USARSim & MOAST

highly realistic simulation and control for multi robot systems

Orlando-FL
ICRA 2006

Organizers

- Stefano Carpin
  International University Bremen, Germany
- Mike Lewis
  Univ. of Pittsburgh, USA
- Stephen Balakirsky, Adam Jacoff
  NIST, USA

Tutorial outline

1. USARSim & MOAST: high level overview

Coffee break

2. Low level details

Lunch Break

2. USARSim hands on

Coffee break

4. MOAST hands on

Session 1: high level overview

- USARSim overview
- Client server structure
- Available sensors and actuators
- The Robocup Rescue Virtual Robots competition
- Some validation results
- MOAST (Mobility Open Architecture Simulation and Tools) overview

What is USARSim?

- High-fidelity multi-robot simulator developed on top of an existing commercial game engine
  - High performance physics and 3D rendering
- Originally conceived as tool for Urban Search and Rescue (USAR), it has a much broader breadth
- Initially developed at Univ. Pittsburgh, CMU, & NIST. Turned open source in 2005

Possible applications

- Basic research in
  - Autonomous robotics
    - Cooperation
  - Human Robot Interfaces
- Fast prototyping of robot software
  - The same code runs both in simulation and on real robots
- Robotics education

Goal

- Ideal scenario:
  Design, develop and debug your code in simulation. Then move it to a real system without changing a single line of code.
- Simulate:
  - Robots
  - Environments
  - Sensors
  - Actuators

Components

- The simulator itself
  - Game engine
  - USARSim on top
- Editing application for modeling new environments and new robots
  - Geometric aspects (shape, texture, etc)
- A host scripting language to add new features to robots, sensors, actuators

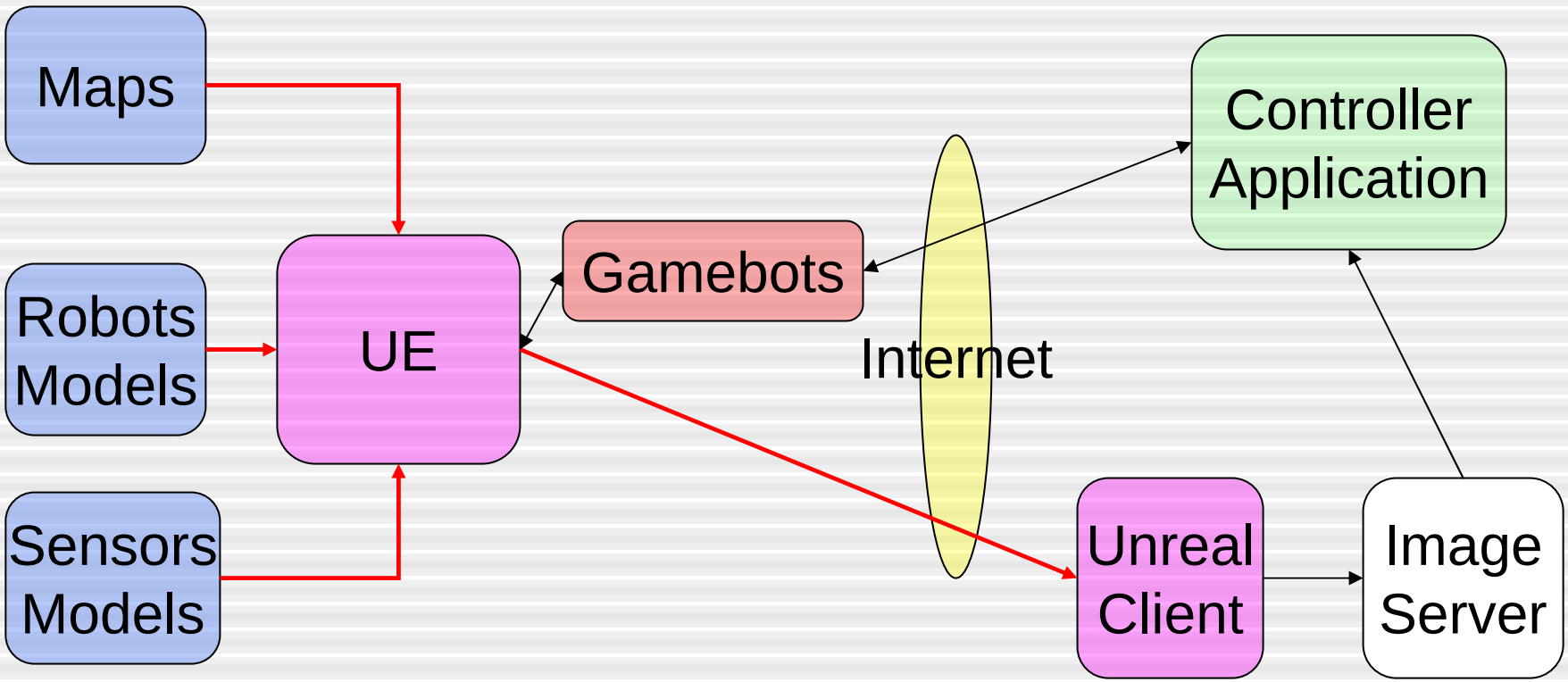
About the game engine

- UnrealEngine (UE) (www.epicgames.com)
  - ☺ Superior performance
  - ☹ Not open source
    - The engine is not open source but the game is…
- Modest cost (<50$)
- Runs on multiple platforms (Linux, Win,…)
- Wide network of additional components developers, newsgroups, etc.

Interacting with the game engine: Gamebots

- *Gamebots* opens a bidirectional communication channel between UE and external applications www.planetunreal.com/gamebots
- TCP/IP based communication
- USARSim builds on top of Gamebots so that external programs can control characters inside the simulated environment

Architecture overview



Components

- UE: simulates the environment
  - As described by maps, sensors and actuators
- Unreal Client: renders the scene
- ImageServer: auxiliary USARSim component providing controller applications images from simulated cameras (JPG format, different resolution)
  - Needed only if you process images
- Controller application: *your* code driving the robot around
  - Any language capable of opening a TCP/IP socket will do

Components (con’d)

- Environment
  - Geometric properties
  - Lights
  - Textures
- Sensors & actuators
  - Location, velocity, sonar, laser, touch sensor, PTZ camera, …
  - Noise can be added and tuned

A robot in an environment

Red arena

Available robots

- Commercial platforms
  - P2AT, P2DX, ATRV-Jr
- Research robots
  - PER, Corky (CMU)
  - Contributed:
    - Papagoose (IUB)
    - Tarantula (Univ. Freiburg)
    - Your robot should show up here…
- Forthcoming:
  - AIBO
  - humanoids

Available robots

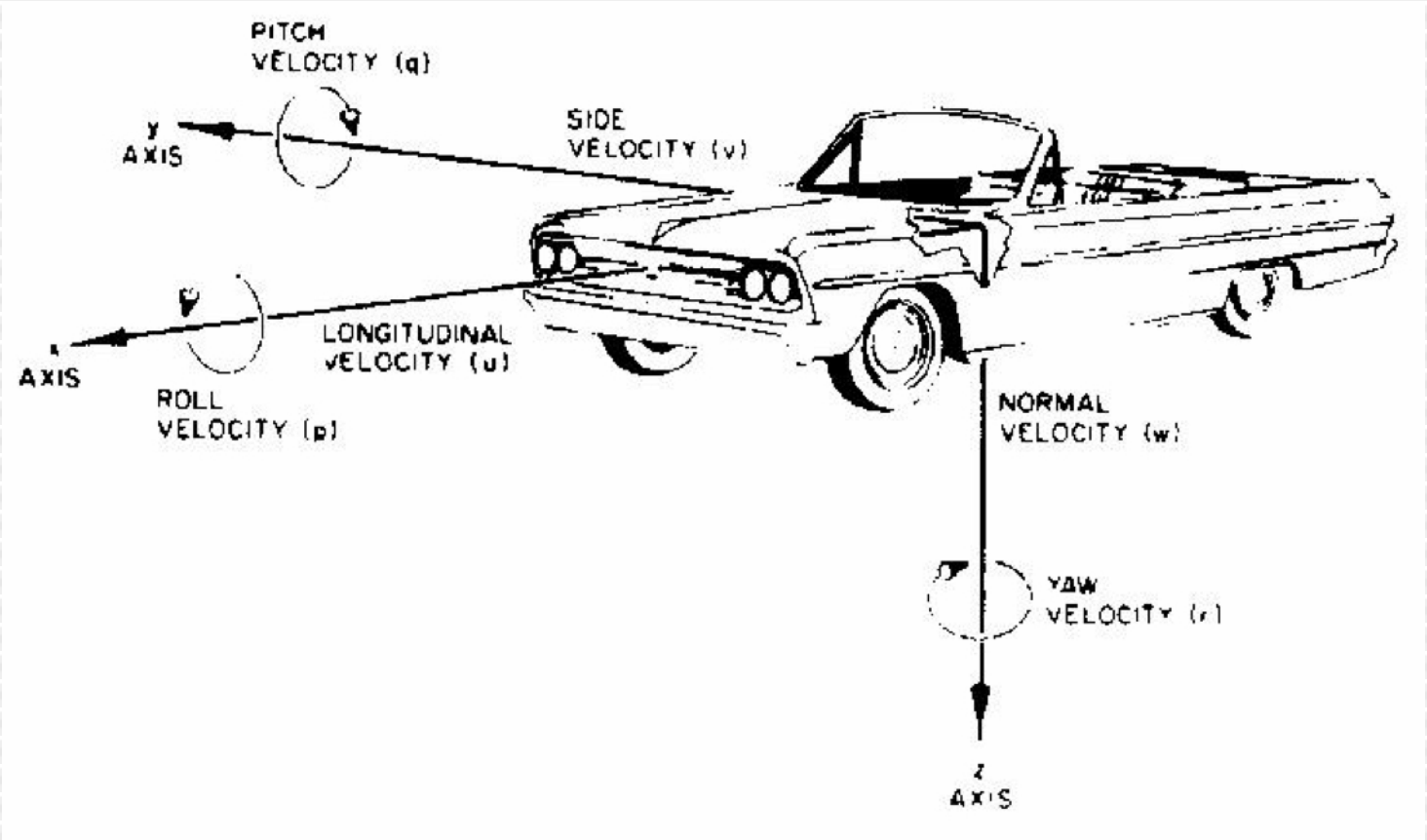
Robot simulation

- UE provides the simulation of actors spawned inside the environment
  - Karma engine handles rigid-body physics
- A robot is a collection of parts with physics properties (mass, etc.) whose simulation is delegated to Karma
- Sensors retrieve their input from UE
- Actuators show their effect upon interactions with the Karma engine

Coordinates

- All coordinates are given according to a left-hand coordinate system
- Units returned to the controller application are in SI
  - But internally UE uses a different system
    - Unreal units (52.5 UU = 1m)
    - Left handed system
- On the application interface the SAE J670 coordinate system is used

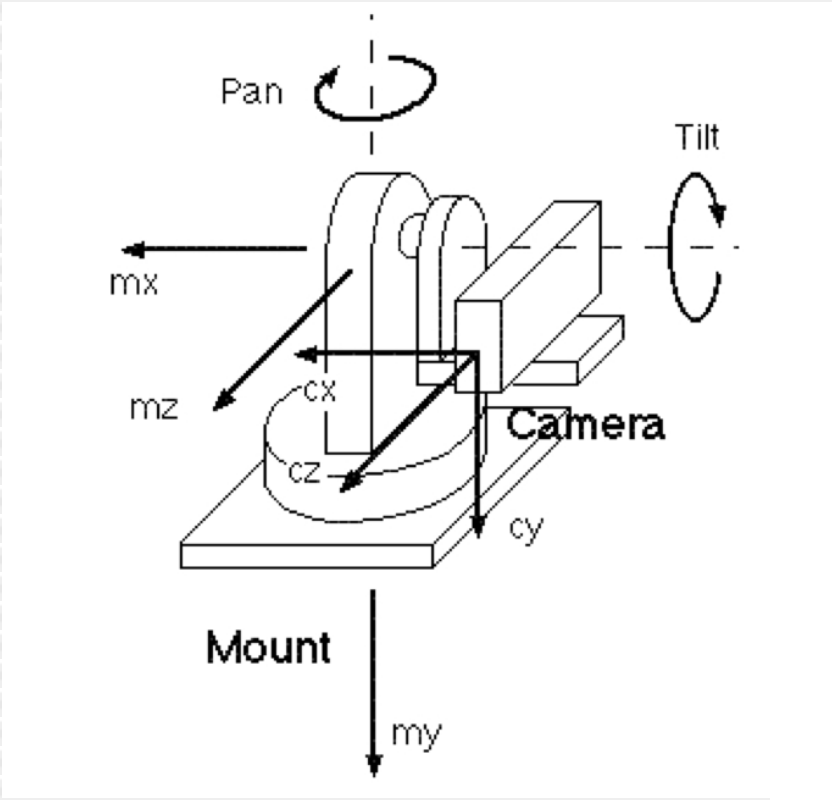
Robot coordinates

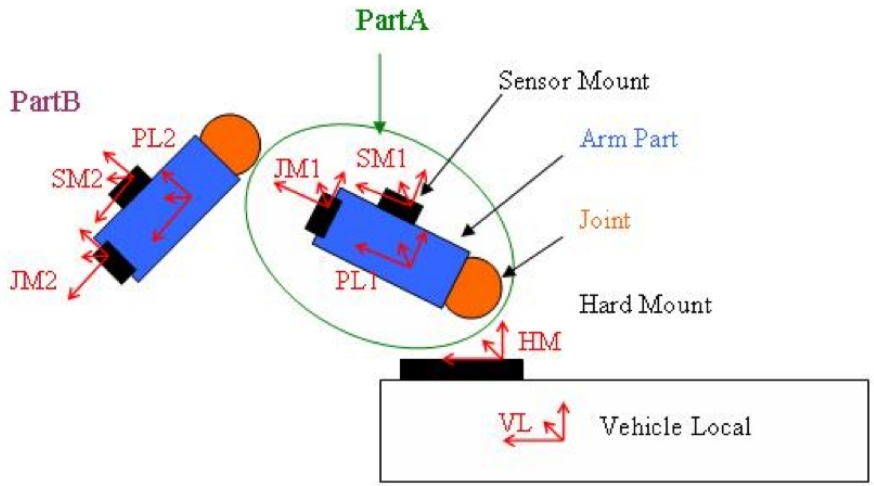


The mission package

- Control parts beyond wheel-based mobility
  - Pan tilt cameras
  - Flippers for tracked robots
  - Robotic arms
- Collection of parts and joints
  - Users can move individual DOFs or specify desired target points and let USARSim solve the IK

Mission package examples





Mission package

The front forks of the robot are modeled using as a mission package (more details in following sessions).

Robot controllers

- Simple text based protocol used for information exchange (details in section 2)
- Interaction USARSim boils down to string exchange over TCP/IP
- Off the shelf options: plugins available for USARSim
  - MOAST: moast.sourceforge.net
  - Player: playerstage.sourceforge.net
  - Pyro: pyrorobotics.org

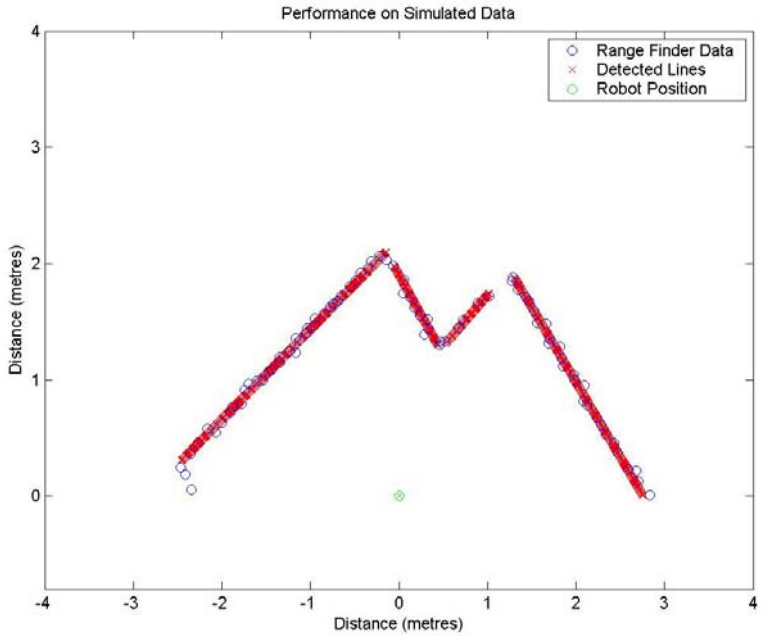
Validation

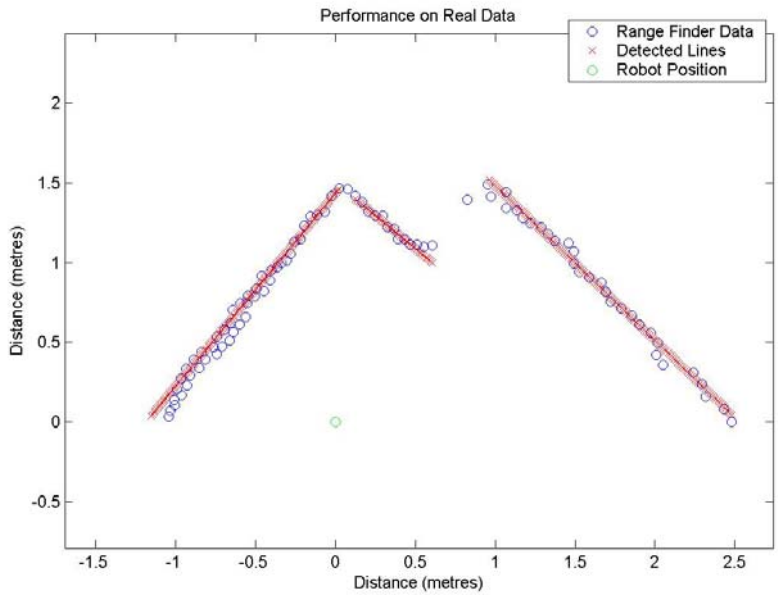
- Ideal scenario
  - Develop algorithms in simulation
  - Move them to real robots without any change
- How far is USARSim simulation from reality?
- Quantitative validation is still ongoing but there are encouraging premises

An example: validating a simulated range scanner

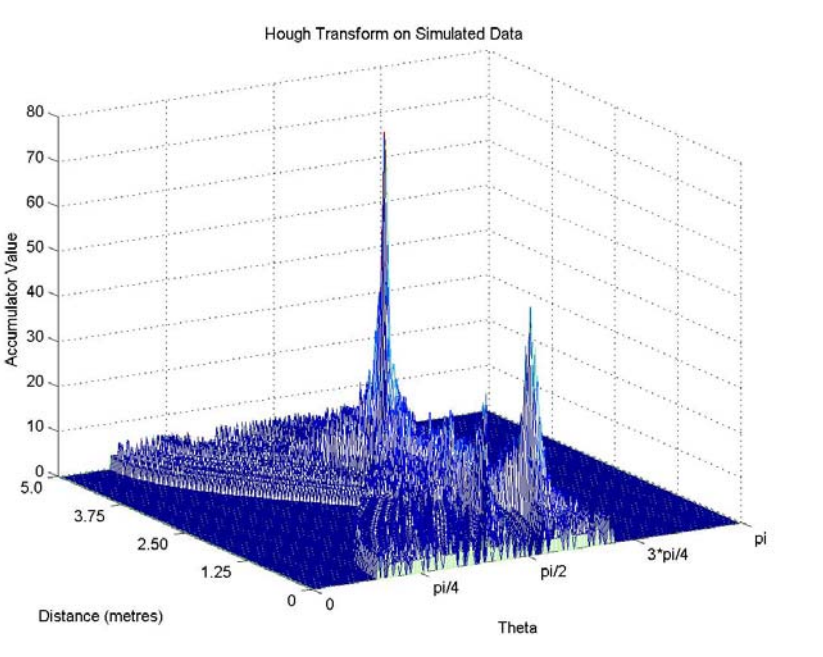
- Define an algorithm processing range scanner data
  - Line extraction with Hough transform
- Get data from a real sensor
- Create a model of the testing environment within USARSim
- Spawn a robot and get data
  - Simulate the same robot, same sensor, same operative conditions
- Run the same algorithm on the two data sets

