.
Direction	The mounting direction relative to its parent.
uuDirection	The reserved variable that stores the direction parameter in unreal units.

In the robot class, “Sensors” is used for all the sensors. “Effecters” is used for all the effecters. And the “Cameras” stores all the cameras.

14.4.3 Step3: Customize the robot (Optional)

After finishing the previous two steps, your robot should work. You should be able to use the DRIVE command to control every joint and you can also get the sensor data from the robot. To go further beyond this, you can do three things:

1) Write your own control mode

The general robot mode only supports controlling every joint separately and two types of mission package control, pan-tilt and flipper. However, you can define some control pattern or even control model in your class.

USARSim uses the 'DRIVE {Left xxx} {Right xxx}' command to interact with Pyro. The Left and Right means left side and right side wheels separately. This is an example of a control pattern. In the robot class, you can transfer the left, right parameters into a series of joint control parameters to control the wheels. This can be reached by overriding the "ProcessCarInput()" function of the KRobot class. In your own ProcessCarInput(), you need to call the ProcessCarInput() function in KRobot to let your robot interpret the joint control command. Once you added the left, right parameters interpretation, your robot should be able to be controlled by Pyro. As an example, you can open the source code of P2AT to learn how it supports the 'DRIVE {Left xxx} {Right xxx}' command. The 'CAMERA' command is another command used to interact with Pyro. You also can learn how to interpret it in the P2AT.uc file.

2) Add your own commands

Besides supporting the commands used by USARSim, you also can add your own command. As we mentioned before, the commands come from Gamebots. A robot connects with Gamebots through its controller whose class is USARRemotebot. Every USARRemotebot is associated with a USARBotConnection that listens to a TCP/IP socket and parses the incoming commands. Once a new command is received, USARBotConnection realizes it and gets the value in the command. Then it sets the corresponding variable in USARRemotebot to the new value. In your robot class, you only need to check the USARRemotebot's variable to get the command data.

In summary, to add a new command:

1) Add a new variable in USARRemotebot to store the command's data.
2) In USARBotConnection, add your code into the ProcessAction function to interpret your command and store it in the USARRemotebots's variable.
3) In your robot class, check the USARRemoteBot's variable to get the command and do something you want.

3) Maintain the robot's state by yourself

Some robots may have special states to maintain, for example, the following wheel of the P2DX robot, the chassis of PER. The state of the following wheel of P2DX is totally decided by the other two wheels. This is not included in the general robot model. So you need to maintain its state by yourself. It's the same as the chassis of the PER. PER's chassis is controlled by a differential that force the chassis's pitch angle to always be the average of the left and right wheel rocker angles.

To maintain the robot's state, you need to override the Tick() function. In every Unreal tick, you update the robot's state and you also need to explicitly or

implicitly call the Karma update state function KUpdateState(). You can use the code of P2DX as example to learn how to maintain your own state.

Lastly, besides the three aspects mentioned above, obviously, you can do just about anything you want in your robot class.

14.5 Build your controller

The client/server architecture makes it easy to build your own control client. You only need to follow the communication protocol. Since the protocol is line based, you need to use the '\r\n' to determine when a message ends. When you send out a command, you need to add '\r\n' to inform USARSim that the command is finished. In unreal engine, a tick is the minimum time used for checking and updating states. If you send commands at a higher frequency than the time interval between two ticks, then the engine will only process the last command. So please don't send your commands at very high frequency.

NOTE: Don't send your command at a frequency higher than the engine's state update frequency.

If you want to do some image processing or include the video feedback on your own interface, there are some technical details you may need to know. As discussed in section 10.13, there are four ways to get/use video feedback. Except directly using Unreal Client as a separated window, the other three approaches are discussed below:

14.5.1 Embedding Unreal Client

The idea is to attach the Unreal Client into your application. Basically, under windows, this can be reached in 4 steps:

1) Get the window handle of Unreal Client.

   For example, in C++, we can use:
   CWnd * m_AppWnd = FindWindow(NULL, "Unreal Tournament 2004");

2) Move and scale the Unreal Client to your desired region.

   In C++, it may looks like:
   m_AppWnd->SetWindowPos(this, 60, 40, 400, 300, NULL);
   where 'this' is the pointer of your application.

3) Modify the Unreal Client's window style to let it look like a part of your application.

