USARSim

A Game-based Simulation of the NIST Reference Arenas

Prepared by Jijun Wang

USARSim

Contents

1 Introduction

1.1 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. .

1.2 What is USARSim

USARSim is 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. 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 pre-Beta release of the simulation consists of: **environmental models** (levels) of the NIST Yellow, Orange, and Red Arenas as well as a partially textured version of the Nike Silo reference environment, **robot models** of commercial and experimental robots, and **sensor models**. As a simulation user, you are expected to supply the user interfaces and 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 or Pyro middleware. A collection of interface components such as an adjustable frame rate/FOV video window, support for a wider range of commercial robots, and support for "approximately native", Pyro, and Player control interfaces are planned for future releases.

2 System Overview

2.1 System architecture

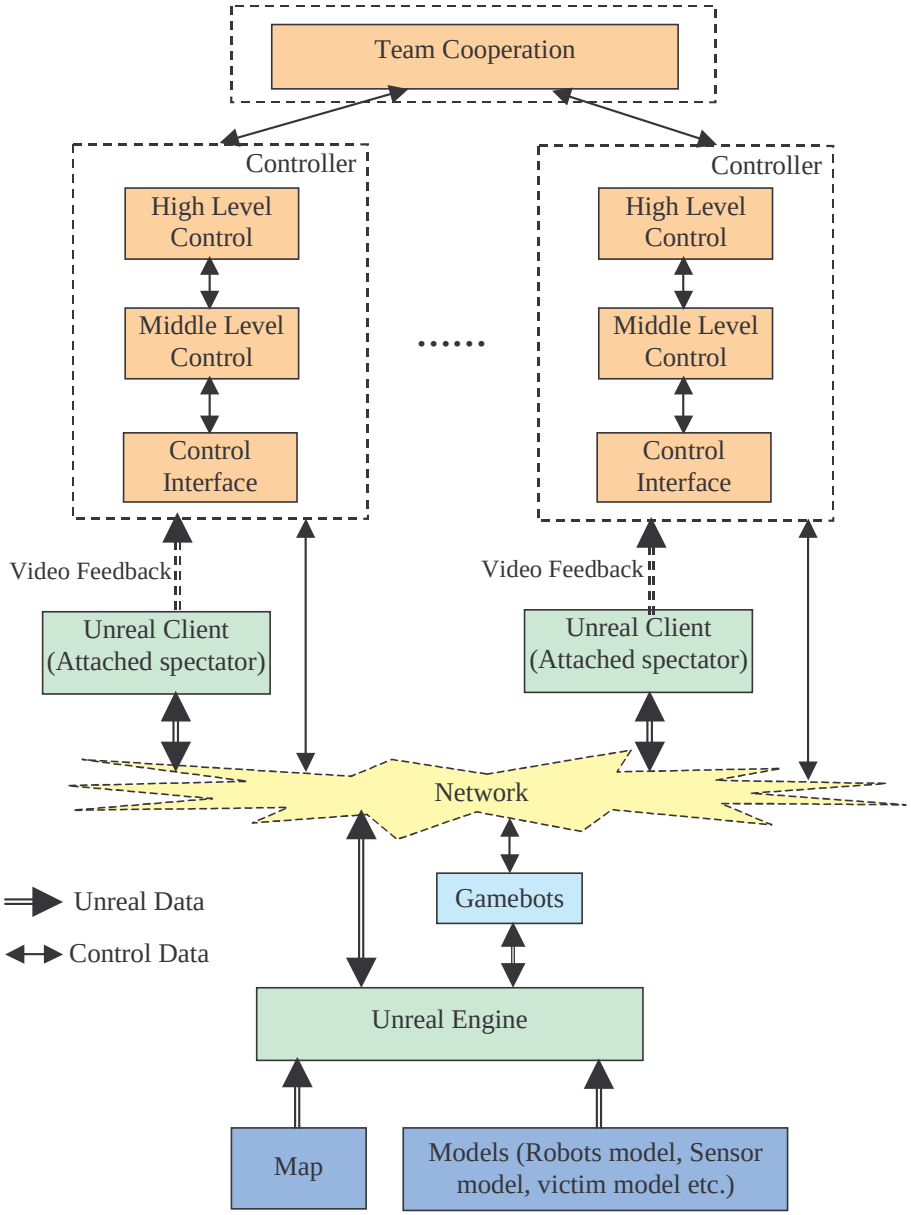

Figure 1 System Architecture

The system architecture is showed in figure 1. Below the dashed boxes is the simulator that provides interactive virtual environment for the users. The dashed box is the user side where you can use the simulator to do your research. The system uses 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. Unreal client renders the simulated environment. In the Unreal client, through changing the viewpoint, we can get the view of the robots in the environment. All the clients exchange data with the server through the network. The server side is called Unreal server. It includes Unreal engine, Gamebots, map and the models, such as robots model, victims model etc.. Unreal server maintains the states of all the objects on the simulator, responds to the data from the clients by changing the objects' states and send 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) Control client that controls the robots on the simulator.

2.1.1 Unreal engine

The Unreal engine used in the simulator is released by Epic Games (http://www.epicgames.com/) with Unreal Tournament 2003 (http://www.unrealtournament2003.com/). 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 physics engine, which is known as Karma engine, which 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 actor's behaviors. Unreal Editor is the 3D authoring tool comes with the Unreal engine to help developers build their own map, geometric meshes, terrain etc. For more information about Unreal engine, please visit the Unreal Technology page: http://unreal.epicgames.com/.

2.1.2 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 the 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.

2.1.3 Controller (Pyro)

Controller is the user side application that is used for your research, such as robotics study, team cooperation study, human robot interface 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 was created on the simulator, the controller listen the sensor data and send 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.

To facilitate the users and as an example of implementation, a Pyro plug-in is included in USARSim. With this plug-in, we can use Pyro to control the robot in the simulator. Pyro (http://pyrorobotics.org/) is a Python library, environment, GUI, and low-level drivers used for explore AI and robotics. More information about Pyro can be located from the following site: http://pyrorobotics.org/pyro/?page=PyroModulesContents. The details of the Pyro plug-in are described on section 8.

2.2 Simulator components

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

2.2.1 Environment simulation

Environment makes a very important role in simulations. It provides the context for the simulation and only with it, can the simulation make sense. USARSim provides simulated disaster environments in the Urban Search and Rescue (USAR) domain. Our environments are the 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.