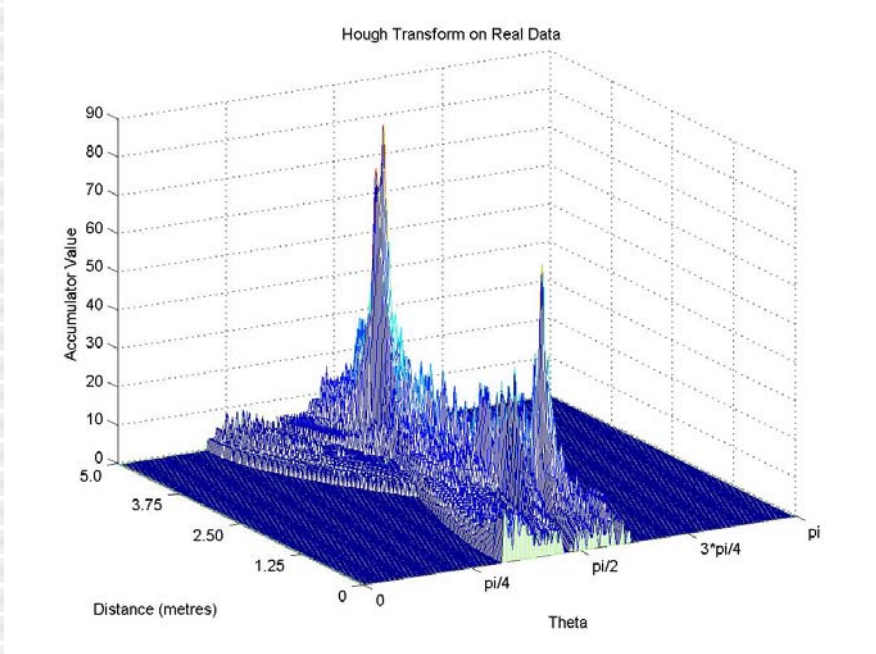
Collected data





Processed data





Virtual Robots Competition

- Urban Search And Rescue competitions in Robocup are held since 2000
  - Real robots
  - Simulated agents
- In Osaka (2005) a demo based on USARSim was held
- Starting from this year a new competition based on USARSim will take place
  - www.robocup2006.org

Competition goals

- Bridge the gap between the existing leagues
- Promote research on
  - cooperation between embodied agents with realistic capabilities
  - HRI: one human supervising tens of robots
- Let people concentrate on what they know better
- Lower entry barriers

Competition schema

- Mainly borrowed from the other leagues
  - Spawn one or more robots inside an unknown environment
  - Explore, locate victims, produce useful information for human responders
  - Minimize unintended interaction with the environment and victims
  - Minimize the number of humans in the loop

2006 competition

- 16 teams registered from Europe, Asia, and the Americas
- Wide student participation
- More info on www.faculty.iu-bremen.de/carpin/VirtualRobots/
- Interested for 2007? Stay tuned (event will take place in Atlanta)

Embodied Intelligent Agent

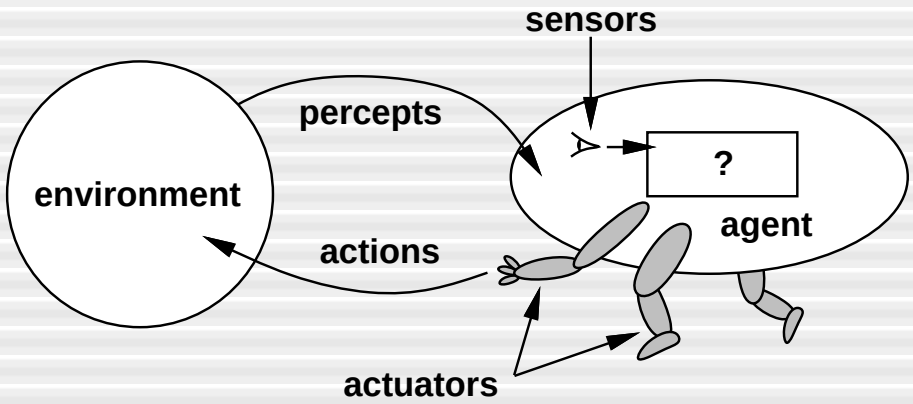


Image from Russell, Norvig *Artificial Intelligence*

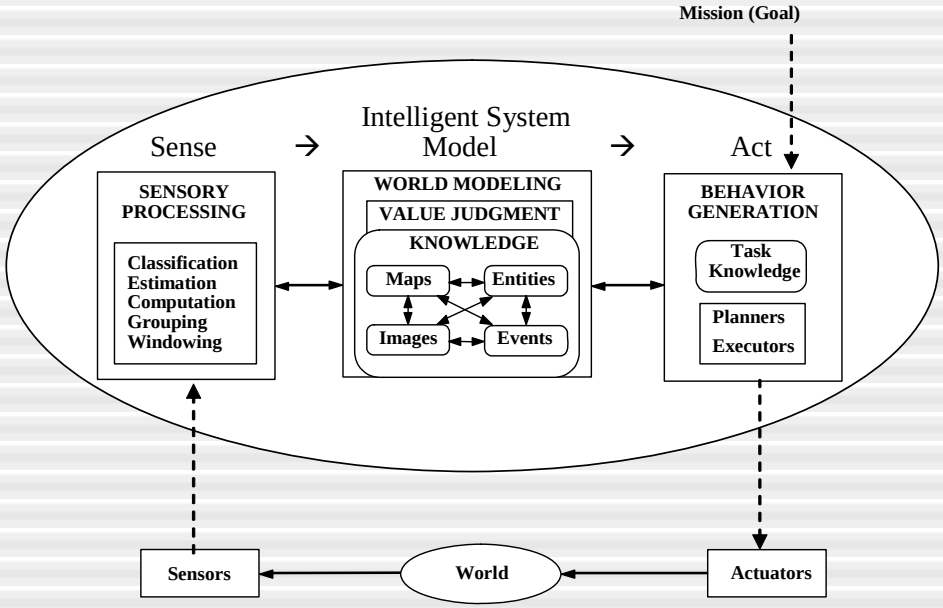
- Environment – provides some place to exist and objects to interact with **(USARSim)**
- Embodiment – provides someway to move around in and effect change of the environment **(USARSIM)**
- Intelligence – provides some way to reason over percepts and to decide on appropriate actions **(MOAST)**

Mobility Open Architecture, Simulation, and Tools
Project Summary

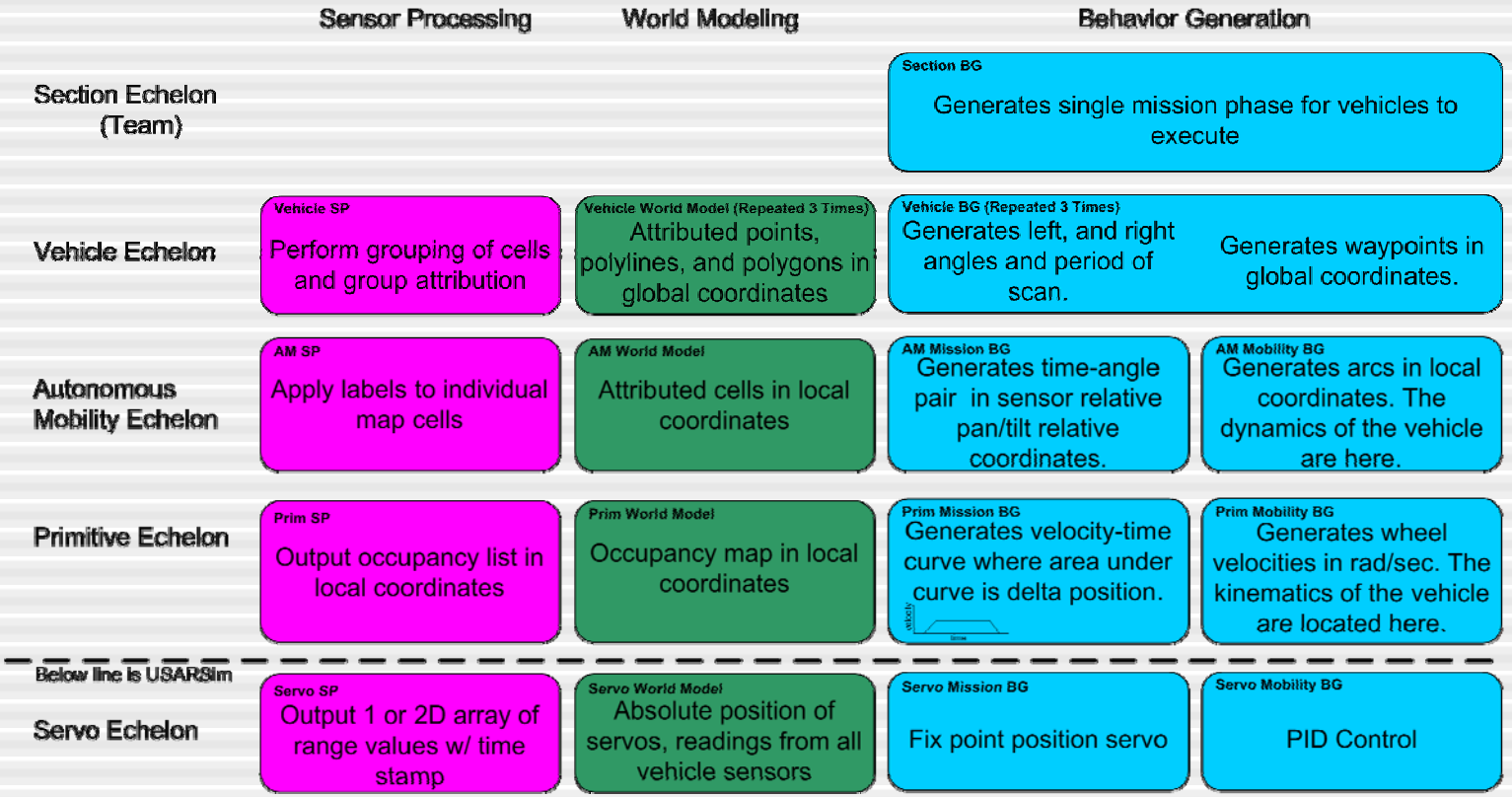
- MOAST is a framework that provides an infrastructure for the development, testing, and enhancement of autonomous systems
- Framework guided by three principles
  - Create a multi-agent simulation environment and tool set that enables developers to focus their efforts on their area of expertise without having to have the knowledge or resources to develop an entire control system
  - Create a baseline control system which can be used for the performance evaluation of the new algorithms and subsystems
  - Create a mechanism that provides a smooth gradient to migrate a system from a purely virtual world to an entirely real implementation

Architecture

- An intelligent system must translate a mission command into actuator voltages
- While this may be done in a single monolithic module, MOAST implements a hierarchical control structure
  - Compartmentalizes responsibility
  - Partitions domain knowledge
  - Standardized interfaces allows multi-programmer development
- As one moves up the hierarchy, have increasing scope and horizons (planning, sensing, …) and decreasing resolution
- Each level in the hierarchy contains:
  - Sensor processing
  - World Modeling
  - Behavior Generation



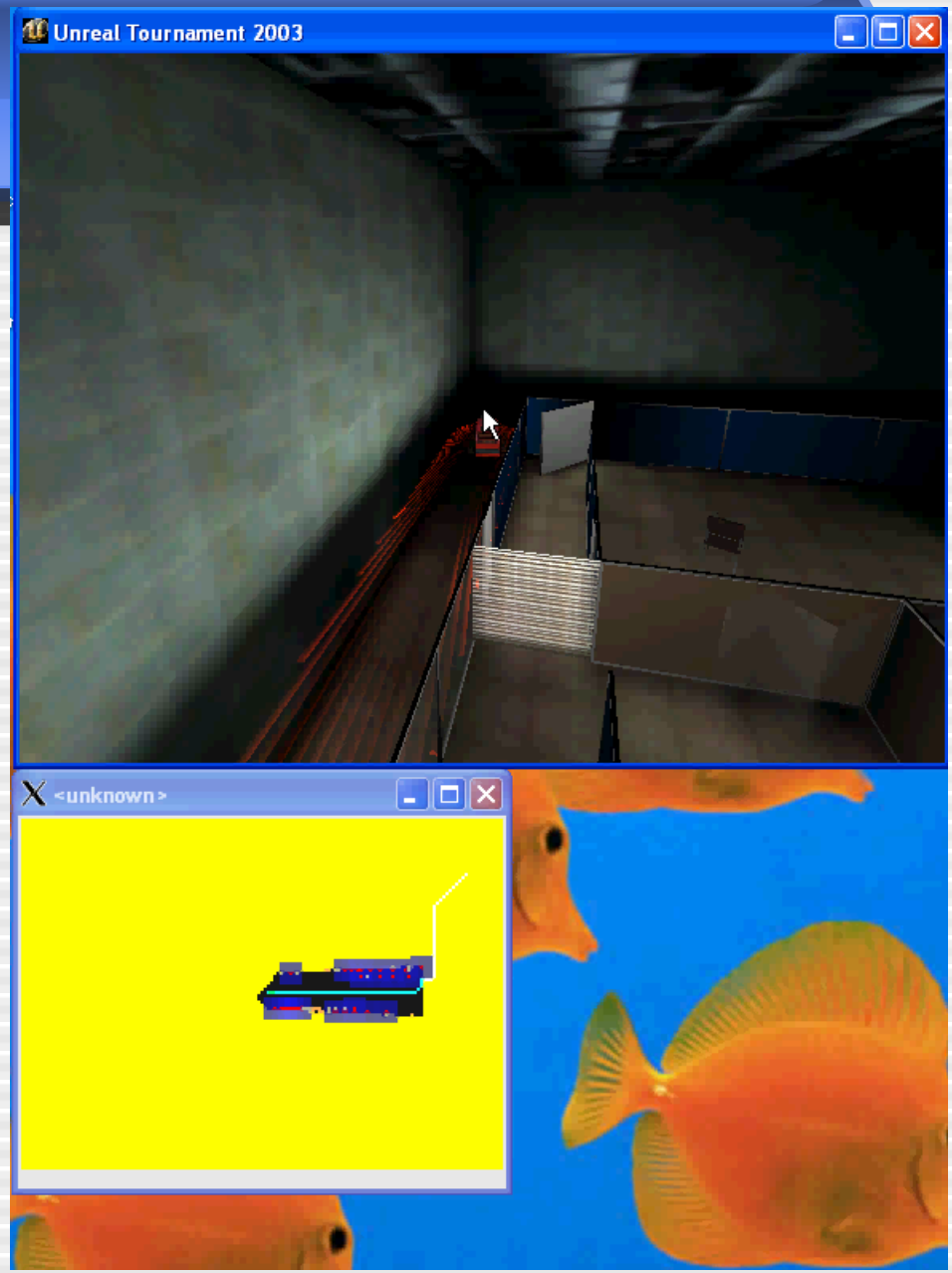
Architecture Modularity



- A user's module may replace one or more of the software modules
- A user may plug their module into any/all of the system's interfaces

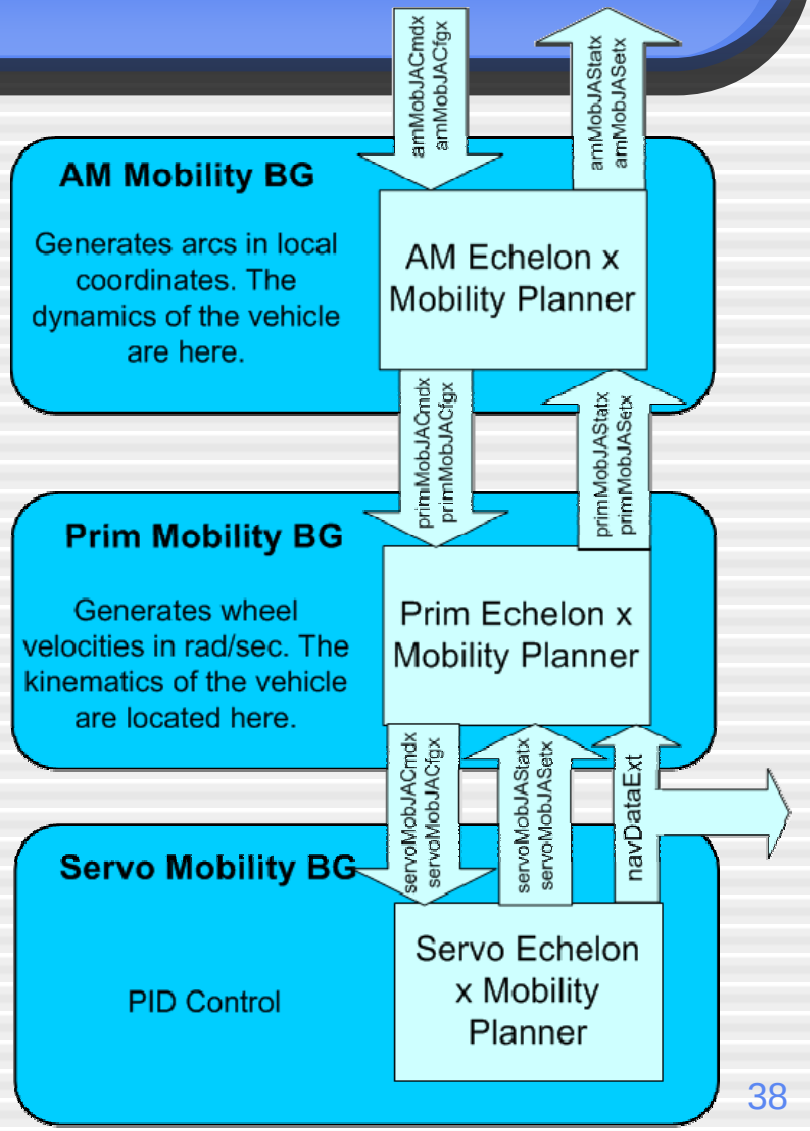
MOAST
Baseline Controller

- Fully integrated with USARSim
- 4D/RCS compliant architecture
  - Specifies hierarchy from actuators up to groups of 10s or 100s of platforms
  - Toolset for unit testing
- Five echelons of implemented control
  - Sensor processing
  - World modeling
  - Behavior generation
- High-fidelity ground truth of real test arenas
  - Elevation data
  - Vector feature data
  - Derived databases
- Central knowledge repository
  - Contains both static and dynamic information
  - Based on domain specific schemas implemented through a central SQL server
- Concurrent operation of real subsystems (hardware-in-the-loop), high-fidelity and low-fidelity simulations



Standard Interfaces

- Module-to-module communications performed via Neutral Messaging Language (NML)
- Each module command, status, configurations, and settings messages implemented as a separate class and defined in C++ header file(s)
  - Automatic tool to create necessary C++ or Java code
- Module may use a separate interface for data transmission



USARSim to MOAST Interface Mappings

- Interface provides direct mapping between USARSim socket data and C++ classes

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

Where:

{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.

```
class ServoMobJACmdML:public RCS_CMD_MSG {
public:
  double v;                    // vehicle speed, [m/s]
  double w;                    // vehicle angular speed, [rad/s]
};
```

Laser Sensor

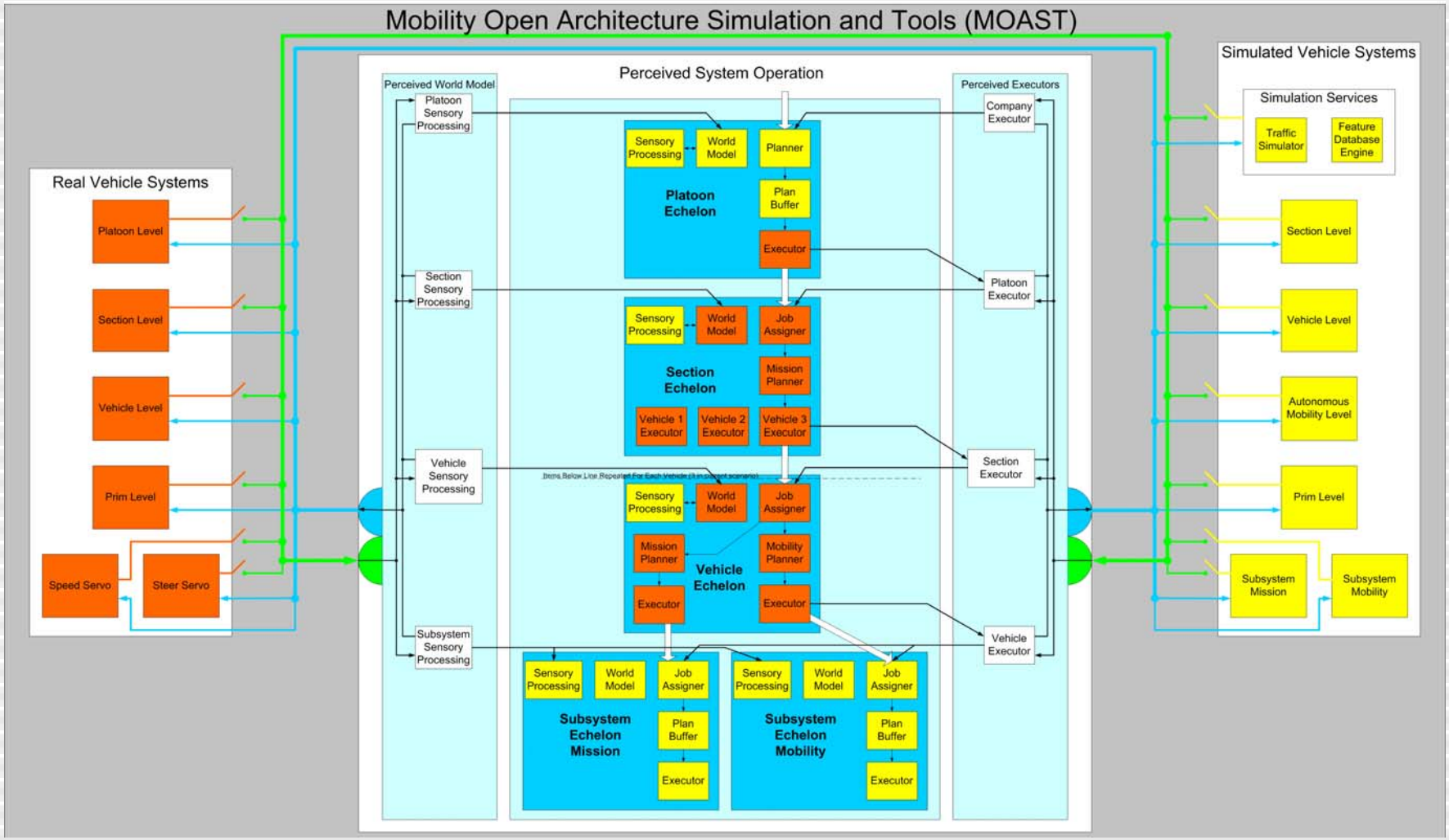
SEN {Type RangeScanner} {Name *string*} {Location *x,y,z* Rotation *pitch,yaw,roll*} {Range *r1,r2,r3…*}