   For example, we use the following C++ code to remove the title bar and change the border to thick frame.
   m_AppWnd->ModifyStyle(WS_CAPTION, NULL, SWP_DRAWFRAME);
   m_AppWnd->ModifyStyle(WS_THICKFRAME, NULL, SWP_DRAWFRAME);

4) Set your application to be Unreal Client's parent window.

   In C++, we use:

   m_AppWnd->SetParent(this); // where 'this' is the pointer of your application

14.5.2 Capturing Unreal Client

In USARSim, Hook.dll provides help to get the scenes from the Unreal Client. This DLL uses Detours technology (http://www.research.microsoft.com/sn/detours/) to capture the back buffer of DirectX 8.x and store it as a raw picture in a block of shared memory. To use this DLL, we need to:

1) Attach the DLL to the Unreal Client

   We can use the detours function, DetourCreateProcessWithDll(), to combine Hook.dll to the Unreal Client. For more details about this function, please read the withdll example that comes with Detours. You can also find the example code in the SimpleUI source files. The LoadUT() function of CControlDlg class is the exact function that attaches Hook.dll to the Unreal Client and then launches it. Because the Hook.dll needs to catch the Direct3DCreate8() function, the DLL must be attached to the Unreal Client before the Unreal Client runs. This is the reason why we use the Detours function DetourCreateProcessWithDll().

2) Get the address of the shared memory

   getFrameData() is the function provided by Hook.dll that tells you the address of the shared memory. To get the memory address, we need to:

   i. Get the module handle of Hook.dll by using LoadLibrary().
   ii. Get the function address of getFrameData() by using GetProcAddress().
   iii. Call the function getFrameData() to get the memory address.

   The example code can be found at GetpfFrameData() function of CControlDlg class in the SimpleUI source files.

The format of the data in the memory is defined below:

```
#define FRAME_PENDING 0
#define FRAME_OK 1
#define FRAME_ERROR 2
typedef struct FrameData_t {
  BYTE state;
  BYTE sequence;
  USHORT width;
  USHORT height;
  UINT size;
  BYTE data[640*480*3+1];
} FrameData;
```

where

state	The state of the memory. It can be:
	FRAME_PENDING The memory is in use by the DLL
	FRAME_OK The memory is ready for reading
	FRAME_ERROR Something is wrong with the data
sequence	The sequence of the data. The DLL only captures a new

picture when it gets a new sequence number. You can use it to control when the DLL captures a picture.

width	The width of the captured picture. The maximum width is 640. If the Unreal Client's window width is larger than 640, the DLL will not capture any pictures.
height	The height of the captured picture. The maximum height is 480. If the Unreal Client's window height is larger than 480, the DLL will not capture any pictures.
size	The actual data length in the 'data' array. When the picture width is not in DWORD boundary, '0' is padded to reach the DWORD boundary. In this case, the size isn't width*height*3.
data	The array stores the picture data. The picture is stored from left to right, from top to bottom. A pixel is represented as Red + Green + Blue. Each color occupies one byte.

14.5.3 Using the Image Server

The Image Server simulates a web camera. How to run it is described in section 5.2.5.2. Its workflow is:

1) Send out a picture when the client connects with it.
2) Wait for the acknowledgement from the client.
3) If the current time is the sending time triggered in the specified frame rate, then send out the next picture.
4) Go to step 2).

The server supports both raw pictures and jpeg pictures. The image data format is:

ImageType (1 byte) + ImageSize (4 bytes) + ImageData (n bytes)

Where:

ImageType	The format of the image. It can be: 0 raw data 1 jpeg in super quality 2 jpeg in good quality 3 jpeg in normal quality 4 jpeg in averagequality 5 jpeg in bad quality
ImageSize	The total length of ImageData in bytes.
ImageData	The actual data of the image. For raw data, the ImageData is: width (2 bytes) + height (2 bytes) + RGB (1 byte + 1 byte + 1 byte) data from left to right, from top to bottom. For jpeg, the ImageData is the real jpeg data which can be decompressed by any jpeg decoders.

The image transfer protocol is very simple. When the client gets the image, it sends back an acknowledgement meesage 'OK' (in plain text) to the image server.

To use the image server, you only need to follow this simple protocol and the image format. As an example of how to use the image server, the source code of SimpleUI is included in the USARSim package. The SimpleUI uses the FreeImage (http://sourceforge.net/projects/freeimage) DLL to decode jpeg pictures.