We built all the virtual arenas from the AutoCAD model of the real arena. 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, stair, ramp etc.
- *Obstacles simulation*: that simulates the objects that can move and change their states. In addition, these objects also can impact the state of a robot. For example, they cam change a robot's attitude. These objects include bricks, pipes, rubbles etc.
- *Light simulation*: that simulates the light environment in the arena.
- *Special effects simulation*: that simulates the special stuff such as glasses, mirrors, grid fenders etc.
- *Victim simulation*: is the simulation of victims that can have their actions such as waving the hand, groaning, and other distress actions.

All the virtual arenas are built with Unreal Editor. With it, users can build their own environment. Details please read section 9.1.

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

Figure 6 Red Arena

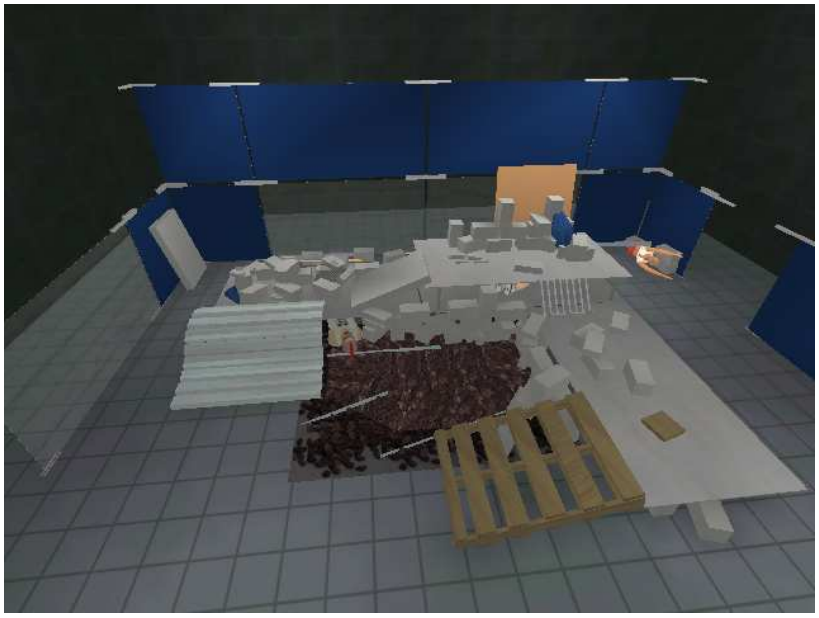

Figure 7 Simulated red arena

2.2.2 Sensor simulation

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

- Proprioceptive sensors

It includes battery state and headlight state.

- Position estimation sensors
  It includes location, rotation and velocity sensors.
- Perception sensors
  It includes sonar, laser and pan-tilt-zoom (ptz) camera.

All the sensors in USARSim are configurable. You can easily mount a sensor on the robot by adding a line into the robot's configuration file. When you mount the sensor, you can specify its name, type and the position where it's mounted and the direction it will face. For every kind of sensor, you also can specify its properties, such as, the maximum range of sonar, the resolution of laser and FOV (field of view) of camera. For more information about configuring a sensor please read section 6. Details of mounting a sensor on the robot please go to section 7.

2.2.3 Robot simulation

Using the rigid-body physics engine, Karma engine, embedded in Unreal Tournament 2003, 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 sensors, headlight 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 spin a specified angle. The first-order control lets the joint rotate under the specified spin speed. The second-order control applies the specified torque on the joint. The robot gets the control command through the Gamebots.

With this robot model, users can build a new robot without or only with a few of Unreal Script programming. For the steps of building your own robot, please read section 9.3.

In USARSim, five robots are already built for you. They are Pioneer robots: P2AT and P2DX, the Personal Exploration Rover (PER) built by CMU, the Corky built by the CMU USAR team and a typical four-wheeled car. These robots are explained later in section 7.

3 Installation

3.1 Requirements

Operating System: Windows 2000/XP or. Linux
Software: UT2003 and the 2225 patch
Optional requirements: If you want to use Pyro as the controller, you need Pyro 2.2.1.

3.2 Install UT2003 and the patch

3.2.1 Windows

1) Install UT2003.
2) Go to UT2003 website download the 2225 patch (http://www.unrealtournament.com/ut2003/downloads.php). And then double click the file to install the patch.

3.2.2 Linux

1) Install UT2003
   a. Copy the 'linux_installer.sh' in the third UT2003 cdrom to a temporary directory on the hard drive.
   b. Before running the script you need to tell the system where the CDROM is. You can do this by

      export SETUP_CDROM = */mnt/cdrom*

      change */mnt/cdrom* to wherever your cdrom mounts.
   c. Run the installer from the temporary directory:

      sh /tmp/linux_installer.sh

      and follow the prompts.
   d. The installer can be a little quirky when asking for disks. Basically when it asks for the disk, try all three. Sometimes it asked disk 1, but only continued when gave it disc 2.
   e. The game only works with Nvidia cards. You may need to install the NVIdia driver.
2) Install 2225 patch
   a. Download ut2003 2225 patch from http://www.unrealtournament.com/ut2003/downloads.php
   b. Restore the patch somewhere on your drive:
      tar -xvIf ut2003lnx_patch2225.tar.bz2
      or
      bzip2 -d ut2003lnx_patch2225.tar.bz2
      and then
      tar -xvf ut2003lnx_patch2225.tar
   c. Copy the files to override the file in ut2003 installation directory:
      cp -rfv ut2003-lnx-2225/* /to/where/ut2003/

3.3 Install USARSim

The installation is simple. You just unzip the files to the UT2003 installation directory. There is a testing control interface written in C++. If you don't want to install Pyro, you can copy USAR_UI to your machine and try it. USAR_UI only works on Windows. You can use it to send commands to USARSim and any message gotten from the Unreal server will be displayed in USAR_UI.

3.4 Install the controller (Pyro)

This step is optional. Install it only when you want to use Pyro to control USARSim.

3.4.1 Windows

Pyro is designed for Linux. Although Python, the development language used by Pyro, works under any system, Pyro uses some features only supported by Linux, such as Linux environment variable, 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) Following 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 restored Pyro, you need not run 'make' to compile it. Since it uses gcc, gmake to compile files, if you have none of them installed on your machine, 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 to USARSim. After you restore Pyro, you need to download and unzip pyro_win.zip to overwrite the files in the Pyro directory.

3.4.2 Linux

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

4 Run the simulator

4.1 The steps to run the simulator

Basically, running the simulator needs three steps.

1) Start Unreal Server
   Go to UT2003/system directory, and then execute:

   ucc server *map_name*?game=USARBot.USARDeathMatch –i=USARSim.ini

   where *map_name* is the name of the map. It can be DM-USAR_yellow (the yellow arena), DM-USAR_orange (the orange arena) or DM-USAR_red (the red arena).
2) Start Unreal Client
   Go to UT2003/system directory, and then execute:

   ut2003 *ip_address*?spectatoronly=true?quickstart=true

   where *ip_address* is the ip address of the server. If you run the server and client on the same machine, the ip address is 127.0.0.1.
3) Start the Controller
   After start the Unreal server, you can run your own application now.