Where:

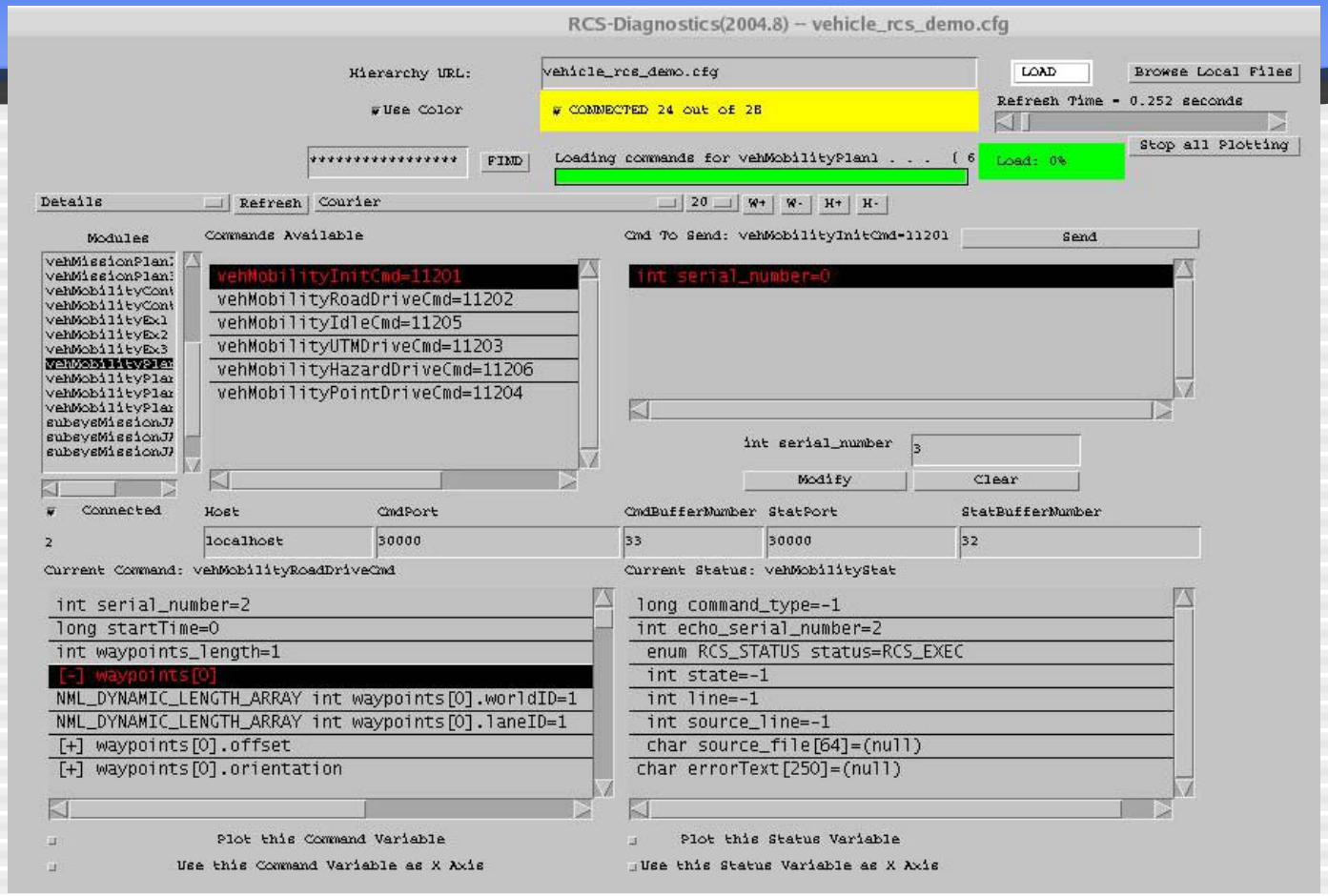
{Name *string*}	'*string*' is the sensor name.
{Location *x,y,z* Rotation *pitch,yaw,roll*}	'*x,y,z*' is the sensor position in UU. '*pitch,yaw,roll*' is the rotation of the sensor. NOTE: the rotation is the absolute rotation.
{Range *r1,r2,r3…*}	'*r1,r2,r3…*' is a serious of range values.

```
class SensorDataList:public NMLmsg {
public:
  //! are the cells in local or global coordinates?
  bool isGlobal;
  /*! what is the increment between array members. Example, meters between
      cells or angle between returns. Since SensorPoints identify their
      location, however, some applications may not need to use the
      value of increment.
  */
  double increment;
  /*! maximum range of sensor. SensorPoints whose distance between
      sensorLoc and dataLoc exceeds this should not be used.
      The value should be positive.
  */
  double maxRange;
  /*! minimum range of sensor. SensorPoints whose distance between
      sensorLoc and dataLoc is less than this should not be used.
      The value may be zero but should not be negative.
  */
  double minRange;
  /*!
     Variable length array of sensor points to add. The name of the array is
     "featList", whose maximum length is defined by MAX_WM_SENSOR_LIST.
     NML creates a variable named "featList_length" that contains the number
     of elements in the current message.
   */
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY(SensorPoint, featList,
                                   MAX_SENSOR_DATA_LIST);
};
```

Real/Virtual Operation



4D/RCS Command/Status Control/Visualization



- Ability to send any command or status to any module
- Ability to examine current active command or status

Session 2

- Low level interfaces and the socket API
  - Spawning a robot
  - Reading from a sensor
  - Moving the robot
- Vision within USARSim
- A simple controller

Talking to UE

- Communication between controllers and UE happens through Gamebots
- When the server starts, Gamebots waits for TCP connections
  - By default it listens on port 3000 and will accept at most 16 connections
  - Both parameters can be changed via the configuration files

Protocol

- Information is exchanged using Gamebot's text based protocol having the following structure

  `DATA_TYPE {Segment1} [{Segment_n}]*`

- Each text string terminates with `\r\n`
- Two flows of information:
  - Messages: from UE to the controller
  - Commands: from the controller to UE

Data types

- Specify the content of a string
- Each type defines its own segments
- Messages and commands have varying length
- Data types are case sensitive

Segments

- Segments are couples *Name/Value*
  - Surrounded by curly braces
  - There can be multiple couples
  - Name and values are separated by a space
  - If the value consists of multiple pieces, they are separated by commas
    - Do not put extra spaces in between the value part
- Examples:

```
{Time 1.28}
{Name F1 Range 2.34}
```

Main command data types

- **`INIT`**: spawns a robot in the environment
- **`DRIVE`**: moves the wheels of a robot, or a specific joint
- **`CAMERA`**: controls the camera (direction, zoom)
- **`SEN`**: sends commands to sensors (reset, etc.)
- **`MISPKG`**: controls a mission package

Messages data types

- **`STA`**: robot status (ground truth!)
- **`MIS`**: status of all mission packages
- **`SEN`**: input from a sensor
- **`GEO`**: information about sensor/actuator geometry (location, orientation, etc.)
- **`CONF`**: information about sensor/actuator configuration
- **`RES`**: response messages

Spawning a robot

- Needed information for the command:
  - Robot type
  - Robot name (optional)
  - Initial position
  - Initial orientation (optional)
- `INIT {ClassName robot_type} {Name name} {Location x,y,z} {Rotation r,p,y}`
- Example:
  `INIT {ClassName USARBot.P2AT} {Location 1.4,2.0,2.8} {Name IcraBot}`

Starting positions

- Each environment provides a set of *good* candidate start positions
  - Spawning a robot inside a wall is possible, but will obviously give unexpected results
  - If the *z* coordinate is not appropriate the robot is spawned flying in the air and will fall down!
    - And after if falls it will bounce back and forth…
- If no orientation is provided the robot will face the north direction

Status message

- As soon as the robot is spawned, it starts receiving messages of type STA

```
STA {Time t} {Location x,y,z} {Orientation r,p,y} {Velocity vx,vy,vz} {LightToggle lt} {LightIntensity li} {Battery b}
```

- STA messages are not really a sensor, because they report ground truth
  - Useful to perform quantitative evaluation of algorithms operating on noisy sensor data

Mission package status messages

- In case the robot is equipped with mission packages, it receives also state messages concerning all of them

MIS {Type *string*} {Name *string*} {Part *string* Location *x,y,z* Orientation *x,y,z*} [{Type *string*} {Name *string*} {Part *string* Location *x,y,z* Orientation *x,y,z*}]*

Mission package status messages

- **MIS {Type PanTilt} {Name Cam1} {Part CameraBase Location -0.0000,-0.0000,0.0000 Orientation 0.0000,0.0000,0.0000} {Part CameraPan Location -0.0000,-0.0000,0.0280 Orientation -0.0001,0.0003,0.5693} {Part CameraTilt Location -0.0000,-0.0000,0.0000 Orientation 0.0000,0.0000,0.0000}**

Driving the robot around

- **`DRIVE`** commands can be used to move robots around
- Two forms
  - Drive left and right wheels independently
    - Useful for differential drive robots
  - Drive a single joint of the robot
    - Useful for more general platforms, like omnidrives

Moving a differential drive

- **`DRIVE {Left` *`float`*`} {Right` *`float`*`}`**
- The two floats specify the speed in rad/s of the wheels on the left and right side. Ex:
  `DRIVE {Left 0.5} {Right -0.5}`
- The **`DRIVE`** command accepts more parameters (see USARSim manual for details)
- The command is *persistent*: wheels keep turning at the given speed until a new command is given

Driving individual joints

- **`DRIVE {Name` *`string`*`} {Steer` *`int`*`} {Order` *`int`*`} {Value` *`float`*`}`**
  - **`Name`**: joint to drive
  - **Steer**: steer angle of the joint
  - **Order**: 0(Position), 1(speed), 2(torque)
  - **Value**: value to apply
- Commands of order 1 and 2 are persistent

Controlling a camera

- If the robot mounts a PTZ camera, it is possible to control it
  `CAMERA {Rotation x,y,z} {Order int} {Zoom float}`
  - **Order**: same meaning as for the **`DRIVE`** command
  - **`Rotation`**: target rotation
  - **Zoom**: sets the field of view
- In the future this command will likely be replaced by a suitable mission package command (mostly useful if the robot mounts more cameras)

Controlling a camera: examples

- **`CAMERA {Rotation 0,0.2,0}`**
  Specifies relative position
  - Absolute positioning can be obtained changing the configuration file
- **`CAMERA {Rotation 0,0.2,0} {Order 1}`**
  Sets rotational speed about the y axis
- **`CAMERA {Zoom 1.57}`**

Reading from sensors

- Each robot can be equipped with multiple sensors
- Sensors deliver their input sending suitable **`SEN`** messages
  - At regular frequency or upon request
- Sensor readings can be time stamped or not (configurable option)

Range Sensors

- Range sensors simulate sonars and IR sensors
  - Sonars: return the minimum distance measured inside and *emitted cone*
  - IR: only one beam that is not reflected by glasses
- For both sensors it is possible to specify the amount of noise, and the maximum range

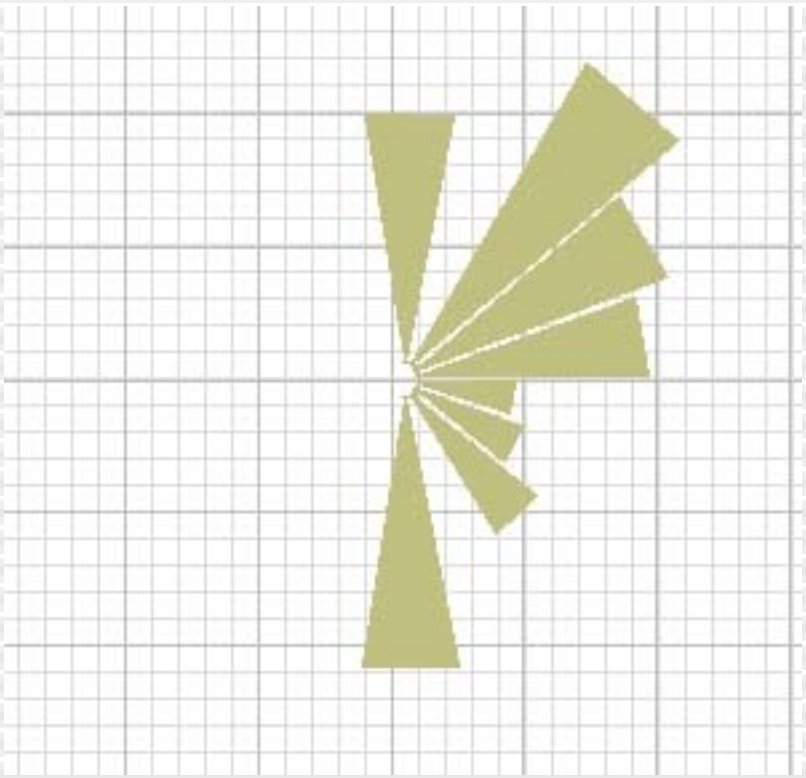
Behind the scenes: how are they implemented?

- Inside UE it is possible to emit a *ray* from a certain point in a given direction and to detect where the line hits the first element in the world
  - These are builtin functions within the Unrealscript language
- Sonar emits many rays within a cone and returns the minimum distance (if within the maximum range). IR emits only one ray.

Range messages

- **SEN {Type *string*} {Name *string* Range *float*} [{Name *string* Range *float*}]***
- **Type** can be **Sonar** or **IR**
- **Name** is the unique name (string) given to the sensor
  - more sensors of the same type can be used as long as they have unique names
- **Range** is the value returned
- Example:
  SEN {Type Sonar} {Name F1 Range 0.98}
  The above sensor is not time stamped

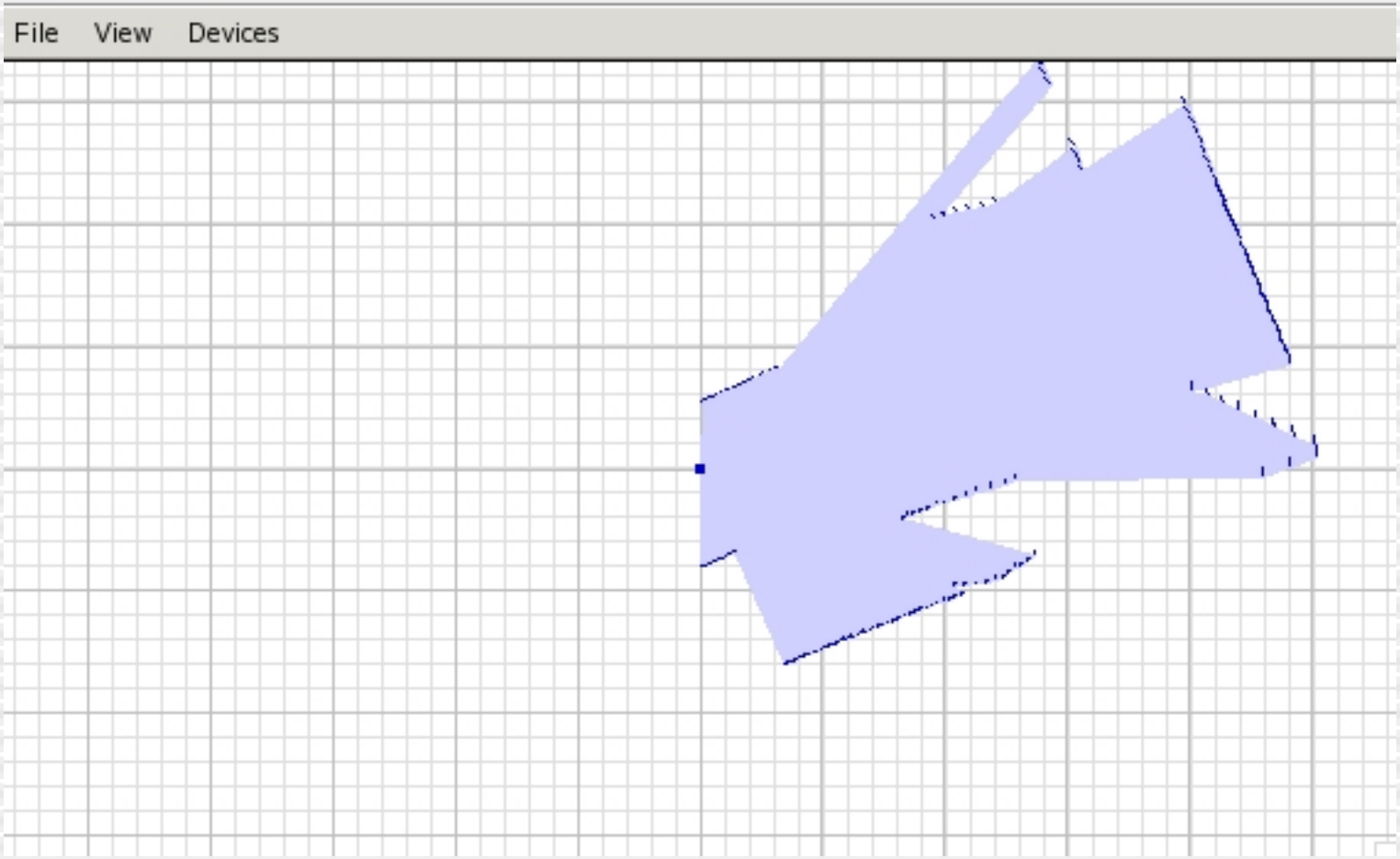
How do they look like?