15 Information for Gamers

Some public UT2004 game servers running the AntiTCC anti-cheat service may be set up to automatically ban users who have Hook.dll installed, as it (or at least files like it) can be used for aimbots (programs used by cheats to improve aim). USARSim uses this file to hook into UT2004's framebuffer to get images for the simulated camera and it can be found in the <UT2004directory>\System directory (alongside UT2004.exe).

The best way around this is to temporarily move hook.dll to somewhere outside the UT2004 directory whilst playing on public servers.
Saving the following 3 lines to something like "playUT.bat", putting it in <UT2004directory>\System and making a play link that points to it is one way around this problem.

```
========================
move hook.dll ..\..\
UT2004.exe
move ..\..\hook.dll .
========================
```

16 Bug report

Please use the sourceforge message forums to report bugs in USARSim. This forum may be found at http://sourceforge.net/tracker/?group_id=145394&atid=761824.

17 Contributors

The primary software author:

- Jijun Wang at University of Pittsburgh, USA
- Major rewrites and contributions by Ben Balaguer of NIST, USA

The primary manual authors:

- Jijun Wang at University of Pittsburgh, USA
- Stephen Balakirsky of NIST, USA

Management and Coordination:

- Stephen Balakirsky at National Institute of Standards, USA
- Michael Lewis at University of Pittsburgh, USA
- Stefano Carpin at University of California, Merced

USARSim drivers for Player:

- Version 1.6 from Erik Winter at Uppsala University, Sweden
- Modified Version 1.6 from Stefan Markov & Ravi Rathnam at International University Bremen, Germany
- Version 2.0 from Stefan Stiene at University of Osnabrück

System architecture

- Mission package concept from Stephen Balakirsky and Chris Scrapper at National Institute of Standards, USA
- Unit and coordinate conversion idea from Chris Scrapper at National Institute of Standards, USA
- Effecter concept from Chris Scrapper at National Institute of Standards, USA

Robot

- The KDHinge from Marco Zaratti at University of Rome "La Sapienza", Italy
- Hummer by Ben Balaguer of NIST
- Stereo P2AT from Giuliano Polverari at University of Rome "La Sapienza", Italy
- Lisa, Kurt2D, Kurt3D from Stefan Stiene at University of Osnabrück
- Passarola from Ricardo Alcácer (ricardoalcacer@gmail.com) IST – Instituto Superior Técnico Department of ISR - Institute for Systems and Robotics Portugal
- Rugbot from Todor Stoyanov, Jacobs University, Breman, Germany.
- Kenaf from Kensuke Kurose of Tadokoro Laboratory, Tohoku University, Japan.

Sensors

- INU sensor from Stefan Markov and Ivan Delchev at International University Bremen, Germany
- IR and Sonar sensor from Erik Winter at Uppsala University, Sweden
- The first draft Encoder sensor was written by Andreas Nüchter at University of Osnabrück, Germany
- IR scanner from Giuliano Polverari at University of Rome "La Sapienza", Italy
- The first version of RFID tag and sensor from Alexander Kleiner at University of Freiburg, Germany
- The current version RFID tag and sensors from Mentar Mahmudi at International University Bremen, Germany
- Version 1 of the Victim sensor and tags from Stephen Balakirsky at National Institute of Standards of Technology, USA.
- Version 2 of the Victim sensor by Ben Balaguer
- GPS sensor from Tyler Folsom, University of British Columbia, Canada
- Modified GPS from Ben Balaguer, University of California, Merced.

Radio Model

- Yashodhan Nevatia from International University Bremen, Germany

Tools

- ImageServer from Jijun Wang at University of Pittsburgh, USA. Modifications to allow client only operation by Marco Zaratti.
- MultiView from Marco Zaratti.
- Omni-view camera from T. Schmits of the University of Amsterdam

18 Acknowledgements

This simulator was developed under grant NSF-ITR-0205526, Katia Sycara and Illah Nourkbaksh of Carnegie Mellon University and Michael Lewis of the University of Pittsburgh Co-PIs. Elena Messina and Brian Weiss of NIST provided extensive assistance. Joe Manojlovich, Jeff Gennari, and Sona Narayanan contributed to the development of the simulator. Eric Garcia and Stephen Balakirsky edited this document.