Note: Only *start Unreal server once*. Sometimes, you may forget to stop the Unreal server, and then start another one. This will bring troubles to you. So make sure you only have one Unreal server on a machine.

After the Unreal client 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 picture 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 look over viewpoint.

Besides the step by step manually run 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 will you show you how to run USARSim manually and automatically.

4.2 Examples

There are two controllers in the USARSim package. We explain them in section 4.2.1 and section 4.2.2 separately.

4.2.1 The testing control interface

USAR_UI is a testing interface written by 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 came from the server to you. Follow the step 1 and 2 to start Unreal server and client. And then execute usar.exe. This will pop up a window. The usages of the interface are:

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. The spawning command looks like: "INIT {ClassName USARBot.USARCar} {Location 312,-600,-305}", where 'ClassName USARBot.USARCar' is the robot name. It can be USARCar, USARBc, P2AT, P2DX and Rover. The 'Location' is the initial position of the robots. You can refer the recommended start position in table 1 to decide the 'Location' parameter.
3) After add the robot to the simulator, you can try 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 the message.
4) You can also use joystick or keyboard+mouse to control the robot. To do it, 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 enable in Unreal, you need to disable it. So the system will not be confused. The ways of using joystick are:
   - Push joystick forward/backward will move the robot forward/backward.
   - Push joystick to left/right side will turn the robot to left/right.
   - Push POV button up/down will tilt the camera
   - Push 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 set the interface is 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.

4.2.2 Pyro

The Pyro plug-in embeds the Unreal server/client loading into it. 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 directory not a file. Pyro will automatically load 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.[1]
3) Click the 'Robot' button and select the robot you want to add on the plugins\robots\USARBot directory. You will see the robot is added in the virtual environment.
4) You should be able to control the robot using the 'Robot' menu now.
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.

More details about Pyro, please read section 8.

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

To *get control from UT2003*, pressing Esc can let you get window's control back.

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

5 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 robot in the USARSim virtual environment.

[1] In Linux KDE, UT2003 doesn't support switching focus to other applications. This is a known bug of UT2003. As a solution, we use launching UT2003 on another console to let user switch between UT2003 and other applications. When this UT2003 bug is fixed, we will move back to fire UT2003 and Pyro on the same console.

5.1 TCP/IP socket

As we mentioned before, Gamebots is the bridge between Unreal and the controller. It opens a TCP/IP socket for communication purpose. The IP address of the TCP/IP socket is the IP address of the machine runs the Unreal server. The default port number of the socket is 3000 and the maximum allowed connection 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 connection number. You can change or add (if you cannot find the parameters in the INI file) the parameters to the value you want.

5.2 The protocol

The communication protocol is the Gamebots protocol. All 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 space. For example, for "Location 100,200,300", the name is 'Location', the value is '100,200,300'. For the segment "Name Left Range 800.0", the names are 'Name' and 'Range', the values are 'Left' and '800.0'.

A message or command is constructed by a data_type and multiple segments. data_type and segments are separated by space.

5.3 Messages

Right now, we have two types of message. State message is the message reports the robot's state. Sensor message contains the sensor data.

- State message

  A state message looks like:

  STA {Time *t*} {Camera *pitch,yaw,roll*} {Attitude *pitch,yaw,roll*} {Location *x,y,z*} {Velocity *x,y,z*} {LightToggle *bool*} {LightIntensity *int*} {Battery *float*}

  Comment: We may need to change roll to zoom

  Where:

  {Time *t*}: '*t*' is the UT time in second. It starts from the time the UT server start works.

  {Camera *pitch,yaw,roll*}: The attitude of the camera. The values are

	pitch, yaw and roll angle in integer. 65535 equals to 360 degree.
{Attitude *pitch,yaw,roll*}:	The attitude of the robot. The values are the same as Camera's.
{Location *x,y,z*}:	Position of the robot. The values are positions in x, y, z axes in UT unit (UU).
{Velocity *x,y,z*}:	The velocity of the robot. The values are speeds in x, y, z axes direction in UU/s.
{LightToggle *bool*}:	Indicate whether the headlight has been turned on. 'bool' is true means turning on. False value means turning off
{LightIntensity *int*}:	Light intensity of the headlight. Right now, it always is 100.
{Battery *float*}:	'*float*' is the power state of the battery. Its range is 0.0~100.0. 100.0 means the battery is fully charged.

Comment: Make sure battery sensor is added into USARSim

Example: STA {Time 8.29} {Camera 63541,32768,0} {Attitude 65533,0,0} {Velocity -1.59,0.00,-1.95} {LightToggle False} {LightIntensity 100} {Battery 80.0}

- Sensor message
  - Sonar Sensor

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

    Where:
    '{Name *string* Range *float*}' is the sensor data. '*string*' is the sensor name, '*float*' is the range value in UU.
    Example: SEN {Type Range} {Name Right Range 144.69} {Name Left Range 240.19}

  - Laser Sensor

    SEN {Type RangeScanner} {Name *string*} {Location *x,y,z* Rotation *pitch,yaw,roll*} {Data *r1,r2,r3*…}

    Where:

{Name *string*}	'*string*' is the sensor name.
{Location *x,y,z* Rotation *pitch,yaw,roll*}	'*x,y,z*' is the sensor position in UU. '*pitch,yaw,roll*' is the rotation of the sensor. NOTE: the rotation is the absolute rotation.
{Data *r1,r2,r3*…}	'*r1,r2,r3*…' is a serious of range values.

    Example: SEN {Type RangeScanner} {Name Scanner1} {Location 426.28,-599.95,-474.12 Rotation 15136,16384,16} {Data

266.55,359.43,364.46,377.57,400.25,1000.00,1000.00,1000.00,916.72,852.27,811.95,790.88,786.71,799.01,829.10,880.44,739.17,618.73,541.64,490.64,457.05,436.19}

- Human Motion Detection

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

  Comment: Check this with CMU robot

  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*}

  Comment: We may need to change it to frequency

  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}

5.4 Commands

The supported commands are:

- Add a robot to UT world:

  INIT {ClassName *robot_name*} {Location *x,y,z*}

  Where:

  {ClassName *robot_name*} '*robot_name*' is the class name of the robot. It can be USARBot.USARCar, USARBot.USARBc, USARBot.P2AT, USARBot.P2DX and USARBot.Rover.
  {Location *x,y,z*} '*x,y,z*' is the stat position of the robot in UU. For different arena, we need different position. The recommended positions are listed on table 1.