Laser sensors

- *Laser* sensors simulate laser or IR based PLS
- Message structure:
  `SEN {Type string} {Name string} {Resolution float} {FOV float} {Range r1,r2,r3…}`
- **Type**: **RangeScanner** or **IRScanner**
- **Name**: name of the device (multiple instances of the same sensor are possible)
- **`Resolution`**: angular resolution
- **FOV**: field of view
- **Range** : values. The number of values is FOV/Resolution + 1

Data from the laser



Laser sensors

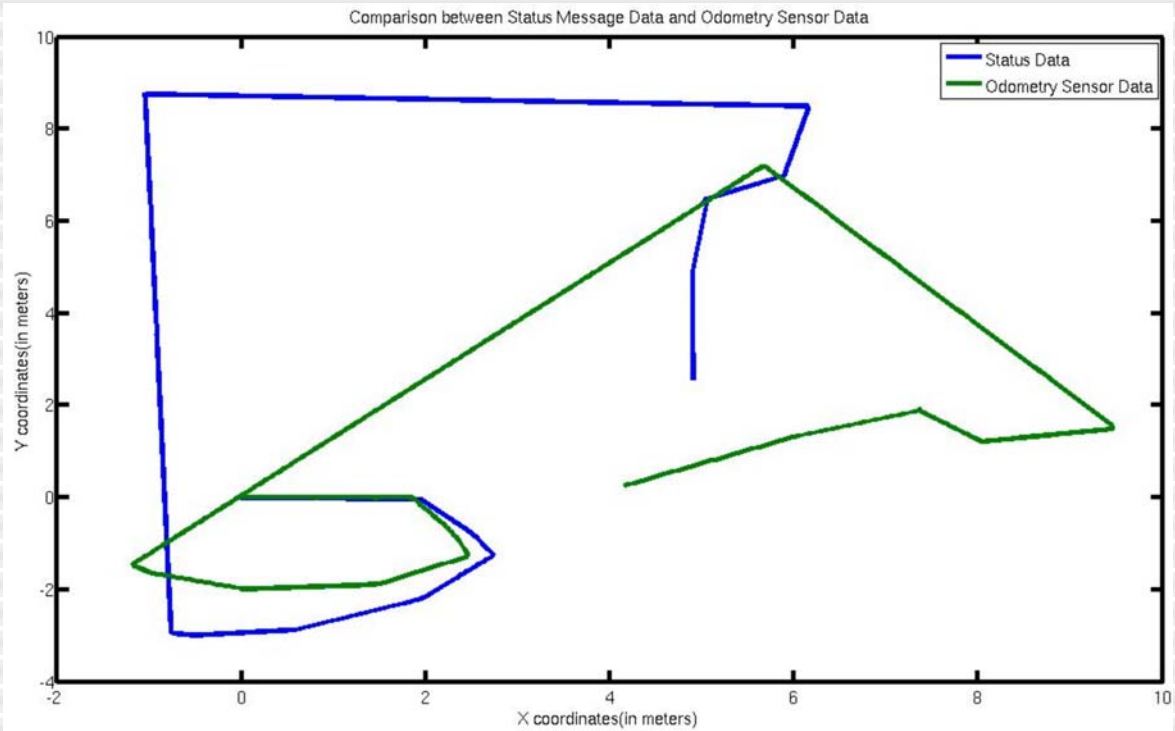
- If polled too frequently, range sensors can be quite demanding for UE
- They can operate in two modes:
  - Automatic: send a message at a fixed frequency
  - Manual: send a message only upon request
- Mode of operation can can be specified in the configuration files

Other sensors available

- The following sensors provide information to the controller application via Gamebots
  - Odometry: noisy estimation of robot's pose (x,y,yaw)
  - INU: orientation
  - Touch sensors
  - RFID
  - Encoders

A word on odometry

- Cumulative errors need to be carefully reproduced, to keep simulation close to reality



Geometry messages

- Messages of type **GEO** return information about sensors' or actuators' position on the robot

```
GEO {Type Camera} {Name Camera Location
0.0820,0.0002,0.0613 Orientation
0.0000,0.0000,0.0000 Mount CameraTilt}
```

- Controllers can therefore inquire about the robot's structure

Configuration messages

- Messages of type **CONF** describe sensors' or mission packages' features not covered by **GEO** messages

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

Responses

- Messages of type **`RES`** are sent as responses to certain commands to notify about their result
- Example: there exists a command to reset the odometry sensor. Upon successful completion, the sensor replies with

```
RES {Time 61.20} {Type Odometry} {Name Odo1} {Status OK}
```

The `SET` command

- Commands are sent to actuators/effecters
- Each device accepts a predefined set of commands

  `SET {Type `*`string`*`}{Name `*`string`*`}{Opcode `*`string`*`} {Params `*`p1,p2,…`*`}`
- **`Opcode`** specifies which operation is asked

  `SET {Type Odometry} {Name Odo1} {Opcode RESET} {Params 1}`

Controlling mission packages

- Individual joints can be actuated with the DRIVE command
- MISPCK allows to specify the terminal's pose

  `MISPGK {Name` *`string`*`} {Rotation` *`x,y,z`*`} {Translation` *`x,y,z`*`} {Order` *`int`*`}`
- Short demo

Simulating a camera

- Goals:
  - provide the controller application a suitable image to test robot vision algorithms
    - Resolution, frame rate and format comparable with commercial devices
  - provide human operators realistic video feedback
- Images are not sent via Gamebots
- Camera simulation needs to be dealt with separately

Getting video feedback

- Four possibilities:
  1. Via UT client (external observer point of view)
  2. Embed UT client into your controller application and attach the POV to the robot camera (human operator point of view)
  3. *Steal* images from UT client (images are then available for being processed by your controller)
  4. Get images from an the external application ImageSrv (let ImageSrv perform task 3 instead of your controller).
- We will concentrate on option 4

Capturing an image

Steal Image

ImageSrv

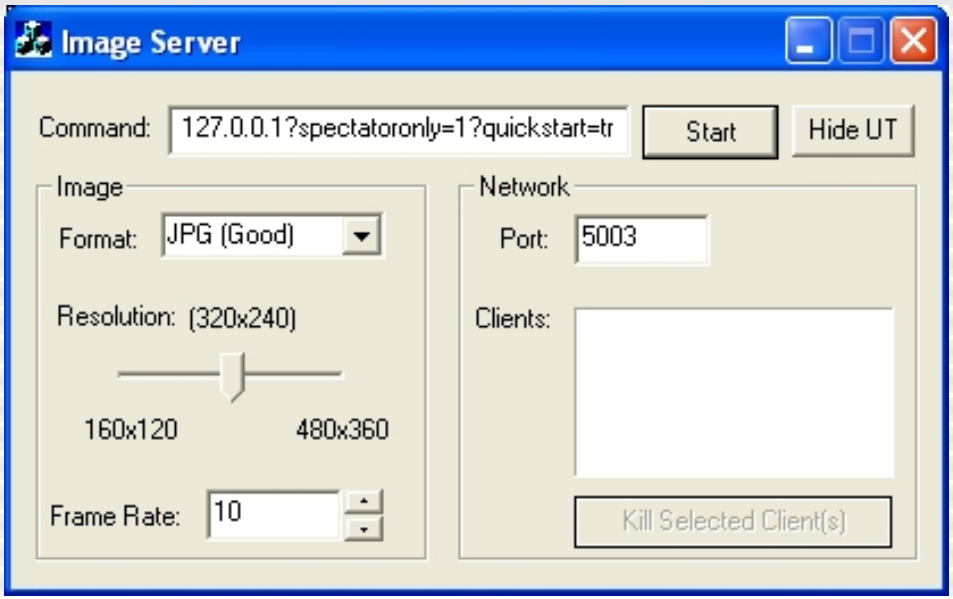
Provide Image

Controller Application

Set POV

ImageSrv

- ImageSrv allows to specify:
  - Frame rate (in fps)
  - Image type
    - Raw or JPG (at different quality)
  - Image resolution



ImageSrv operation

1. Wait for connections
2. Steal image from Unreal client, encode it and send it to the controller
3. Wait for acknowledge message (“OK”)
4. Wait for next time trigger
5. Go to point 2

- Note that as soon as ImageSrv receives connection, it immediately sends an image to the client controller

Image Format

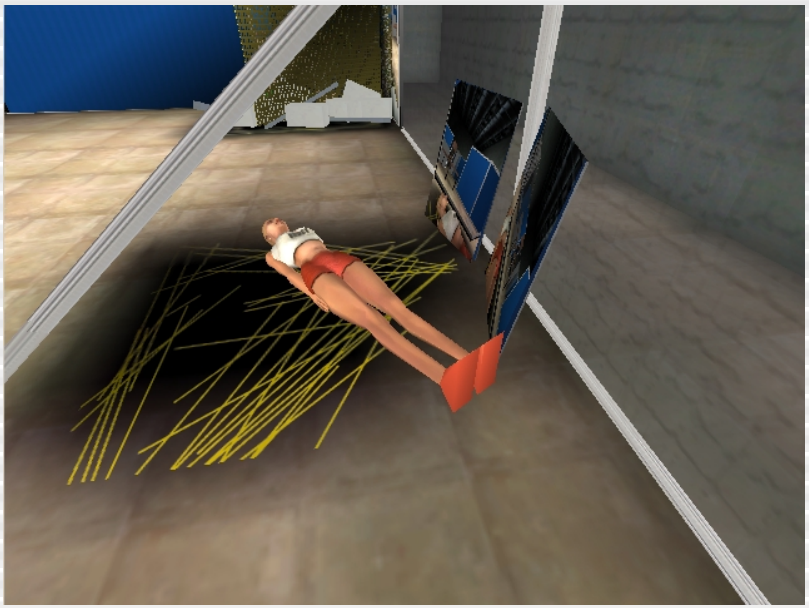
- Images sent by ImageSrv to controllers follow a simple format:
  - Image type: 1 byte (from 0 to 5)
  - Image size: 4 byte
  - Image data
    - Jpg: encoded image
    - RAW: width (2) + length (3) + RGB values for each pixel

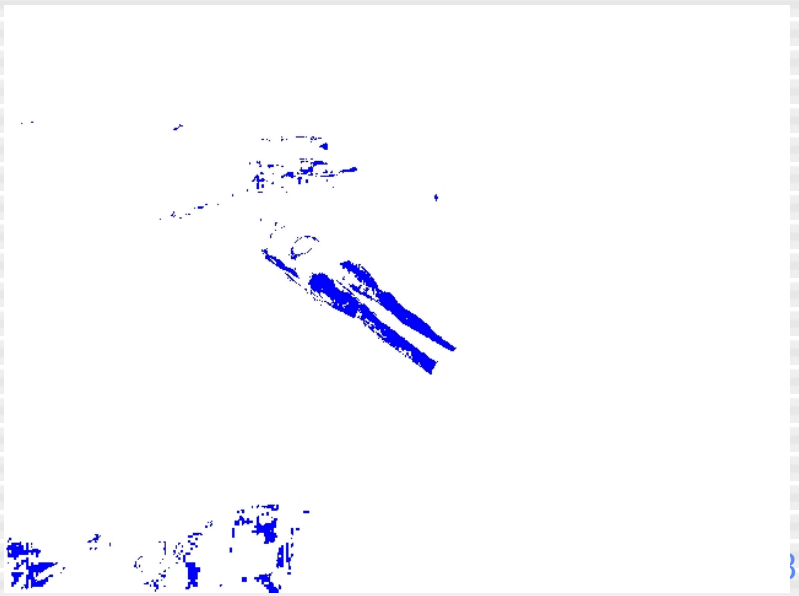
Changing the point of view

- ImageSrv provides the image currently displayed by the UT client
- If multiple robots/cameras are present, the point of view has to be changed accordingly. The command
  `SET {Type Camera} {Robot `*`string`*`} {Name `*`string`*`}{Client `*`ip`*`}`
  performs this task

Example: autonomous victim recognition using ImageSrv

- Simple application used during 2005 Osaka demo
  - Get image from ImageSrv, segment based on skin color, look for big blobs





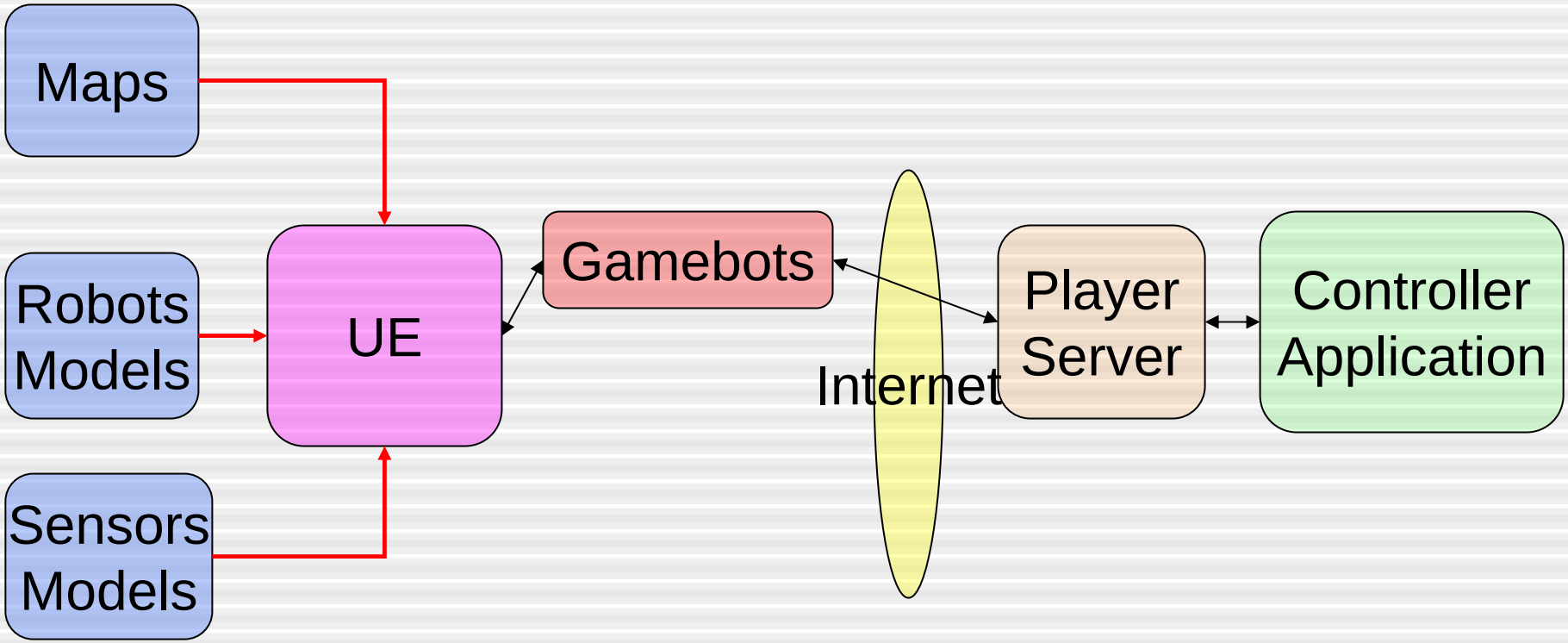
Controlling your robot

- Different options:
  - Talk directly with Gamebots
  - Use pyro plugins
  - Use player plugins (our focus now)
  - Use MOAST (details in session 4)

Player & USARSim

- Player: playerstage.sourceforge.net
- Grand goal: run the same code both un USARSim and on your real robot
- Player plugins for USARSim are available under the *tools* section on sourceforge

Player based controller



Starting the controller

- Start the USARSim server (details in session 3)
- Start the player server
  - Provide a suitable configuration file with details concerning UT
- Start your player-based application. It will transparently talk to UT (through Player and Gamebots)

The configuration file

- The driver `us_bot` allows player to talk to gamebots

```
driver (
    name "us_bot”
    provides ["simulation:0"]
    host "10.50.209.236”
    port 3000
    rot "0,0,-1.55”
    bot "USARBot.PapaGoose”
    robot_radius "0.5”
    tire_radius "0.1”
    botname ”ICRAbot”
)
```

Player drivers available in USARSim

- Laser
- Inu
- Position
- Position3d
- Sonar
- FiducialFinder (for RFID tags)

A simple controller

- Existing player based controllers can be immediately used within USARSim
  - Just change the configuration file
  - Of course, suitable sensor drivers should be available…
- Little demo

Session 3: USARSim hands on

- Installation
- The role of ini files
- Extending the simulator
  - Creating new robots
  - Creating new sensors

USARSim installation for Win

1. Install Unreal Tournament 2004
   - Install the latest available patch (www.unrealtournament.com/ut2004/downloads.php)
2. Get USARSim (sourceforge.net/projects/usarsim)
   1. Tarball and CVS available
3. Unzip USARSim under the UT2004 base folder
4. Done!

   Note that the USARSim tarball contains only the core files. Additional maps, tools, etc. are available under different areas on sourceforge.

   for example ImageSrv is located under the *tools* section

Installation for Linux

- UT2004 is available for Linux as well
  - See USARSim's docs for patch details
- USARSim installs the same way under Linux
- Note: ImageSrv and some other tools available under the *tools* section are available only for Win
- From now on we concentrate on Win only

Running the server

- Move to the folder `UT2004/system`
- Run

  `ucc server` *`map_name`*`?game=USARBot.USARDeathMatch?TimeLimit=0 -ini=USARSim.ini -log=usar_server.log`

  - *`map_name`*: environment to simulate
  - `USARSim.ini` configuration file to be used (more later)
  - `usar_server.log` mostly useful to track erroneous situations
- Upon startup gamebots silently waits for incoming client connections

Maps

- Maps are located under `ut2004/Maps`
  - Maps need also *textures* and *meshes* located under `ut204/Textures` and `ut2004/StaticMeshes`
  - They are not part of the CVS
- Available maps
  - `DM_USAR_yellow_scaled`: planar (maze like)
  - `DM_USAR_orange_scaled`: includes slopes and stairs
  - `DM_USAR_red_scaled`: rubble pile
  - `DM_USAR_black_scaled`: highly 3D environment

Starting the client

- Move to `ut2004/system` and run `ut2004 ip_address?spectatoronly=1?quickstart=true -ini=USARSim.ini`
- *`ip_address`*: server's IP address
- Client and server applications can run on the same machine
- On each maching can be only one instance of the client running
- Upon startup you will be allowed to explore the environment
- Initially the environment is empty: robots have to be spawned by controller applications

Observing from different positions

- Using mouse/keyboard you can move inside the map
- Mouse-left-click switches the point of view to the robot's camera
  - Further left clicks iterate through all cameras
  - Press 'C' to view the robot from above
  - Right click to brings you back to the general viewpoint

Configuring the server

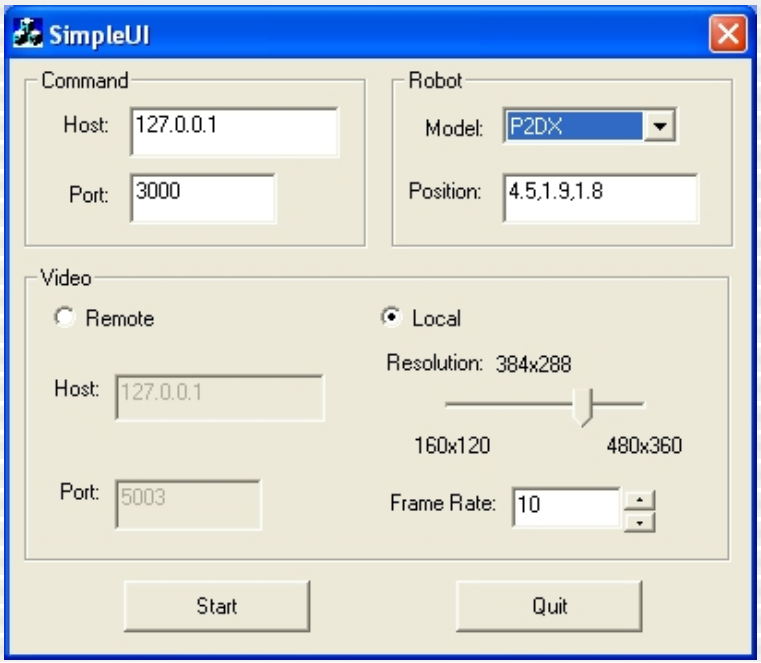
- By default Gamebots will accept at most 16 connections on port 3000
- If needed this can be modified changing the `[BotAPI.BotServer]` section of the `BotAPI.ini` file located in located in `ut2004/system`

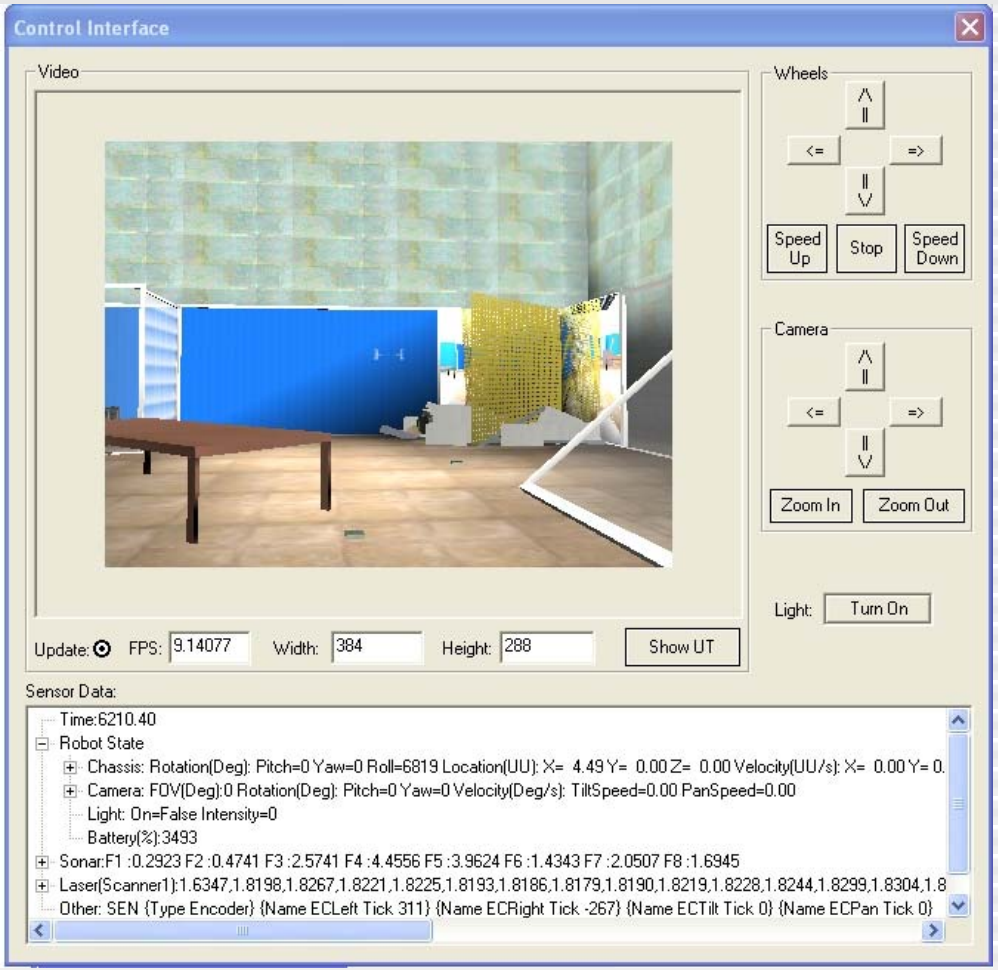
```
[BotAPI.BotServer]
ListenPort=3000
MaxConnections=16
```

Start the controller

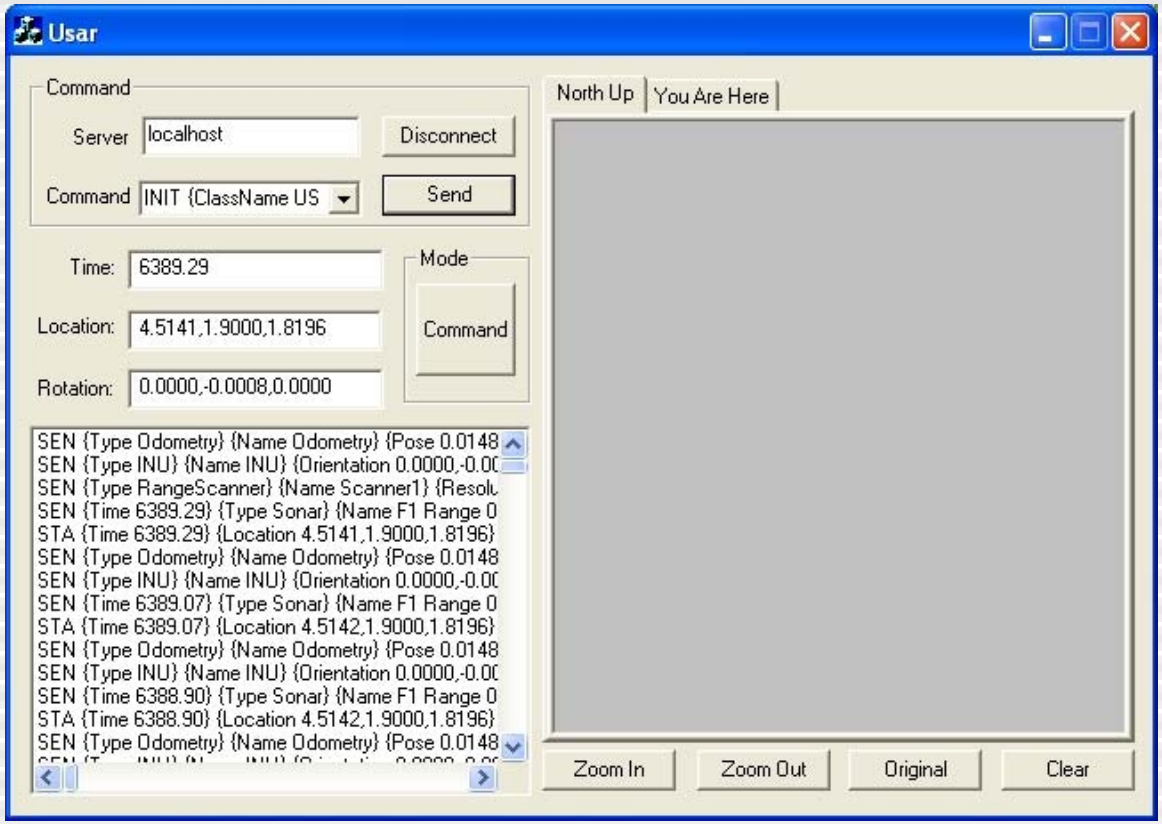
- Connect to Gamebots and spawn a robot. From that point on the game is yours!
- To get acquainted with USARSim robots and arenas, under the *tools* section you find two utility programs
  - SimpleUI: simple user interface
  - UsarUI: USAR user interface

Simple user interface





UsarUI application



With UsarUI it is possible to type commands that are directly sent to gamebots and to examine the received messages.

Configuration

- Almost all USARSim components are configurable
- Parameters can be set in the USARBot.ini file
- Each section defines parameters relevant to a certain device

Configuring sensors

- Most sensors accept the following parameters (more available):
  - `bWithTimeStamp`: reading time stamped?
  - `Noise`: random noise added to the produced value (currently uniform)
  - `OutputCurve`: distortion model used to reshape returned values

Example: configuring a sonar

```
[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)))
```

Advanced use

- USARSim is highly extendible. You can create:
  - New robots
  - New sensors
  - New environments (with UnrealEd)
- Unrealscript object oriented paradigm greatly simplifies things

Building a new robot

- A robot is made of
  - Chassis
  - Parts
  - Joints (connecting parts together)
  - Items (sensors, actuators, etc)
- Developing a new robot from scratch may be a challenge. It is advisable to start modifying existing models

Steps to build a robot

1. Build geometric model
2. Create the part/wheel class
3. Create the robot class
4. Connect parts to the chassis (via joints)
5. Mount auxiliary items (if needed)
6. Specify how the robot is controlled (i.e. which commands it accepts)

- Steps 2 and 3 are simplified thanks to inheritance from the USARSim classes

Geometric model

- Use the UnrealED application
  - Learn more about UE from
    http://udn.epicgames.com
    http://wiki.beyondunreal.com/wiki/
- Models must be *StaticMeshes*
  - Models for chassis, wheels, etc
- Important: align the static mesh with the used reference system
  - *x* points straight ahead the robot
  - *y* points to the left

Create parts

- A wrapper class is needed for the created geometric models

```
class part_class_name extends KDPart;
defaultproperties
{
…
}
```

- In the `defaultproperies` section we specify to which static mesh it is connected to, and physical properties so that the Karma engine can properly simulate its behavior

Create a robot

- A robot extends the class Krobot

```
class robot_class_name extends KRobot
config(USAR);
defaultproperties
{
   …
}
```

- The `defaultproperties` section has the same meaning we saw for the parts

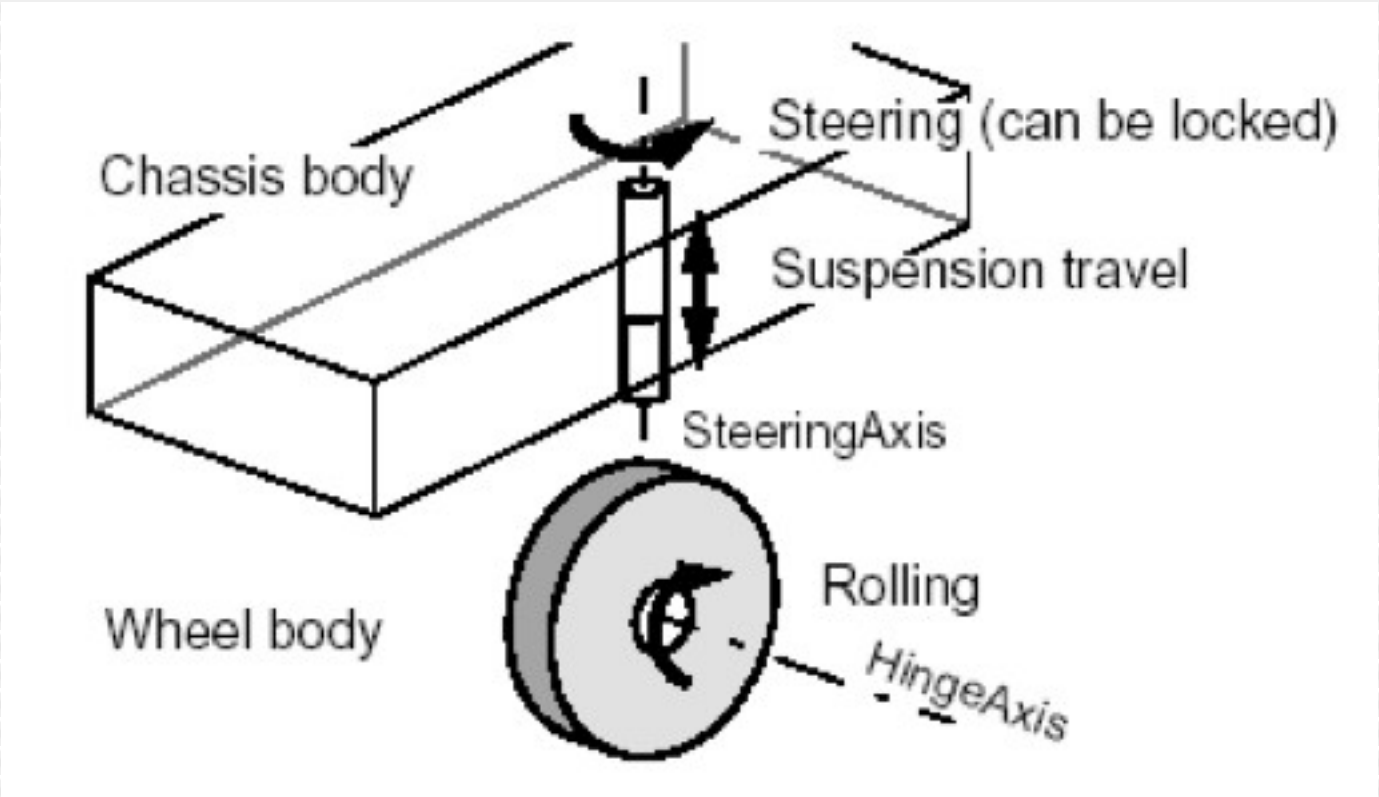
Parameters

- In the *parameters* section you can specify information like
  - Motor torque
  - Mass
  - Associated static mesh
  - Drawing scale
  - Karma parameters
    http://udn.epicgames.com/Two/KarmaReference

Specifying connections

- Part-joint pairs are used to connect parts to the chassis
  - Car-wheel: mainly for wheels
    - Use two axes
  - Hinge: generic parts
    - Use only one axis
- Need to define the involved pairs, and the positions

Car wheel joint



Robot customization

- With the `DRIVE` robot each individual joint can be independently driven
  - Inherited
- Overriding the *ProcessCarInput* method it is possible to change the intepretation of `DRIVE {Left fl} {Right fr}`
- New commands can be added as well

Adding new commands

- Controllers send commands to Gamebot
  - USARRemotebot class
- USARRemotebot is associated with a USARBotConnection listening for incoming commands and parsing information
  - Change the method ProcessAction inside USARBotConnection so that the new command is dealt with
- Example will follow

Example: build an omnidrive

- Goal: develop a robot that simulates and omnidirectional platform
  - Three Swedish wheels arranged on
- Keep things simple: assemble together existing elements coming from other robots
  - No need to develop new parts, joints, etc
  - Specifically, we reused parts from the USARSim provided *PER* robot

Robot properties

```
class Omni extends KRobot config(USARBot);
…
defaultproperties
{
        bDebug=True
        safeDistance=0.18
        ChassisMass=1.000000
        StaticMesh=StaticMesh'USAR_Robots.Robots.PERBody'
        DrawScale=1.0
        DrawScale3D=(X=1.0,Y=1.0,Z=1.0)
        TireRestitution=0.000000
        TireSoftness=0.000001
        TireRollFriction=15.000000
        TireLateralFriction=0.00000
        TireSlipRate=0.0002000
        MotorTorque=30.0
```

Robot properties (cont'd)

```
Begin Object Class=KarmaParamsRBFull Name=KParams0
        KInertiaTensor(0)=0.010000
        KInertiaTensor(3)=0.02000
        KInertiaTensor(5)=0.03000
        KCOMOffset=(X=0.000000,Z=-1.000000)
        KLinearDamping=0.500000
        KAngularDamping=50.00000
        KStartEnabled=True
        bHighDetailOnly=False
        bClientOnly=False
        bKDoubleTickRate=True
        KFriction=1.600000
        Name="KParams0"
    End Object
    KParams=KarmaParamsRBFull'USARBot.Omni.KParams0'
}
```

Putting things together

- Connections, sensors, etc. can be specified in the appropriate section of the `USARbot.ini` file
- The section must be named like the robot, i.e. `USARBot.Omni`

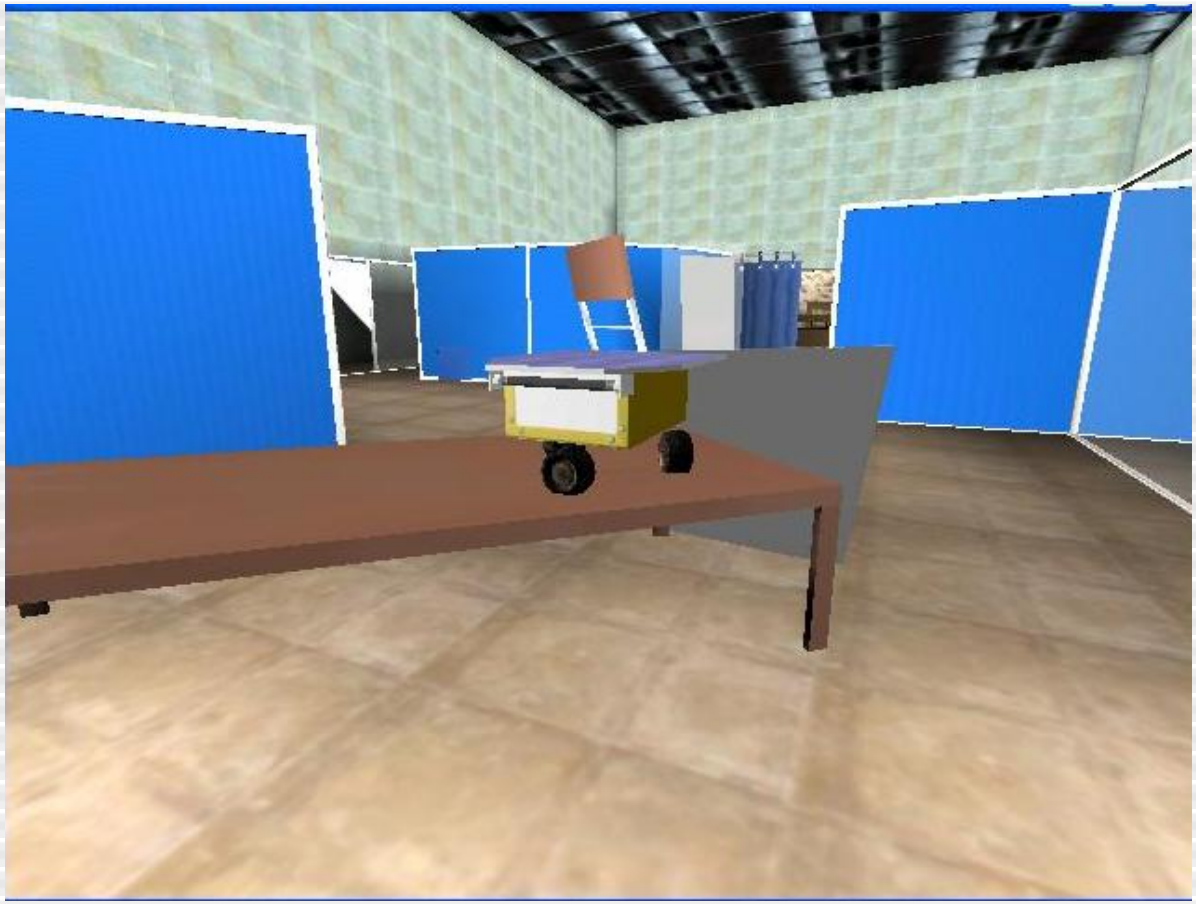
Putting things together

```
[USARBot.Raviomni]
bAbsoluteCameras=True
Weight=5.0
Payload=5
ChassisMass=5.0
MotorSpeed=1.4381
bDebug=False
bMountByUU=False
safeDistance=0.18
JointParts=(PartName="FWheel",
PartClass=class'USARBot.PERTire',DrawScale3D=(X=1.0,Y=1.0,Z=1.0),
bSteeringLocked=true,bSuspensionLocked=true,Parent="",
JointClass=class'KCarWheelJoint',ParentPos=(X=0.11,Y=0,Z=0.09),
ParentAxis=(Z=1.0),ParentAxis2=(X=1),SelfPos=(Z=0.0),SelfAxis=(Z=1.0),
SelfAxis2=(X=0,Y=1.0,Z=0))
```

Putting things together

JointParts=(PartName="LeftRWheel",PartClass=class'USARBot.PERTire',
DrawScale3D=(X=1.0,Y=1.0,Z=1.0),bSteeringLocked=true,
bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',
ParentPos=(X=-0.055,Y=-0.0953,Z=0.09),ParentAxis=(Z=1.0),
ParentAxis2=(Y=-0.866,X=-0.5,Z=0)),SelfPos=(Z=0.0),SelfAxis=(Z=1.0),
SelfAxis2=(Y=1)
JointParts=(PartName="RightRWheel",PartClass=class'USARBot.PERTire',
DrawScale3D=(X=1.0,Y=1.0,Z=1.0),bSteeringLocked=true,
bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',
ParentPos=(X=-0.055,Y=0.0953,Z=0.09),ParentAxis=(Z=1.0),
ParentAxis2=(Y=0.866,X=-0.5,Z=0)),SelfPos=(Z=0.0),SelfAxis=(Z=1.0),
SelfAxis2=(Y=1)
Sensors=(ItemClass=class'USARBot.OdometrySensor',
ItemName="Position1",Position=(X=0,Y=0,Z=0),Direction=(X=0,Z=0,Y=0))
Cameras=(ItemClass=class'USARBot.RobotCamera',ItemName="Camera",
Parent="",Position=(X=0.12,Y=0,Z=-0.1),Direction=(Y=0,Z=0,X=0))

Omnidrive: the final result



Introducing a new command

- Let's introduce a new command
  `OMNIDRIVE {STRAIGHT s} {SIDE d} {ROT r}`

1. Add the relevant information to USARRemoteBot.uc
2. Parse the information in USARBotConnection.uc
3. Use it in ProcessCarInput

Modify USARRemoteBot

Add the following fields to the class USARRemoteBot

```
var float straightspeed;
var float sidespeed;
var float rotspeed;
```

Each instance of USARBotConnection has an instance of USARRemoteBot.