Table 1 Recommended start position for the arenas

Arena	Recommended Start Position
Yellow	1200,345,-450
Orange	312,-600,-305
Red	200,600,-450

Example: INIT {ClassName USARBot.USARCar} {Location 312,-600,-305} will add a four wheels robot.

- Control the Robot:

  There are two kinds of control command. The first kind controls the left and right side wheels. The second kind controls a specified joint of the robot.

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

    {Left *float*} '*float*' is spin speed for the left side wheels. Its range is –1.0~1.0. Positive value means moving forward.
    {Right *float*} '*float*' is spin speed for the right side wheels. Its range is –1.0~1.0. Positive value means moving left.
    {Light *bool*} '*bool*' is whether turn on or turn off the headlight. The possible value is True/False.
    {Flip *bool*} If '*bool*' is True. This command will flip the robot.

  Example: DRIVE {Left 1.0} {Right 1.0} will drive the robot moving forward.
  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 {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 integer. 63356 means 360 degree. For first-order control, it's the spin speed in integer/second. For second-order control, it's the torque.

  Example: DRIVE {Name LeftFWheel} {Steer 16384} will steer the left front wheel 90 degree.
  DRIVE {Name LeftFWheel} {Order 1} {Value 2000} will make the left front wheel spin at 2000*360/65535 degree/second.

- Control the camera:

  CAMERA {Rotation *pitch,yaw,roll*} {Order *int*} {Zoom *int*}
  Where:
    {Rotation *pitch,yaw,roll*} '*pitch,yaw,roll*' is the rotation angle of the camera. It could be relative value or absolute value. This is set on the robot configuration file.

{Order *int*} It's the as the order parameter of DRIVE command. Without specifying the parameter, the system treats it as '0'.

{Zoom *int*} '*int*' is the zoom value. Positive value means zoom in. Negative value means zoom out.

Example: CAMERA {Rotation 1820,0,0} will tilt the camera 10 degree if the robot uses relative value. If the robot uses absolute value, it will tilt the camera to 10 degree.
CAMERA {Zoom 100} will zoom in the camera.

- Control the sensor:

  SENSOR {Scan true}
  This command is used to manually control range scanner. For example, you drive the robot to some place and then want to collect the range data. '{Scan true}' will trigger the range scanner, such as laser, to scan its environment once. After it finishes, it will stop and go back to its initial position.

6 Sensors

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

In USARSim, except the state sensor, all the other sensors can add noise and apply distortion to their data. By changing corresponding parameters, we can get different quality sensor data.

In this section, we will explain how the sensor works and how to configure it. To learn how to build your own sensor, please read section 9.2.

6.1 State sensor

6.1.1 How the sensor works

The State sensor reports the robot's state. Basically, it just checks the robot's state in Unreal engine and then sends it out. So all the data in the state sensor data package should be treated as the truth data.

6.1.2 How to configure it

None. We needn't configure it.

6.2 Sonar sensor

6.2.1 How the sensor works

Sonar sensor is used to detect distance. In USARSim, sonar sensor is simulated by emitting a line from the sensor position along the direction of the sensor in Unreal world. The first point met by the line is the hit point. The distance between the hit point and the sensor is the returned range value. 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 sonar sensor.

6.2.2 How to configure it

The range sensor configuration in the usar.ini file looks like:

```
[USARBot.RangeSensor]
HiddenSensor=true
MaxRange=1000.000000
Noise=0.05
OutputCurve=(Points=((InVal=0.000000,OutVal=0.000000),(InVal=1000.0
00000,OutVal=1000.000000)))
```

Where

HiddenSensor	This Boolean value is used to indicate whether the sensor will be visually showed 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.
MaxRange	It is the maximum distance that can be detected.
Noise	It is the relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data
OutputCurve	It's the distortion curve. It is constructed by a serial of points that describe the curve.

6.3 Laser sensor

6.3.1 How the sensor works

Laser sensor is very similar to sonar sensor. In USARSim, laser sensor is treated as a serial of sonar sensors. The data is obtained by rotating the sonar sensor from the start direction to the end direction step by step. The step interval is calculated from the resolution. Laser sensor can work in two modes. In the automatic mode, laser sensor automatically scans data every specified time interval. In manual mode, laser sensor only work when it gets a scan command. Every time it gets a command, it only scan once.

6.3.2 How to configure it

The laser sensor configuration in the usar.ini file looks like:

```
[USARBot.RangeScanner]
HiddenSensor=False
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=1000.000000,OutVal=1000.000000)))

Where

HiddenSensor	This boolean value is used to indicate whether the sensor will be visually showed 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.
MaxRange	It is the maximum distance can be detected.
ScanInterval	It is the time interval between scanning used in automatic mode.
Resolution	It's the scan resolution, the step length of rotating from start direction to the end direction. The unit is integer. 65535 means 360 degree.
ScanFov	It's the scan range in integer.65535 means 360 degree.
bPitch	A Boolean value that indicates the scan plan. True means scanning in the tilt plan (x-z plan).
bYaw	A Boolean value that indicates the scan plan. True means scanning in the pan plan (x-y plan).
Noise	It is the relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data
OutputCurve	It's the distortion curve. It is constructed by a serial of points that describe the curve.

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

6.4 Sound sensor

6.4.1 How the sensor works

Sound sensor detects the victims' sound. In USARSim, sound sensor finds all the sound sources and calculates the source that hears to be the loudest at the robot's location. The loudness decreases in the square of the distance.

6.4.2 How to configure it

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

[USARBot.SoundSensor]
HiddenSensor=True
Noise=0.05

OutputCurve=(Points=((InVal=0.000000,OutVal=0.000000),(InVal=1000.0
00000,OutVal=1000.000000)))

Where

HiddenSensor	This boolean value is used to indicate whether the sensor will be visually showed 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.
Noise	It is the relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data
OutputCurve	It's the distortion curve. It is constructed by a serial of points that describe the curve.

6.5 Human-motion sensor

6.5.1 How the sensor works

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

6.5.2 How to configure it

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

[USARBot.HumanMotionSensor]
HiddenSensor=True
MaxRange=1000
FOV=60
Noise=0.1
OutputCurve=(Points=((InVal=0.000000,OutVal=0.000000),(InVal=1000.0
00000,OutVal=1000.000000)))

Where

HiddenSensor	This boolean value is used to indicate whether the sensor will be visually showed 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.
MaxRange	It's the maximum detecting range in UU.
FOV	It's the filed of view of the sensor in integer. 65535

	meams 360 degree.
Noise	It is the relative random noise amplitude. With the noise, the data will be data = data + random(noise)*data
OutputCurve	It's the distortion curve. It is constructed by a serial of points that describe the curve.

7 Robots

All robots in USARSim have a chassis, multiple wheels, a bunch of sensors, a camera, and headlight. The robots are configurable. You can specify which sensor and where the sensor is mounted. You also can configure the properties of the robots, such as the battery life and the frequency of data sending 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 it.

7.1 P2AT

7.1.1 Introduction

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

In summary, P2AT has:

- Four wheels
- Skid-steer
- Size: 50cm x 49cm x 26cm
- Wheel diam: 21.5cm

In our simulation, it's equipped with

- PTZ camera
- Front sonar ring
- Rear sonar ring

Figure 9 Real P2AT robot

Figure 8 Simulated P2AT

7.1.2 Configure it

The whole P2AT robot configuration can be found in the section [USARBot.P2AT] of usar.ini file. The following list the parameters you may need to change. Other parameters please refer section 9.3.2.