Modify USARBotConnection

- Inside the method ProcessAction:

```
case "OMNIDRIVE":
  if (GetArgVal("STRAIGHT")!="")
        urBot.straightspeed = float(GetArgVal("STRAIGHT"));
  if (GetArgVal("SIDE")!="")
        urBot.sidespeed = float(GetArgVal("SIDE"));
  if (GetArgVal("ROT")!="")
        urBot.rotspeed = float(GetArgVal("ROT"));
```

Modifying ProcessCarInput

```
function ProcessCarInput()
{   […]
    xspeed=USARRemoteBot(Controller).straightspeed;
    yspeed=USARRemoteBot(Controller).sidespeed;
    rot=USARRemoteBot(Controller).rotspeed*2;
    // solve kinematics; put results in s0,s1,s2
    JointsControl[0].state = 1;
    JointsControl[0].order = 1;
    JointsControl[0].value = s0;
    JointsControl[1].state = 1;
    JointsControl[1].order = 1;
    JointsControl[1].value = s1;
    JointsControl[2].state = 1;
    JointsControl[2].order = 1;
    JointsControl[2].value = s2;
    […]
```

Developing new sensors

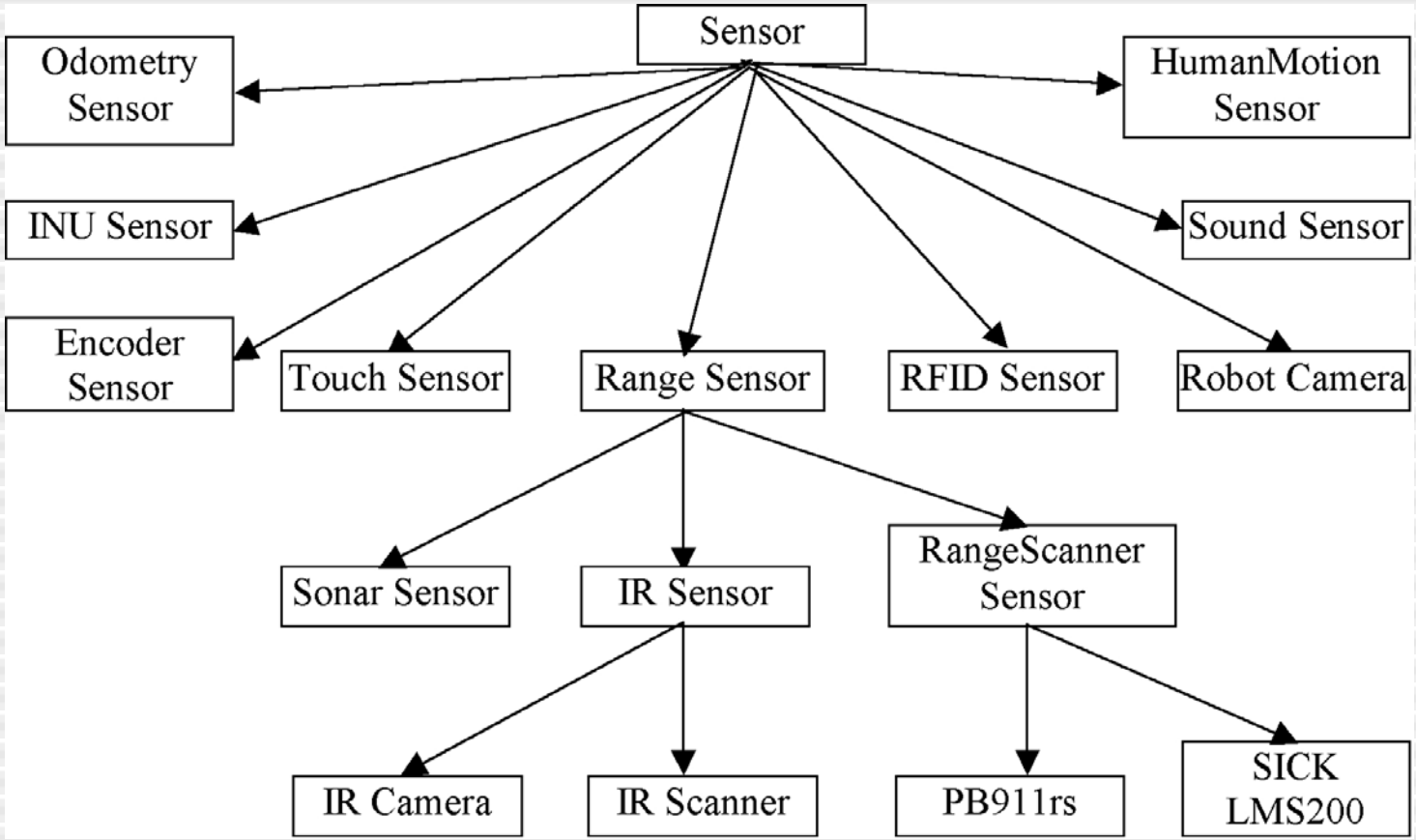
- All sensors inherit from the base USARSim *Sensor* class
  - *Sensor* itself inheriths from *Item*, i.e. it can be mounted on a robot
- Attributes:
  - HiddenSensor
  - OutputCurve
  - Noise

Sensor methods

- The following methods should be redefined

```
function String GetHead()
function String GetData()
function String GetGeoHead()
function String GetGeoData()
function String GetConfHead()
function String GetConfData()
```

Sensor hierarchy



Example: developing an RFID sensor

- A sensor that detects RFID tags scattered in the environment
  - Returns the position of the RFID relative to the sensors, and its ID
  - RFIDS are detected only within a certain range
  - Returns all the RFIDS currently detected by the sensor
  - Produce a message like:
    `SEN {Type RFIDTag} {Name` *`string`*`} {ID` *`int`*`}{Location` *`float, float, float`* `}`

Class sketch

```
class RFIDSensor extends sensor config(USARBot);

var config float MaxRange;
var float uuMaxRangeSquare;
var array<int> detectedTags;

[…]

defaultproperties
{
     MaxRange=0.50
     ItemType="RFID"
}
```

Class sketch (cont'd)

- Returning configuration data to the controller

```
function String GetConfData()
{
        local string outstring;
        outstring = Super.GetConfData();
        outstring = outstring@"{MaxRange "$MaxRange$"}";
        return outstring;
}
```

Class sketch (cont'd)

- Producing data

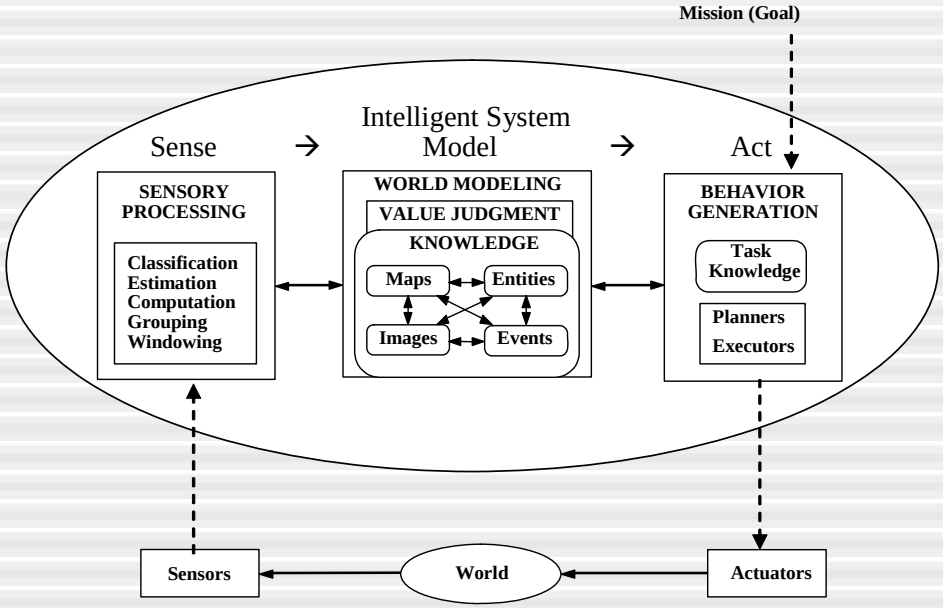
```
function String GetData()
{
   local string Outstring;
   […]
   Outstring = Outstring$" {Location
        "$converter.VectorString(pos, 2)$"}";
   […]
   if (Outstring!="") Outstring = ”
        {Name "$ItemName$"}"$OutString;
      return Outstring;
}
```

Session 4: MOAST Hands On

- MOAST control system overview
- Extending MOAST; an illustrative example
  - Creating new communications channels
  - Creating a new control process
  - Extending the diagnostics tools

Sense… Model… Act…

- Sensory processing (SP) populates the world model with facts
  - Based on raw data and prior results
- World modeling (WM) stores and provides access to information
  - Results of SP
  - Information on system self
  - General world knowledge and rules
  - Value judgments (cost/benefit analysis) based on information
- Behavior Generation (BG) computes possible courses of action based on:
  - Knowledge in the WM
  - System goals
  - Plan simulation
  - Plan execution



Architecture and Echelon Modularity

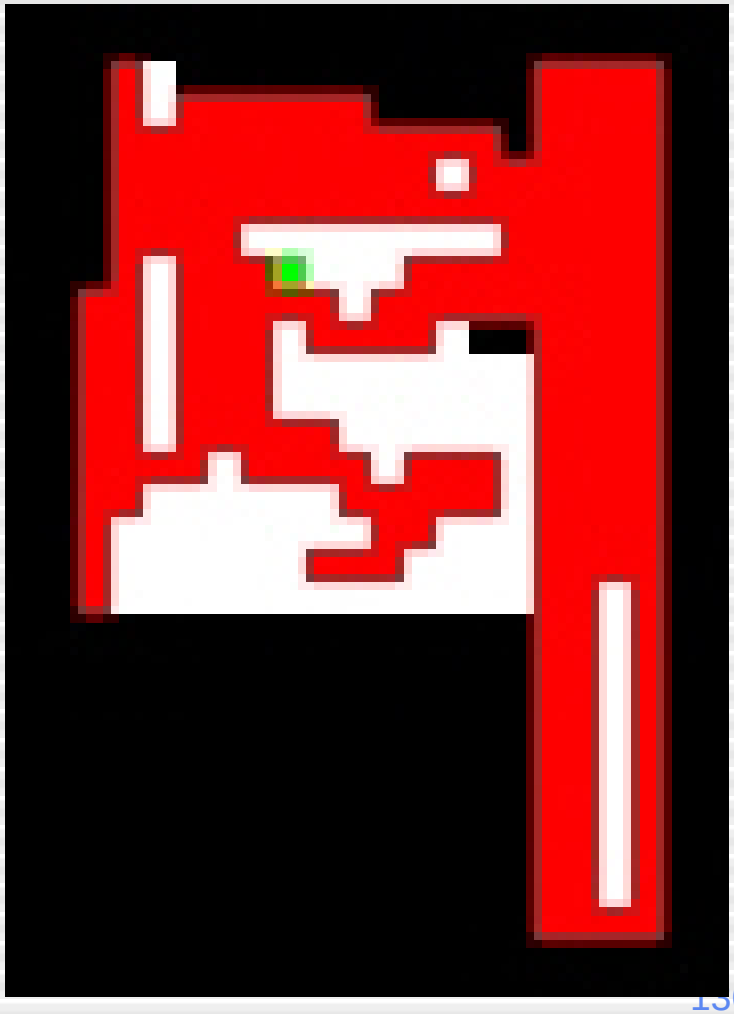
	Sensor Processing	World Modeling	Behavior Generation	
Section Echelon (Team)			Section BG Generates single mission phase for vehicles to execute	
Vehicle Echelon	Vehicle SP Perform grouping of cells and group attribution	Vehicle World Model (Repeated 3 Times) Attributed points, polylines, and polygons in global coordinates	Vehicle BG (Repeated 3 Times) Generates left, and right angles and period of scan.	Generates waypoints in global coordinates.
Autonomous Mobility Echelon	AM SP Apply labels to individual map cells	AM World Model Attributed cells in local coordinates	AM Mission BG Generates time-angle pair in sensor relative pan/tilt relative coordinates.	AM Mobility BG Generates arcs in local coordinates. The dynamics of the vehicle are here.
Primitive Echelon	Prim SP Output occupancy list in local coordinates	Prim World Model Occupancy map in local coordinates	Prim Mission BG Generates velocity-time curve where area under curve is delta position. 	Prim Mobility BG Generates wheel velocities in rad/sec. The kinematics of the vehicle are located here.
Below line is USARSim Servo Echelon	Servo SP Output 1 or 2D array of range values w/ time stamp	Servo World Model Absolute position of servos, readings from all vehicle sensors	Servo Mission BG Fix point position servo	Servo Mobility BG PID Control

Section Echelon

- This is what we will be building today
  - Section Echelon controls multiple vehicles
    - Long planning horizon
    - Infrequent replanning (10 second cycle)
    - Large extent, low resolution world model
  - Desired behavior is to coordinate the exploration of an unknown area

Section Echelon SP/WM

- SP will read vehicle map and categorize as explored or unknown
- WM needs to stitch together SP results to track unexplored areas



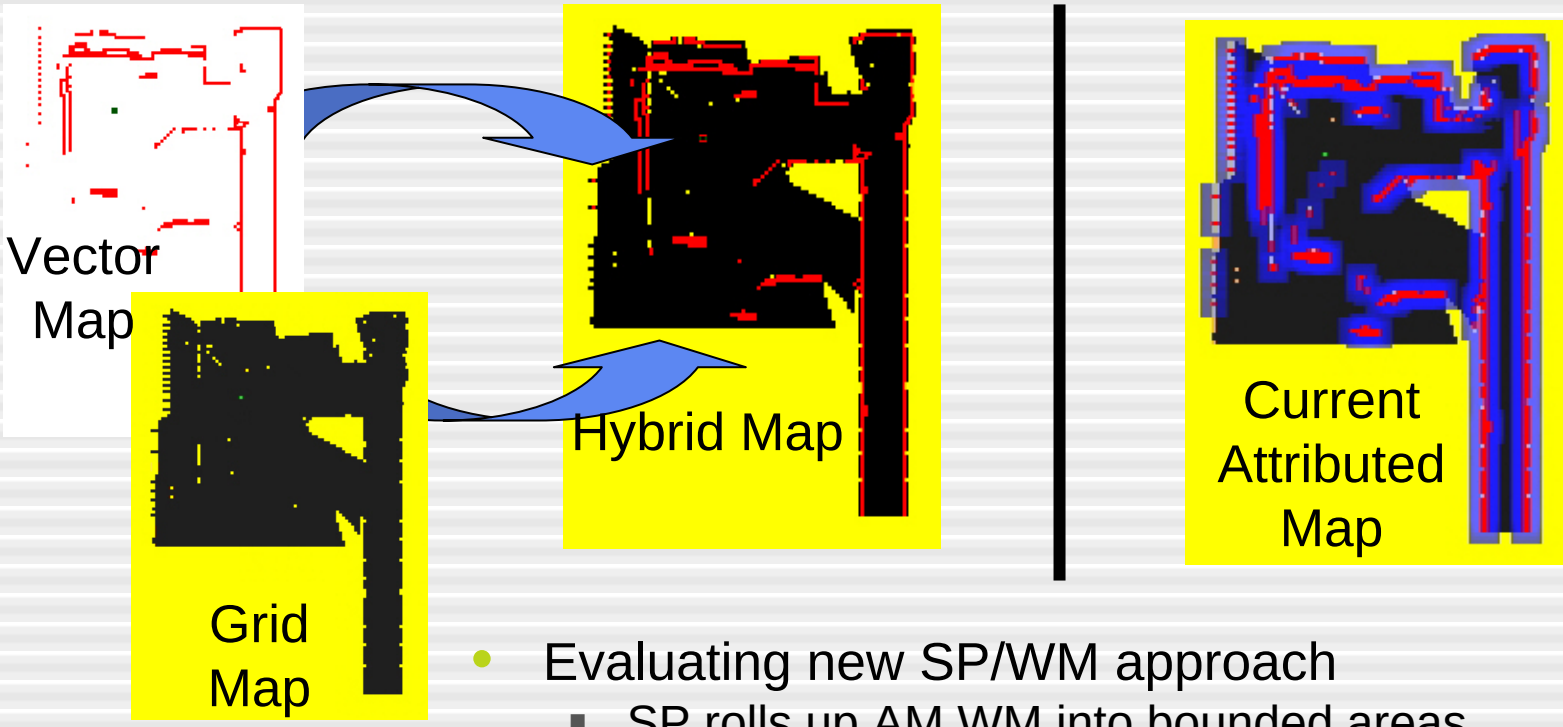
Section Echelon BG

- BG will need to coordinate multiple vehicles' motions
- Employ simple random waypoint control
  - Send each vehicle to a random unexplored point within radius *x* of its current location
- Replanning generates a new random point

Vehicle Echelon

- Controls a single vehicle via waypoint control
  - Adjustable cycle time (default 1 second)
- MOAST allows one to compare/contrast approaches. Current version:
  - Utilizes standard A* search in BG to find cost optimal path through WM
    - Optimized on avoiding proximity of obstacles while minimizing path length
  - SP summarizes AM WM into attributed occupancy map (obstacle, near obstacle, free, unknown)
  - WM based on standard grid with settable resolution and extents

Vehicle Echelon SP/WM



- Evaluating new SP/WM approach
  - SP rolls up AM WM into bounded areas
  - WM maintains attributed hybrid grid-vector model of world
    - Grid based known/unknown
    - Vector based obstacles

Vehicle Echelon BG

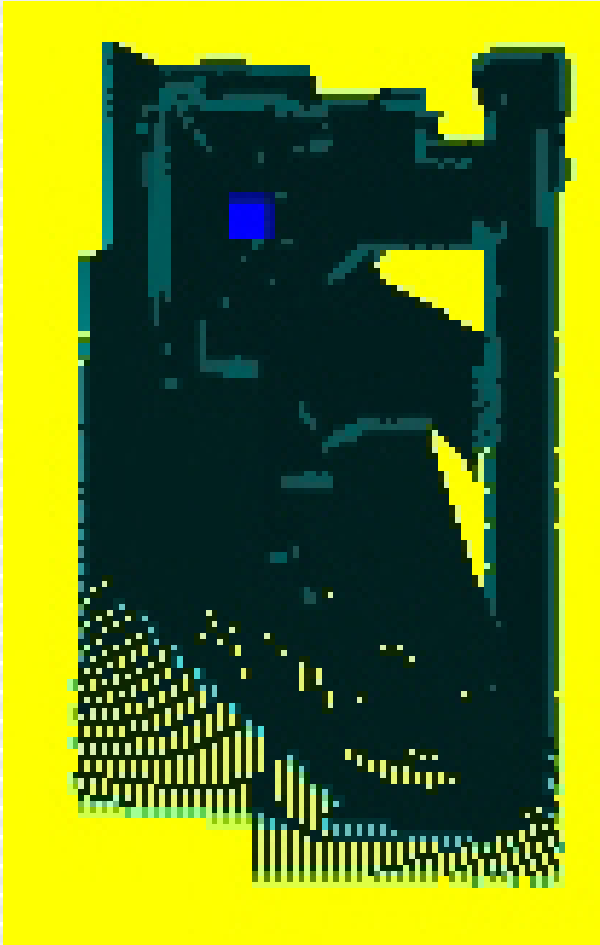
- New world model allows for BG search graph to be based on visibility graph instead of regular grid
  - Proven to generate “better” path segments
- Cost of path computed based on grids traversed
- Stay tuned for comparison results

Autonomous Mobility Echelon

- Computes vehicle trajectory that takes vehicle dynamics into account qualitatively:
  - Uses large-radius arcs to the extent possible
  - Reduces speed to keep lateral acceleration under half a G
  - Ground slope not yet used
- Copes with moving obstacles
  - Obstacle map updated every cycle
  - New plan made every cycle, but changes minimized when obstacles static
- Two BG planning methods
  - Rubberbanding identifies obstacles between waypoints, boxes them,
    plans around (and in and out of) the boxes
  - A*

Autonomous Mobility Echelon SP

- Reads data from Prim SP and updates large cellular map every cycle
  - Area 100 x 100 meters, 0.1 meter resolution (adjustable)
  - Cycle time 0.1 seconds (adjustable)
  - Vehicle-centered scrolling map
  - Each map cell contains: height, max range from which viewed, recent viewing history, obstacle probability (continuous)
  - Displays discrete obstacle probability map (optionally) every tenth cycle, with vehicle shown
- Writes data every cycle for AM BG and Vehicle SP
  - Area 10 x 10 meters, 0.1 meter resolution
  - Each map cell contains: obstacle probability (discrete)



Autonomous Mobility Echelon BG

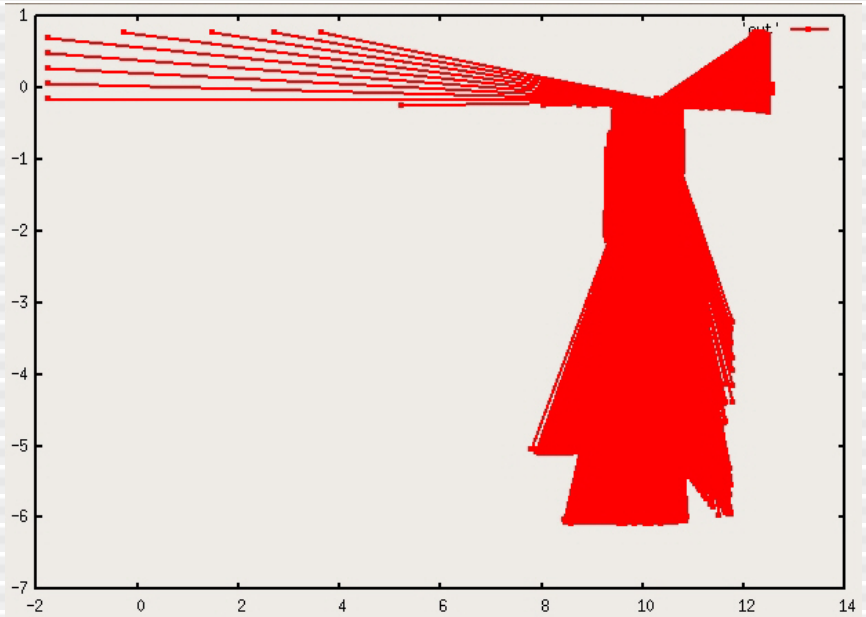
- Follows MOAST module template with tasks:
  - Generate and follow smooth path through given waypoints
  - Plan from current location to points on planning horizon (10 x 10 meters)
  - Do both simultaneously
- Paths
  - Are smooth sequences of arcs of circles and straight line segments
  - Have speed and tolerance for each arc and line segment
- Reads data from AM SP and updates cellular planning map every cycle
  - Area 10 x 10 meters, 0.1 meter resolution (adjustable)
  - Vehicle-centered, non-scrolling, discrete obstacle probability map
  - Obstacles grown; vehicle treated as a point when checking collisions
  - Map displayed (optionally) every cycle, with vehicle and planned path

Primitive Echelon

- The Primitive echelon (“Prim”) SP reads raw sensor data and creates sensor maps
- Prim BG transforms path goal segments from AM into periodic setpoints for the Servo echelon
  - Prim Mobility converts vehicle path segments along the ground into vehicle actuator commands
  - Prim Mission converts manipulator paths (e.g., robot arm or pan/tilt goals) into device actuator commands
- Prim BG does the kinematics transformations
  - Typically gets inputs non-periodically, runs periodically at ~ 10 milliseconds

Primitive Echelon SP

- Reads and processes data from Servo SP every cycle
  - Input data has multiple ranges (collected at equal increments of angle)
  - Output data has multiple pairs of points (sensor location, sensed location)
  - Cycle time 0.01 second
- Techniques
  - Processes points from straight ahead to sides (helps avoid disappearing walls)
  - Interpolates vehicle position and heading between points



Prim Mobility BG

- Prim Mobility plans instantaneous vehicle speed and heading, given a list of arcs and tolerances on arc width
- Prim models the vehicle dynamics to plan feasible speed and heading
- Speed and heading are transformed into actuator settings, such as wheel speeds, and sent to the Servo echelon

Prim Mobility BG Constraints

- In general, the problem is overconstrained; for example:
  - The desired speed for a small-radius arc may cause the vehicle to tip over
  - The desired heading may cause the vehicle to leave its tolerance region
- The speed constraint is relaxed first in favor of geometric constraints
  - A tight move is not made wider to maintain speed;
  - Rather, motion is slowed down to keep on the heading

Prim Mobility BG Sensing

- Planning is complicated by the vehicle's unpredictable performance, e.g.,
  - Slipping off the nominal path due to wet conditions
  - Skewing due to wheel hangups
- Prim continually monitor sensors, such as GPS and INS, to determine actual position
- Plans are continually regenerated to keep the vehicle on the arc, within tolerance

Prim Mobility BG Planning

- Prim converts each arc to a series of waypoints, depending on the tolerance
  - The tolerance sets a neighborhood around each waypoint
- For the next waypoint, prim computes speed and angular speed. As heading deviation increases,
  - Speed is reduced, to keep from leaving tolerance zone
  - Angular speed is increased, to move back to the desired heading

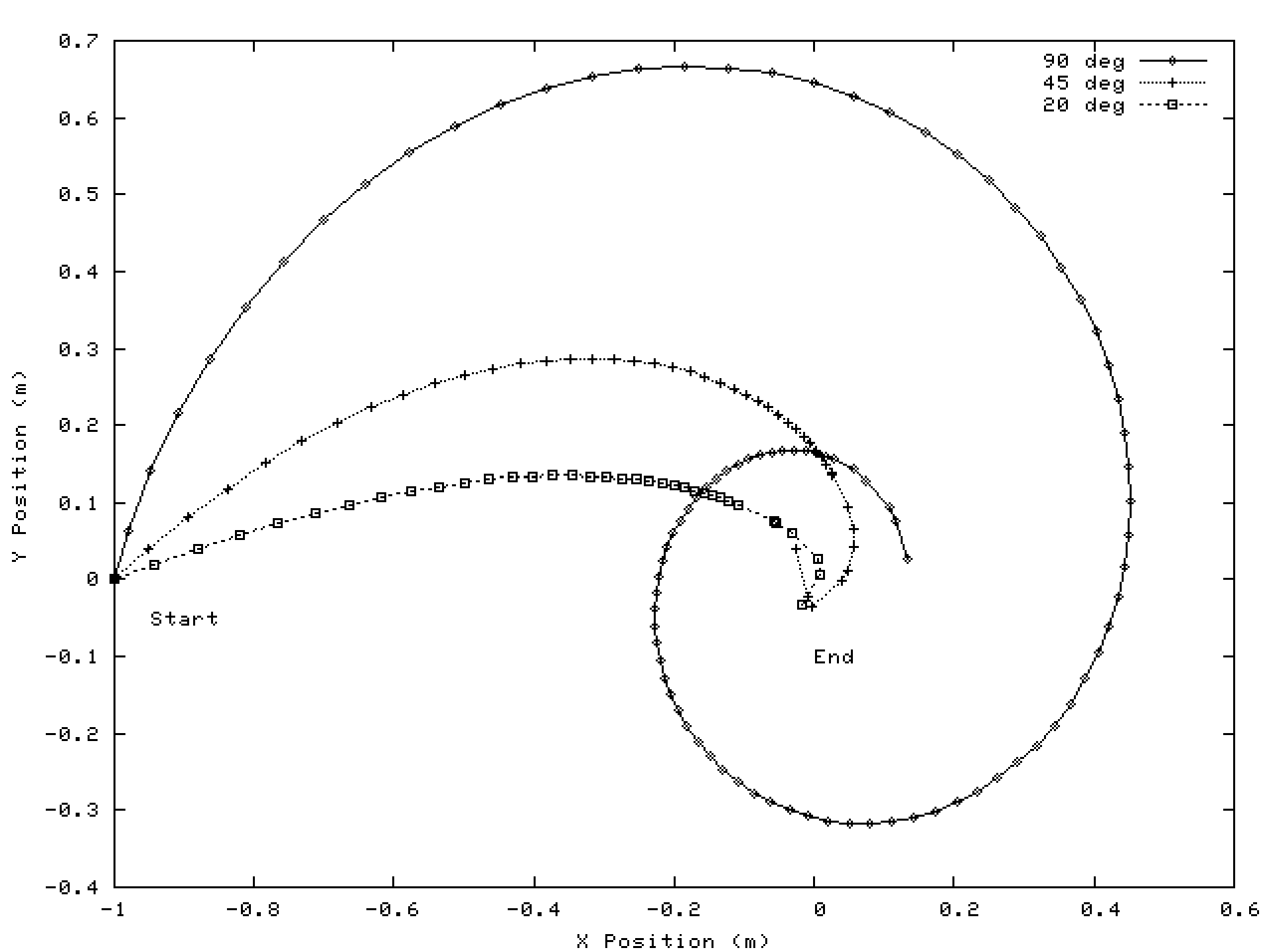
Prim Calculations

- Vehicle maximum speed v_m, angular speed ω_m are tunable, as is the cutoff angle θ_c above which speed is zero. Then for actual heading devation θ,

  speed $v = v_m (1 - |\theta| / \theta_c)$

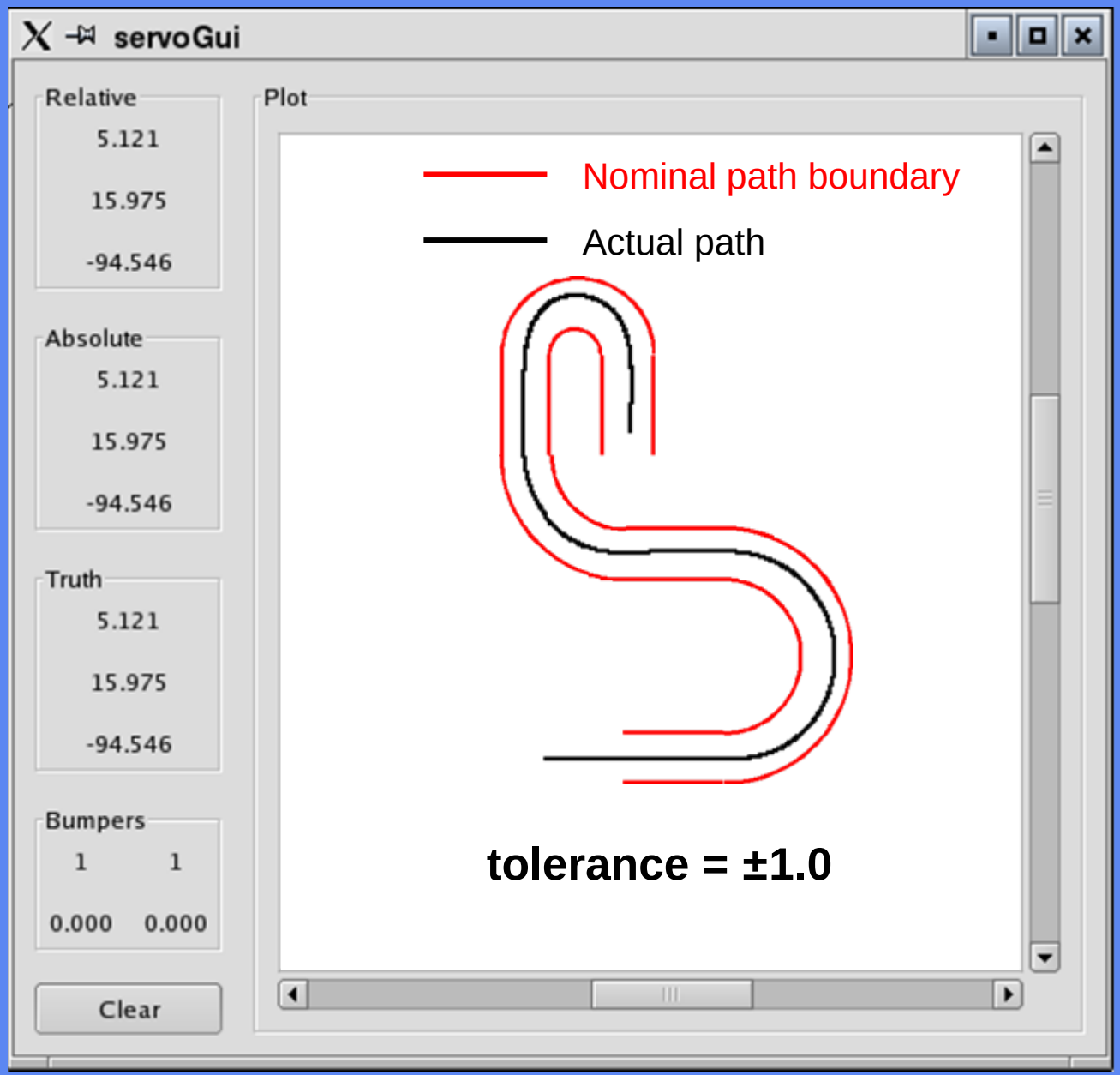
  angular speed $\omega = \omega_m (\theta / \theta_c)$

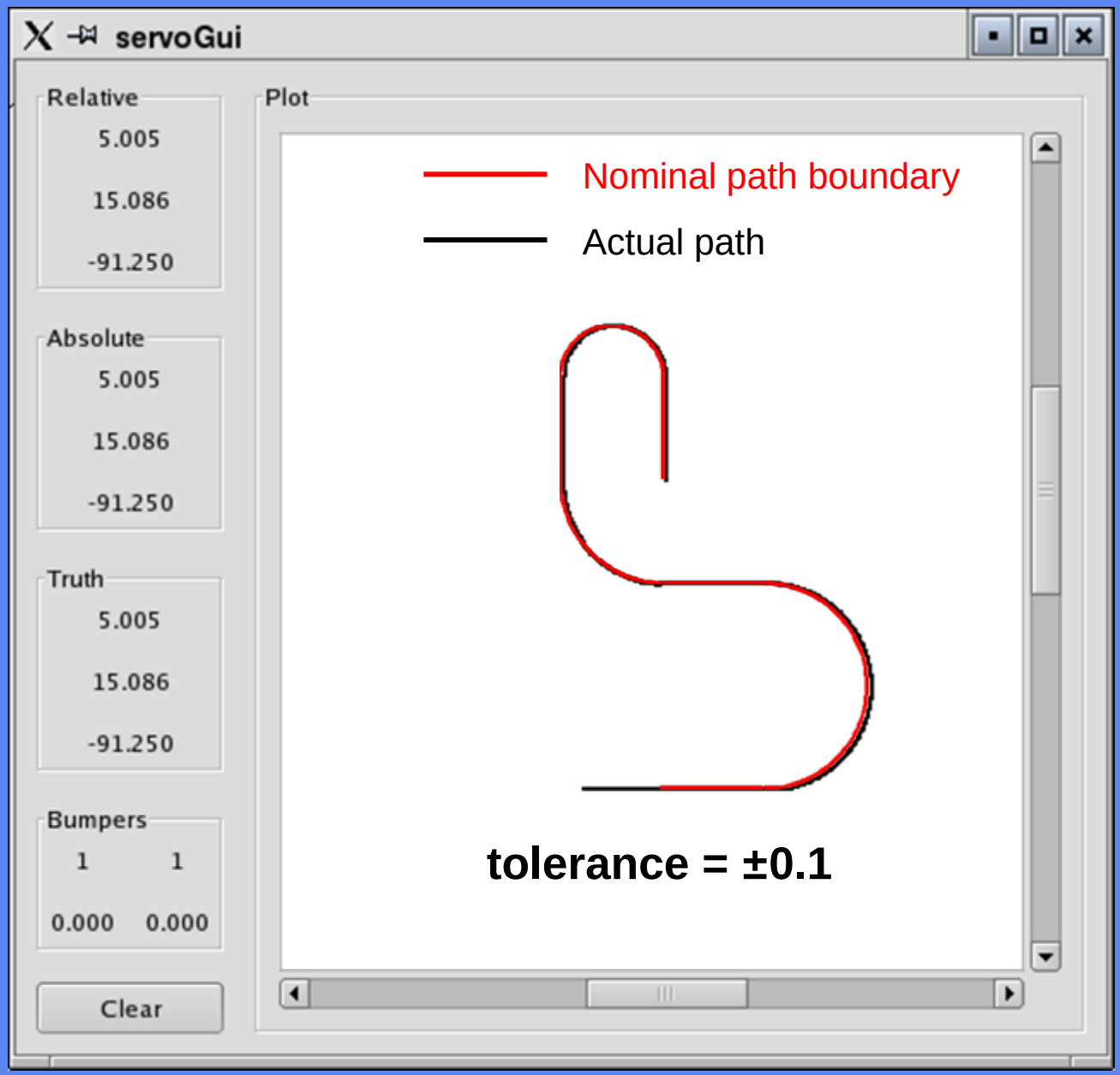
- Tends to move toward goal more tightly for smaller cutoff angles, as shown…


90 deg
45 deg
20 deg
Start
End
Y Position (m)
X Position (m)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6

Prim Performance

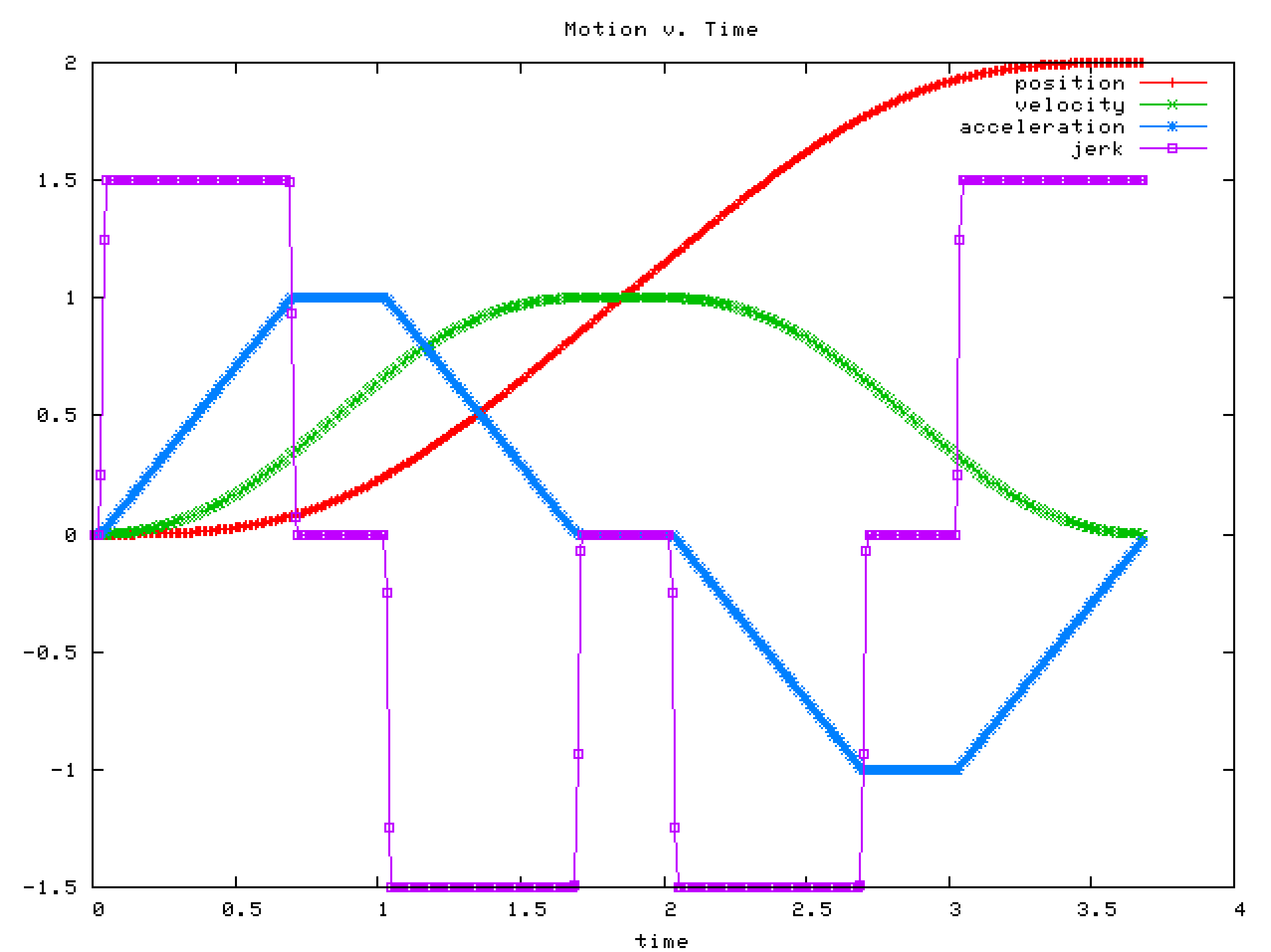
- MOAST allows for repeatable trials and the gathering of statistics
- Baseline Prim mobility algorithm can be replaced with others
  - Must honor input command and configuration messages, provide output status and settings messages
  - Must provide Servo echelon with appropriate actuator setpoints
- Performance can be compared with logging and plotting utilities, as shown…


servoGui
Relative
5.121
15.975
-94.546
Absolute
5.121
15.975
-94.546
Truth
5.121
15.975
-94.546
Bumpers
1
1
0.000
0.000
Clear
Plot
Nominal path boundary
Actual path
tolerance = ±1.0


servoGui
Relative
5.005
15.086
-91.250
Absolute
5.005
15.086
-91.250
Truth
5.005
15.086
-91.250
Bumpers
1
1
0.000
0.000
Clear
Plot
Nominal path boundary
Actual path
tolerance = ±0.1