```
[USARBot.P2AT]
msgTimer=0.200000
bAbsoluteCamera=true
Sensors=(ItemClass=class'USARBot.RangeSensor',ItemName="F1",Positio
n=(Y=-35.11,X=20.18,Z=13),Direction=(Pitch=0,Yaw=-16384,Roll=0))
...
Sensors=(ItemClass=class'USARBot.RangeSensor',ItemName="R8",Positio
n=(Y=-35.11,X=-20.18,Z=13),Direction=(Pitch=0,Yaw=-16384,Roll=0))
Camera=(ItemClass=class'USARBot.Sensor',ItemName="Camera",Parent="
CameraTilt",Position=(Y=0,X=13,Z=0),Direction=(Pitch=0,Yaw=0,Roll=0)
)
CameraFov=50
```

Where

msgTimer	It's the time interval between two messages sending.
bAbsoluteCamera	It indicates whether the camera control command uses an absolute value or not.
Sensors	It's the sensor 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
Position	The mounting position relative to the geometric center of the robot.
Direction	The facing direction of the sensor relative to the robot.
Camera	It's the camera mounted on the robot. It uses the same structure of the sensor.
CameraFov	It is the field of view of the camera in degrees.

7.2 P2DX

7.2.1 Introduction

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

In summary, P2DX has:

- Two wheels
- Differential steering
- Size: 44cm x 38cm x 22cm
- Wheel diam: 19cm

In our simulation, it's equipped with

- PTZ camera

- Front sonar ring

Figure 11 Real P2DX robot

Figure 10 Simulated P2DX

7.2.2 Configure it

It's the same as P2AT.

7.3 RER

7.3.1 Introduction

PER is the Personal Exploration Rover built by CMU for education and demonstration purpose. The robot uses rocker-bogie suspension system to adapt to terrain. It has a pan-tilt camera mounted on it. For details about PER please visit the PER home page: http://www-2.cs.cmu.edu/~personalrover/PER/

In summary, PER has:

- Six wheels. Four drive wheels and two omnidirectional wheels.
- Double Ackerman steering
- Rocker-Bogie suspension system
- Differential body pose adjusting
- A pan-tilt camera that can take 360 degree panorama

In USARSim, we use classname USARBot.Rover to represent PER.

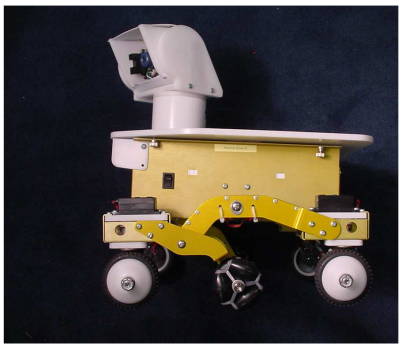

Figure 13 Real PER robot

Figure 12 Simulated PER

7.3.2 Configure it

It's the same as P2AT.

7.4 Corky

7.4.1 Introduction

Corky is the robot built by CMU USAR term. Its features are:

- Two wheels.
- Differential steering
- A pan-tilt camera
- 5 range sensors

In USARSim, an additional headlight is added to the robot. This robot model is our first Karma vehicle model. It's designed for this specified robot. To archive speed control, PID controllers are built for both wheels of Corky.

7.4.2 Configure it

The configuration of Corky in usar.ini file looks like:

Figure 15 Real Corky

Figure 14 Simulated Corky

```
[USARBot.USARBc]
msgTimer=0.200000
bSpeedControl=True
bAbsoluteCamera=False
Sensors=(SensorClass=class'USARBot.RangeSensor',SenName="Front",Pos
ition=(X=-80,Y=0,Z=50),Direction=(Pitch=0,Yaw=32768,Roll=0))
…
Sensors=(SensorClass=class'USARBot.RangeSensor',SenName="Right",Pos
ition=(X=0,Y=-40,Z=50),Direction=(Pitch=0,Yaw=-16384,Roll=0))
Kp=0.2
Ki=0.8
Kd=0.0
MinOut=-20.0
MaxOut=20.0
```

Where:

msgTimer	It's the time interval between two messages sending.
bSpeedControl	It indicates whether Corky uses speed control. Set to false, the value in the control command is interoperated as torque. Otherwise, the value is treated as speed.
bAbsoluteCamera	It indicates whether the camera control uses absolute value or not. Set to false, the value in the control command is interoperated as absolute value.
Sensors	It's the sensor mounted on the robot. The structure of sensor mounting is:
SensorClass	The sensor class or the type of the sensor.
SenName	The name assigned to the sensor
Position	The mounting position relative to the geometric center of the robot.
Direction	The facing direction of the sensor relative to the robot.
Kp	The proportional parameter of the PID control. Both wheel use the same parameter.
Ki	The integral parameter of the PID control.
Kd	The derivative parameter of the PID control.
MinOut	The minimum output torque of the motor engine.
MaxOut	The maximum output torque of the motor engine.

7.5 Four-wheeled Car

7.5.1 Introduction

Very similar to Corky except it's a four-wheeled vehicle. It also has a camera, a headlight, and four range sensors mounted on the front, back and left, right side.

7.5.2 Configure it

It's the same as Corky.

Figure 16 Simulated Foue-wheeled Car

8 Controller – Pyro

The whole description of Pyro can be found on the Pyro Curriculum: http://pyrorobotics.org/pyro/?page=PyroModulesContents. In this section we only explain the contents involved in USARSim.

8.1 Simulator and world

The USARSim simulator loader is put in the 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 Unreal server with the 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 on 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:\ut2003
LoadServer=true
IP=127.0.0.1
Port=3000
Map=DM-USAR_yellow
Location=1200,345,-450
```

Where:

Path	The path you install UT2003.
LoadServer	A Boolean variable indicates whether the loader needs to start Unreal server. If you already started Unreal server or you want to run 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" on BotAPI.ini file in the Unreal system directory (more details see section 5.1).
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_res.
Location	The initial position where the robot will be spawned. Please refer Table 1 to decide the values you want.

8.2 Robots

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

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

1) Creating connection with Unreal server

2) Sending commands to Unreal sever.

3) Listening and parsing messages from Unreal server.

In 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 special 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 Pyro GUI to add a robot into the USARSim virtual environment.

8.3 Services

To help user has a good sense about the sensors, some services are added to visualize the sensor data. These sensor visualizations are modified from the visualization module of PyPlayer: Python Client for Player/Stage (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 codes for these services are in robots\USARBot__init__.py file. The supported sensors are:

- Sonar

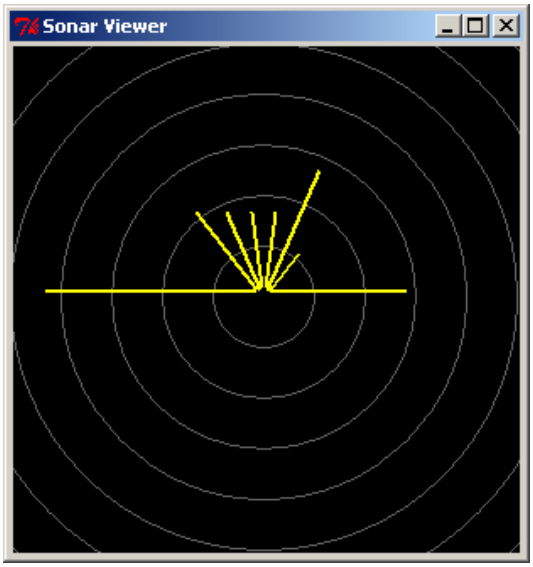

Figure 17 Sonar visualization

- Laser

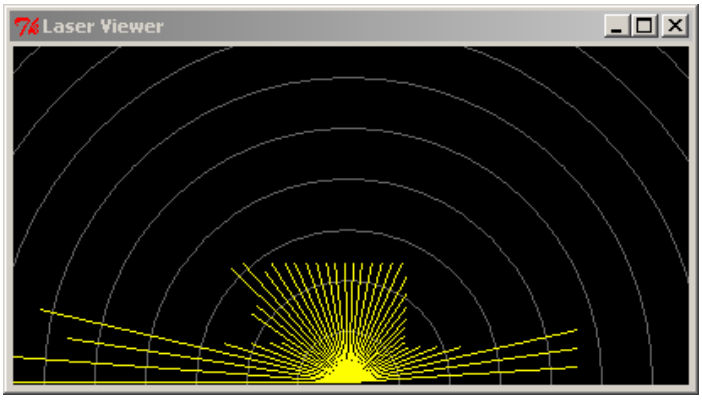

Figure 18 Laser visualization

- PTZ Camera

Comment: Need to add Zoom and camera pose visualization

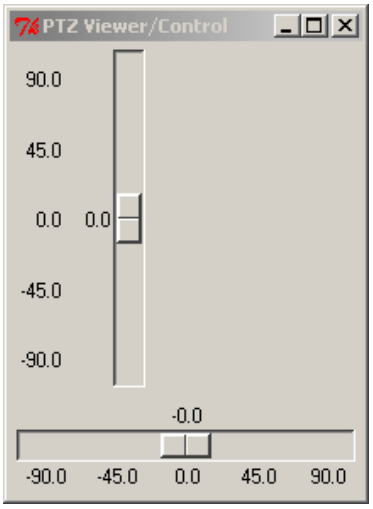

Figure 19 PTZ Camera viwer and controller

8.4 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.

9 Advanced User

This section is for the advanced users who want to build their own 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 ut2003.ini file that locates on the Unreal system directory. Adding the following lines to the corresponding sections in ut2003.ini will let Unreal engine recognize our own model. With this modification, we can compile and use our models in Unreal Editor.

```
[Engine.GameEngine]
ServerPackages=BotAPI
ServerPackages=USARBot

[Editor.EditorEngine]
EditPackages=BotAPI
EditPackages=USARBot

[UnrealEd.UnrealEdEngine]
EditPackages=USARBot
```

NOTE: You need to modify ut2003.ini before you build your own models.

9.1 Build your arena

An arena is an Unreal map. It includes geometric model and the 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 object only works well with static meshes. In addition, static meshes can accelerate 3D graphic rendering.

NOTE: All the meshes must be *static mesh*. Karma engine only works well with static meshes.

When you build a new arena, there are three things you may need to do: 1) building geometric model, 2) simulate some special effects and 3) adding objects such as obstacles and victims into the arena. The three things are explained in the following sections.

9.1.1 Geometric model

We have two options to build geometric model. One is importing an existing model into Unreal. Another is building the model by hand in Unreal. After building the model, we need to transfer it into static mesh.

9.1.1.1 Import the existed model

The basic idea of importing model is converting your model into a format that Unreal Editor can read in. The file formats that supported by 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 the text file holds a text list of Unreal map objects.

The details about how to import a 3D model are described in the document:

UDN: Converting CAD data into Unreal (http://udn.epicgames.com/Content/CADtoUnreal).

9.1.1.2 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 'General Editor' category in UDN contents documents all the details of modeling with Unreal Editor. The 'Topics On Mapping' (http://wiki.beyondunreal.com/wiki/Topics_On_Mapping) lists the topics involved mapping in Unreal Wiki.

9.1.2 Special effects

Most of the special effects are obtained by applying special materials. Please read the UDN: Material Tutorial (http://udn.epicgames.com/Content/MaterialTutorial) to have a sense of what an Unreal material is.

The grid fender effect 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 gray alpha-channel will give us semi-transparent effect. Using shaders 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, in the camera, into the place to fake the mirror effect. The idea comes from Unrealops (http://unrealops.com). The tutorial of adding a mirror can be found at: Security Camera Tutorial (http://unrealops.com/modules.php?op=modload&name=Reviews&file=index&req=showcontent&id=24). According to the author, this approach doesn't work online. To fix this shortcoming, a customized CameraTextureClient named myCameraTextureClient is created in USARSim. Replace all the CameraTextureClient by myCameraTextureClient in the tutorial, will give us 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

9.1.3 Obstacles and Victims

To get high fidelity simulation, we recommend using Karma objects as the obstacles. The example of adding Karma objects in a map can be found at UDN:

Karma Colosseum
(http://udn.epicgames.com/Content/ExampleMapsKarmaColosseum).

Victims are another type of objects we may need to put into the map. Victims are the special objects that can implement some actions. The victim model built in USARSim can be loaded from 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 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 is the parameters that specify the victim's action. 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 degree) of the segment.
FinalRotation	The final rotation (pitch, yaw and roll in integer. 65535 means 360 degree) 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 name. You can use the 'Animations' browser to view the bone name. An example in showed in Figure 20.

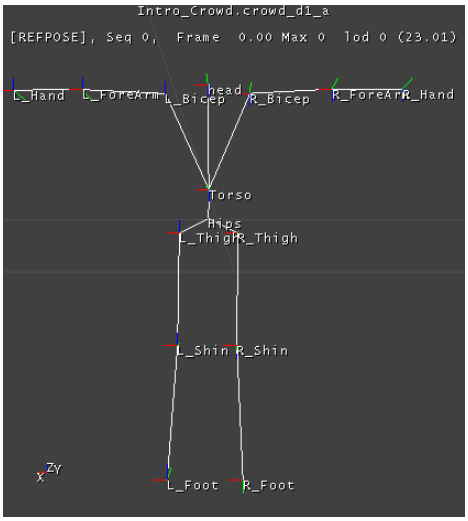

Figure 20 Skeletal bones name

More details about skeletal mesh, please visit:

UDN: AnimBrowserReference (http://udn.epicgames.com/Content/AnimBrowserReference)

UDN: UWSkelAnim2 (http://udn.epicgames.com/Technical/UWSkelAnim2)

After you set the actions, the victim will not move immediately. In Unreal Editor, every thing is static. To let them to be active, you need to play the map.

As we know, there is a bug in 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, shines and feet.

9.2 Build your sensor

Before build your sensors, you need to understand Unreal Script and the client/server architecture of Unreal engine. The following resources may be helpful to you:

UDN: UnrealScriptReference (http://udn.epicgames.com/Technical/UnrealScriptReference)

UnrealWiki: UnrealScript Topics (http://wiki.beyondunreal.com/wiki/UnrealScript)

Unreal Networking Architecture (http://unreal.epicgames.com/Network.htm)

9.2.1 Overview

In USARSim, all sensors are inherited from Sensor class. Sensor class defines the interface that the robot model can interact with. We use hierarchical architecture to build the sensors. The hierarchy chart is showed below.

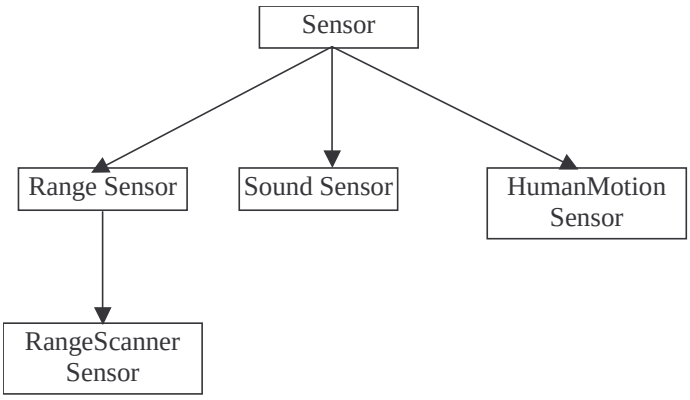

Figure 21 Sensor Hierarchy Chart

9.2.2 Sensor Class

The Sensor class is the ancestor of all the sensor classes. It takes care creating the sensor, mounting itself on the robot and returning sensor data to the robot. The details about Sensor classes are explained below:

Attributes:

```
var string SenName; // the sensor name
var string SenType; // the sensor type
var config bool HiddenSensor; // variable indicts whether show the
                              // sensor in Unreal
var config InterpCurve OutputCurve; // the distortion curve
var config float Noise; // the random noise amount
var vector myPosition; // the mounting position
var rotator myDirection; // the mounting direction
```

Methods:

```
function SetName(String SName) // set the sensor name
function Init(String SName, Actor parent, vector position, rotator
direction) // mounting the sensor
function String GetData() // the interface that send sensor data to robot
function String GetType() // return the sensor type
```

9.2.3 Writing your own sensor

Your sensor should extend from Sensor class. Usually, you only need to override the GetData method. In this method you return the sensor data in string. When you generate the sensor data, you also may need the variable myPosition and myDirection. Although these are Noise and OutputCurve parameters in Sensor class, it does nothing about the noise data simulation and data distortion simulation. It's your responsible to simulate them in the GetData method.

9.3 Build your robot

Usually, building a robot involves a lot of programming, deeply understanding Unreal network architecture and the background knowledge of mathematic and mechanic. It takes you a lot of time in programming and debugging. To facilitate the robot building, we build a general robot model to help users build their own robot. In the robot model, every robot are constructed by:

- Chassis: the chassis of the robot.
- Parts: the mechanic parts, such as tire, linkage, camera frame etc., that construct the robots.
- 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, headlight etc, attached to the robot.

A chassis can connect to multiple parts through joints. However, one part only can has one joint. The attached items can be attached to either chassis or part. The chassis or part can has multiple attached items.

The working flow of building a robot is building geometric model for all the objects that construct the robot. Then create a new robot class that extends from KRobot. In this class you set the physical attributes of the robot instead of program. And you also need to configure how the chassis, parts and auxiliary items are connected to each other. At last, if you want to add some new features not included in the robot model, you will do some programming work.

9.3.1 Step1: Build geometric model

Essentially, this step is the same as building your own arena. Please refer section 9.1.1 to learn how to build 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. It's the same for Y and Z axes. Incorrect axis will bring you incorrect pitch, yaw and roll angle.

NOTE: Make sure the geometric model has the correct x-axis and y-axis. This will affect the attitude data.

9.3.2 Step2: Construct the robot

9.3.2.1 Create the robot class

At 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
{
  //propertise
}
```
where robot_class_name is the name of your class.

9.3.2.2 Prepare the attributes and objects used for you robot

In the defaultproperties block of the class, you can set the attributes of the robot. The attributes are:

MotorTorque	The default motor torque. Default value is 20.
ChassisMass	The mass of the chassis 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. Details please read the UDN: KarmaReference (http://udn.epicgames.com/Content/KarmaReference).

Similar to chassis, every part can has its own Karma physical parameters. Multiple parts can share the same KarmaParams object. These KarmaParams can be defined here.

Besides these properties, you also can set steering 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
SteerPropGap	The proportional gap used for steer speed control.	1000.0
SteerTorque	The torque applied to the steer	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	5.0
TireLateralFriction	Lateral friction of the tire	2.5
TireRollSlip	Maximum first-order (force ~ velocity)	0.06

	slip in tire direction	
TireLateralSlip	Maximum first-order (force ~ velocity) slip in sideway direction	0.04
TireMinSlip	The minimum slip in both direction	0.001
TireSlipRate	The amount of slip per unit of velocity	0.007
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.

Before we connect all the parts together, we still need to prepare the parts. There are two kinds of part.

One is KDPart that is a normal Karma object. It's used to simulate the parts like linkages, camera frames etc. Usually, you needn't build your own part class.

Another kind of part is tire. All tires extend from KTire. There is a limitation in the general robot model that it cannot set the tire collision from the configuration. The Karma collision use the scale parameter statically defined in the defaultproperties block. You can change the DrawScale after the object is created. However, this will not affect the collision used by Karma. To my knowledge, there is no way to dynamically change the scale used by Karma. Therefore, for every tire uses different static mesh scale, you must build a tire class for it. In the tire class, you define the DrawScale in the defaultproperties block. All the tire classes extend from KTire or BCTire. The only difference between KTire and BCTire is that BCTire defined some parameters such as the collision and KarmaParams parameters for you. If your tire class extends from BCTire, you needn't define them again. In the class you only need to add a line in defaultproperties block to define the DrawScale.

NOTE: You cannot set Tire scale in the configuration. It doesn't really work. The solution is creating your own tire class that extends from BCTire. In the class you specify the scale in the defaultproperties block.

9.3.2.3 Connect the parts

After we setup all the attributes, 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 car-wheel joint. A car-wheel joint connects two parts by two axes. One is the spin axis (hinge axis in Figure 22) that the part can spin around. Another is the steering and suspension axis (Steering Axis in Figure 22) that the part can steer around and travel along.

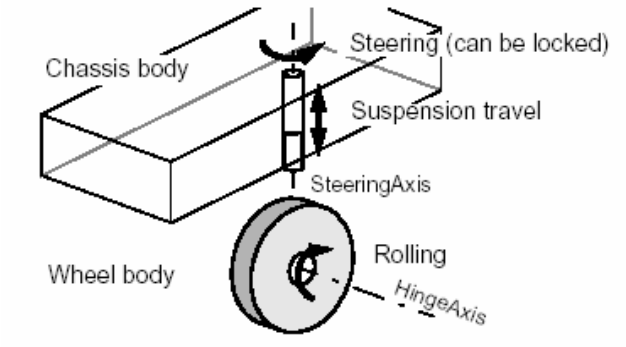

Figure 22 Car wheel joint

The part-joint pair is a structure defined below:

```
struct JointPart {
    // Part
    var() name                  PartName;
    var() class<KActor>         PartClass;
    var() StaticMesh            PartStaticMesh;
    var() KarmaParamsRBFull     KParams;
    var() float                 DrawScale;
    var() vector                DrawScale3D;
    // Joint
    var() class<KConstraint>    JointClass;
    var() bool                  bSteeringLocked;
    var() bool                  bSuspensionLocked;
    var() float                 BrakeTorque;
    var() name                  Parent;
    var() vector                ParentPos;
    var() rotator               ParentRot;
    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.
PartStaticMesh	The static mesh of the part
Kparams	The KarmaParams you defined in last section for the part.
DrawScale	The scale of the static mesh

DrawScale3D	The scale along X, Y and Z axes of the static mesh
JointClass	The joint's class name. It should be: class'KCarWheelJoint'
bSteeringLocked	Indicates whether steering is locked.
bSuspensionLocked	Indicates whether suspension is locked.
BrakeTorque	The brake torque applied when we brake the joint.
Parent	The part or chassis that is part is connecting. NOTE: the part must have already been defined.
ParentPos	The position where the joint connect the parent.
ParentRot	The joint rotation relative to the parent. Right now, we don't use it.
ParentAxis	The steering axis relative to the parent
ParentAxis2	The spin axis relative to the parent
SelfPos	The position where the joint connect the part
SelfAxis	The steering axis relative to the parent
SelfAxis2	The spin axis relative to the parent

The order 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 usar.ini file (you may need to create the robot section by yourself). Using usar.ini file, you needn't compile your class after you changed something. However, there is one issue you need to know:

The KarmaParams must be defined in the defaultproperties block. And it must be used in the defaultproperties block at least once. Only with this, can you use the KarmaParams in the INI file. Otherwise, Unreal cannot recognize it.

NOTE: Define KarmaParams in the defaultproperties block and use it in the block at least once.

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 mechanic world, improper mechanic structure can cause the robot to be unstable. When you create the robot, make sure your geometric model is correct. Especially, you need to check the model whether it has the correct mass distribution. In some case, you may need to specify the mass center offset in the KarmaParams. When you robot is unstable, try to add the part one by one. This can help you figure out which part causes the problem.

TIP: Specify the mass center offset in the KarmaParams can help you simulate the mass distribution.

9.3.2.4 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 rotator         Direction;
};
```

where

ItemClass	The class used to create the item
Parent	The object where 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

9.3.3 Step3: Customize the robot (Optional)

After finished the previous two steps, you robot should work now. You should be able to use the DRIVE command to control every joint and you also can 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. 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 control pattern. In the robot class, you can transfer the left, right parameters into a serial of joint control parameters to

control the wheels. This can be reached by override 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. ‘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 command are came from Gamebots. A robot connects with Gamebots through its controller whose class is RemotBot. Every RemotBot is associated with a BotConnection that keeps listening to TCP/IP socket and parsing the incoming commands. Once a new command is coming, BotConnection realizes it and gets the value in the command. Then it sets the corresponding variable in RemotBot to the coming new value. In your robot class, you only need to check the RemotBot’s variable to get the command data.

In summary, to add a new command:

1) Add a new variable in RemotBot to store the command’s data.
2) In BotConnection, add your code into the ProcessAction function to interpret your command and store it in the RemotBots’s variable.
3) In your robot class, check the RemoteBot’s variable to get the command and do something you want.

3) Maintain the robot’s state by yourself

Some robots may have special state to maintain, for example, the following wheel of 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 PER. PER’s chassis is controlled by a differential that force the chassis’s pitch angle is always the average angle of 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 and Rover as example to learn how to maintain your own state.

At last, besides the three aspects mentioned above, obviously, you con do anything you want in your robot class.

9.4 Build your controller

The client/server architecture makes it's easy to build your own control client. You only need to follow the communication protocol.

10 Bug report

mailto:jiw1@pitt.edu

11 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 edited this document.