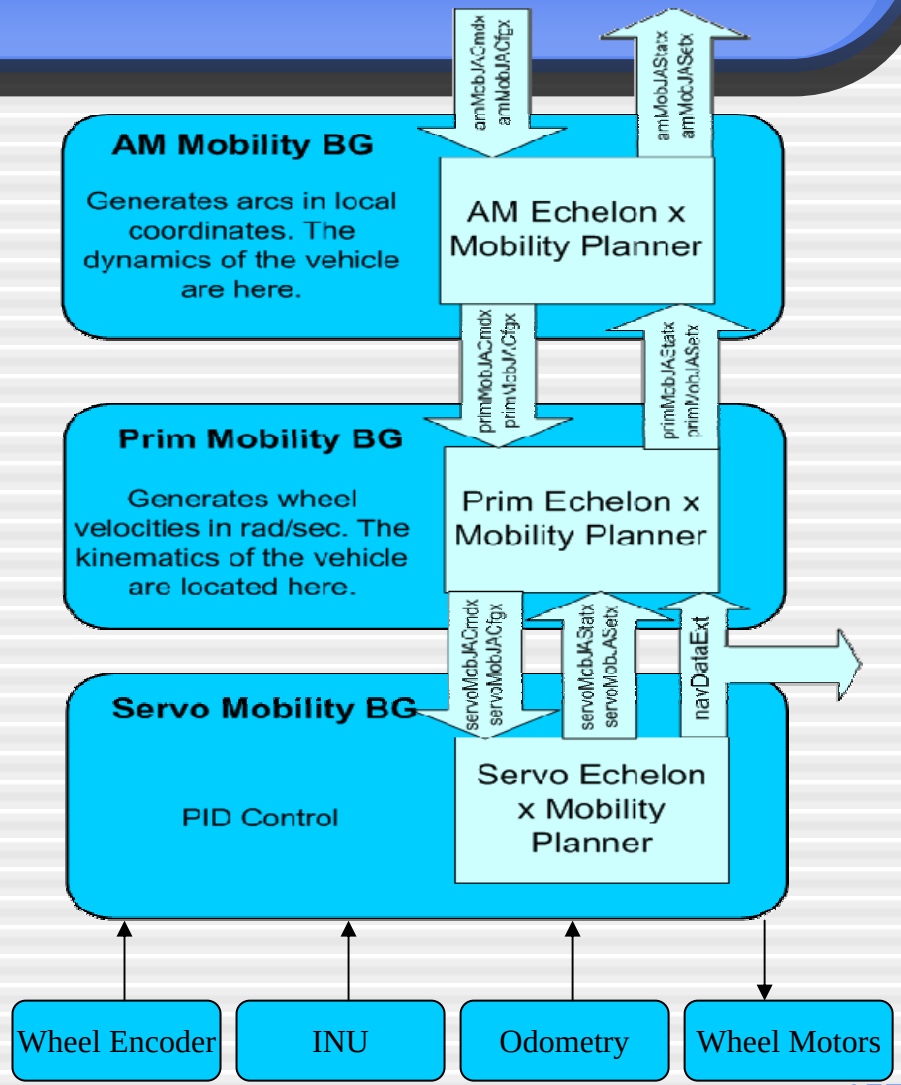
Prim Mission

- Prim Mission plans instantaneous manipulator position, given a list of target goal positions (motion segments) in Cartesian space
- Instantaneous Cartesian poses are transformed into actuator settings using the device kinematics
- Prim plans smooth acceleration, deceleration and blends successive motion segments, as shown…


Motion v. Time
position
velocity
acceleration
jerk
2
1.5
1
0.5
0
-0.5
-1
-1.5
0
0.5
1
1.5
2
2.5
3
3.5
4
time

Servo Echelon

- Low-Level of Abstraction
  - Joint-Level
  - Sensor Level
- Middleware between MOAST and underlying subsystems
  - Real or Virtual
- Intermediary mechanism that
  - Dynamically discovers vehicle subsystems
  - Creates internal representation of subsystem
  - Maps subsystems to NML channels
  - Syntactic and Semantic translation and interpretation of the command and control messages

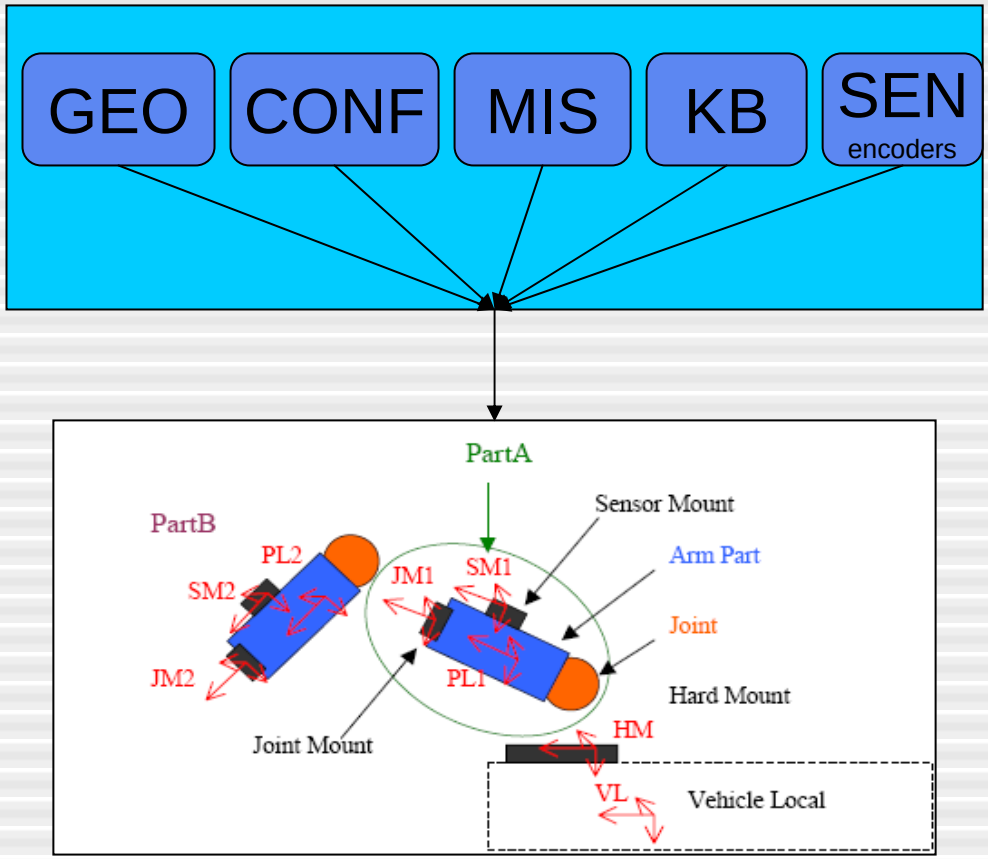


Servo Echelon Discovery

- Dynamically discovers vehicle subsystems
  - Issues GETGEO and GETCONF messages for all USARSim defined types:
    - Sensors
    - Effectors
    - Mission Packages
  - Obtains *a priori* knowledge from Knowledge Base (KB)
    - External knowledge repositories
    - Ontological models
  - *A priori* knowledge in KB supplements the *in situ* knowledge extracted from USARSim

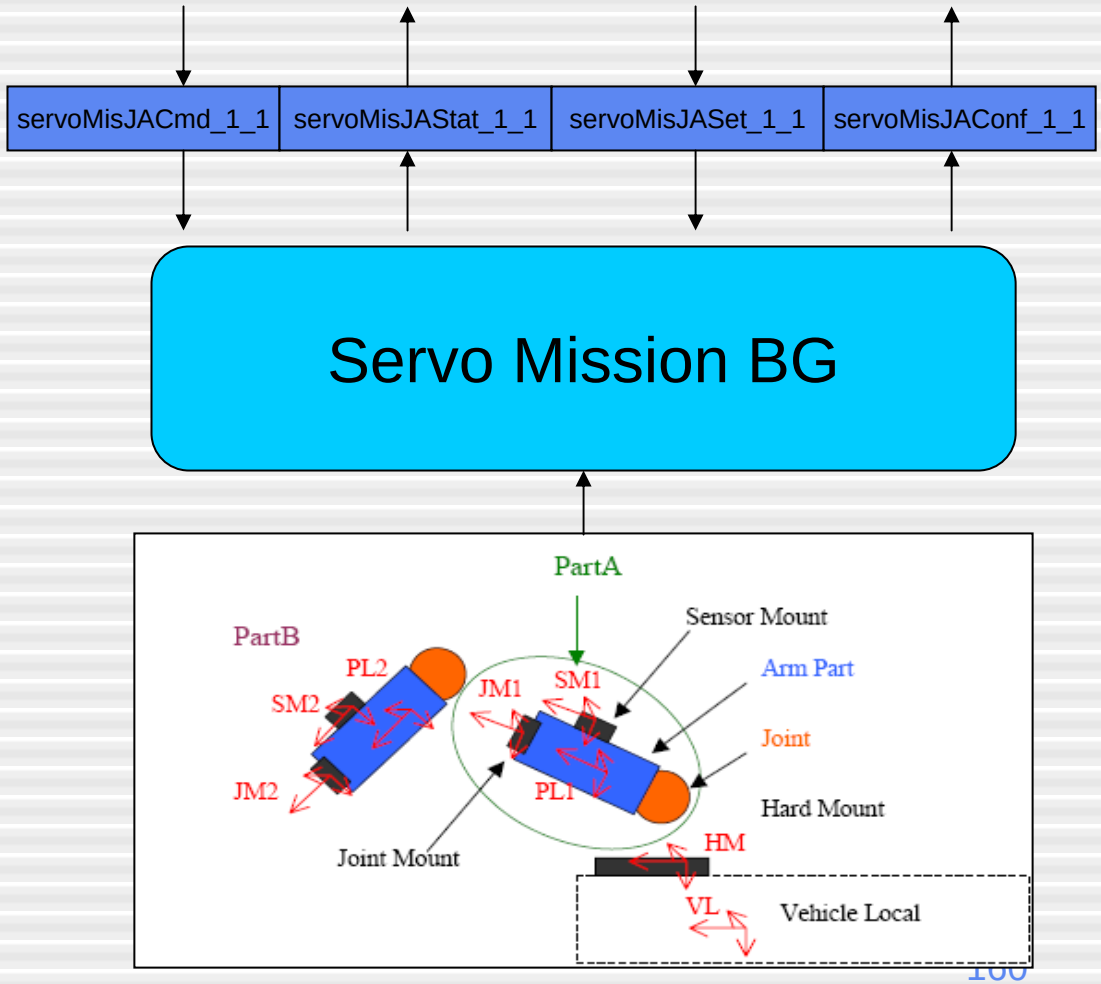
Servo Echelon Representation

- Creates internal representation of the subsystem
  - Combines *a priori* and *in situ knowledge* contained in KB to construct internal representation for all vehicle subsystems
  - Creates relationship between subsystems
    - Spatial
    - Mechanical



Servo Echelon Mappings

- Maps subsystems to NML channels
  - Constructs a strict one-to-one mapping from the vehicle subsystems to the appropriate NML channels.
  - Opens the appropriate NML channels for the subsystems using naming conventions
  - Naming Convention
    - Provides distinct, identifiable channels
    - Permits subsystem correlation and coordinated action in control system



Servo Echelon Translations

- Syntactic & Semantic translation and interpretation of the command and control messages
  - Coordinate Frame Transformation
  - Representation Transformation
  - Ontological Mapping

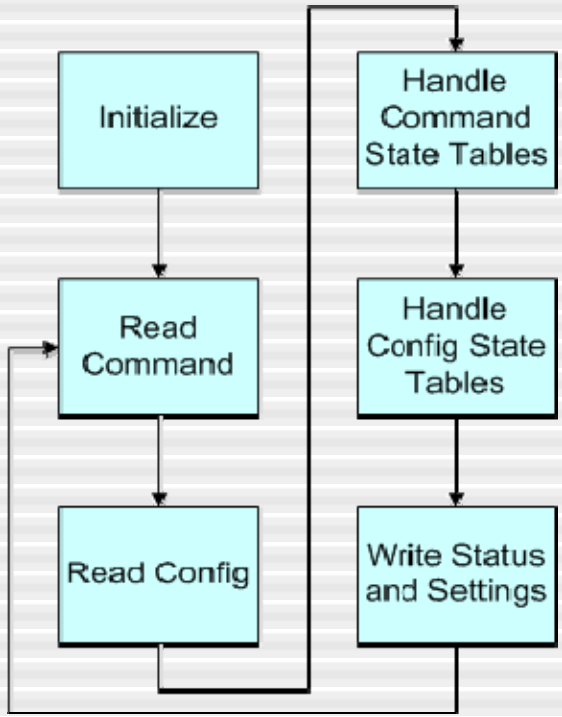
```
CONF {Type PanTilt} {ScanInterval 0.20} {Name Cam1} {Part
CameraBase Parent None} {Part CameraPan Parent None Mount
M_CameraPan_CameraTilt Location 0.0000,0.0000,-0.0719
Orientation 0.0000,-0.0002,0.0000} {Part CameraTilt Parent
CameraPan}
```

```
GEO {Type PanTilt} {Name Cam1 Location 0.1200,0.0000,-0.0826
Orientation 0.0000,-0.0002,0.0000} {Part CameraPan Location -
0.0000,0.0000,0.0279 Orientation 0.0000,0.0000,0.0000} {Part
CameraTilt Location -0.0000,0.0000,-0.0000 Orientation
0.0000,0.0000,0.0000}
```

```
// SERVO_MIS_JA_SET
//////////////////////////////////////////////////////
//! Denavit-Hartenberg Parameters
/*
 * DH Parameters defines manipulators as a set of joints and links.
 * The links are the bodies that connect neighboring joints.
 * !!Assumption: Each joint has one degree of freedom.
 */
class DH_Parameter {
public:
  static const int MAX_LINK_COUNT = 6;
  //! Defines DH Parameters Types
  enum DH_ParameterTypes {
    DH_PARAM_LINK_LENGTH,
    DH_PARAM_LINK_TWIST,
    DH_PARAM_LINK_OFFSET,
    DH_PARAM_LINK_JOINT_ANGLE
  };
  //! Defines the fixed parameters
  bool fixedParams[4]
  //! Defines the values for the fixed parameters.
  double fixedVal[4];
  //! Defines the range of the free variable
  double min, max;

};
/*!
  ServoMisJASet - reports the settings for mission package
 */
class ServoMisJASet : public RCS_STAT_MSG {
public:
  ServoMisJASet():RCS_STAT_MSG(SERVO_MIS_JA_SET_TYPE, sizeof(ServoMisJASet)) {};
  void update(CMS *);

  //! controller's cycle time period, [s]
  double cycleTime;
  //! debug level
  int debug;
  //! Pose relative to the vehicle
  PM_POSE misToVeh;
  //! Identifies the type of mission package
  MisPkgType type;
  //! Specifies the number of links
  int linkCount;
  //! Specifies the parameters for each link, where the first link is linkParam[0].
  DH_Parameter linkParam[DH_Parameter::MAX_LINK_COUNT];
  //! Specifies how many sensor are attached to mission package
  int sensorAttached;
  //! Identifies a sensor mounted on a link
  ObjLinkMap sensorMap[ObjLinkMap::OBJ_LINK_MAX];
  //! Specifies how many effectors are attached to mission package
  int effectorsAttached;
  //! Identifies a effector mounted on a link
  ObjLinkMap effectorMap[ObjLinkMap::OBJ_LINK_MAX];
};

extern int servoMisJA_format(NMLTYPE type, void *buf, CMS * cms);
extern const char * servoMisJA_symbol_lookup(long type);
```

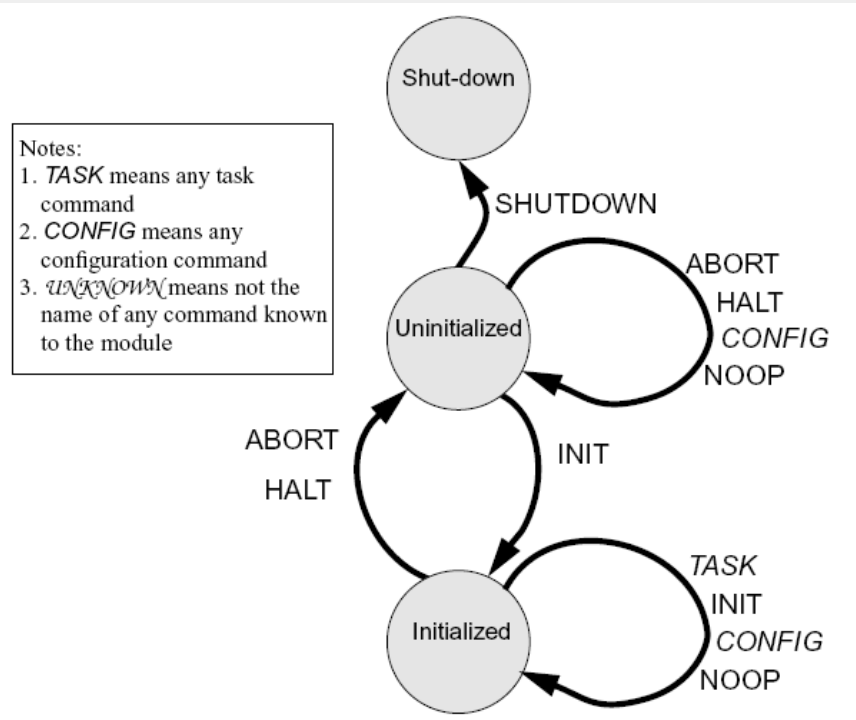
MOAST Processes

- All modules based on identical skeleton
- Modules implement two parallel state machines
  - Command → Status: Receipt of general commands and reporting of module status
  - Configuration → Settings: Receipt of parameter and configuration information and reporting of same
- Modules operate on fixed cycle time (complex state tables must be reentrant)
- Module state machines implement default commands and configurations
- State machines may call possibly reentrant auxiliary systems (e.g. cost based planners)



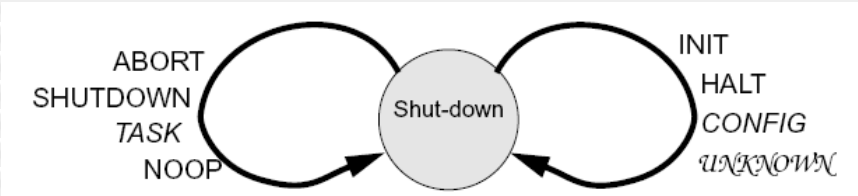
Administrative States

- All process must be in one of three administrative states
- Standard commands exist for navigating from state-to-state
  - NoOp – do nothing
  - Init – prepare to run
  - Halt – orderly stop
  - Abort – if possible, stop quickly and safely; if not, stop anyway
  - Shutdown – turn off process

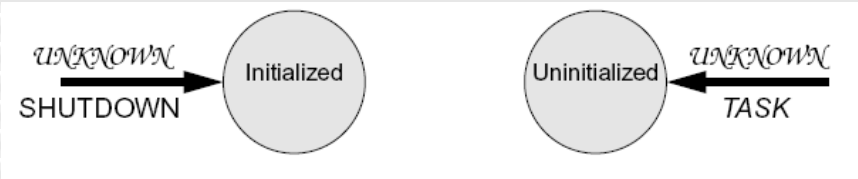


Receipt of Illegal Commands

- When the system is in the shutdown state, responses will not be given and commands will not be executed

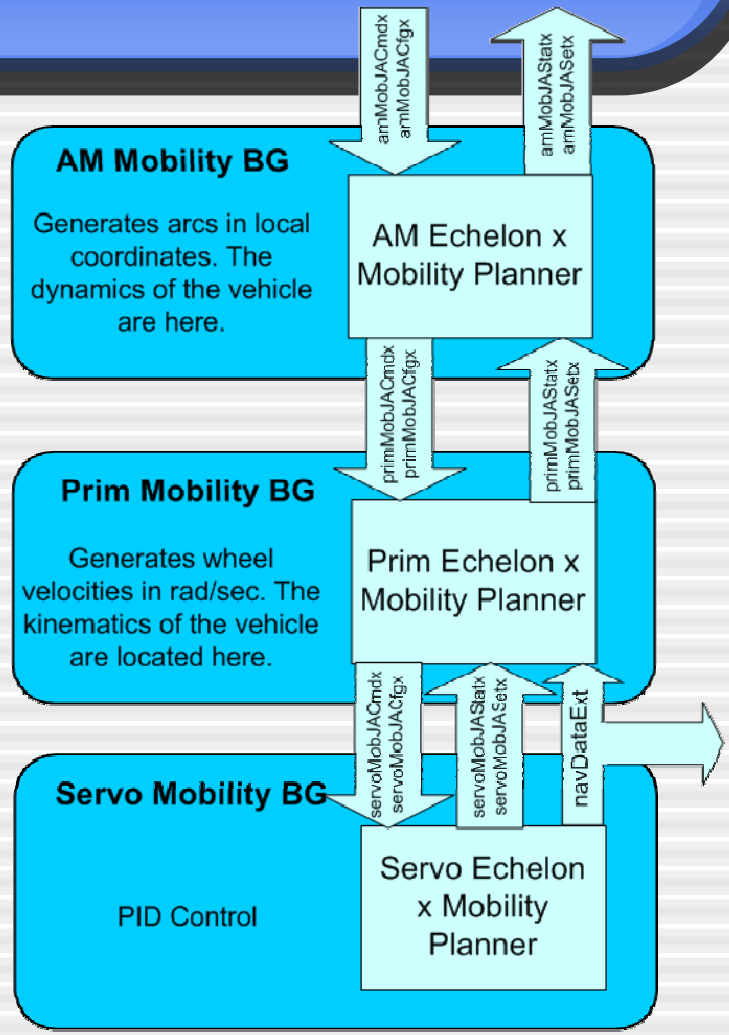


- When illegal commands are given, the command will not be executed and an error will be reported



Standard Interfaces

- Module-to-module communications performed via Neutral Messaging Language (NML)
- Each module command, status, configurations, and settings messages implemented as a separate class and defined in C++ header file(s)
  - Automatic tool to create necessary C++ or Java code
- Module may use a separate interface for data transmission



NML Nomenclature

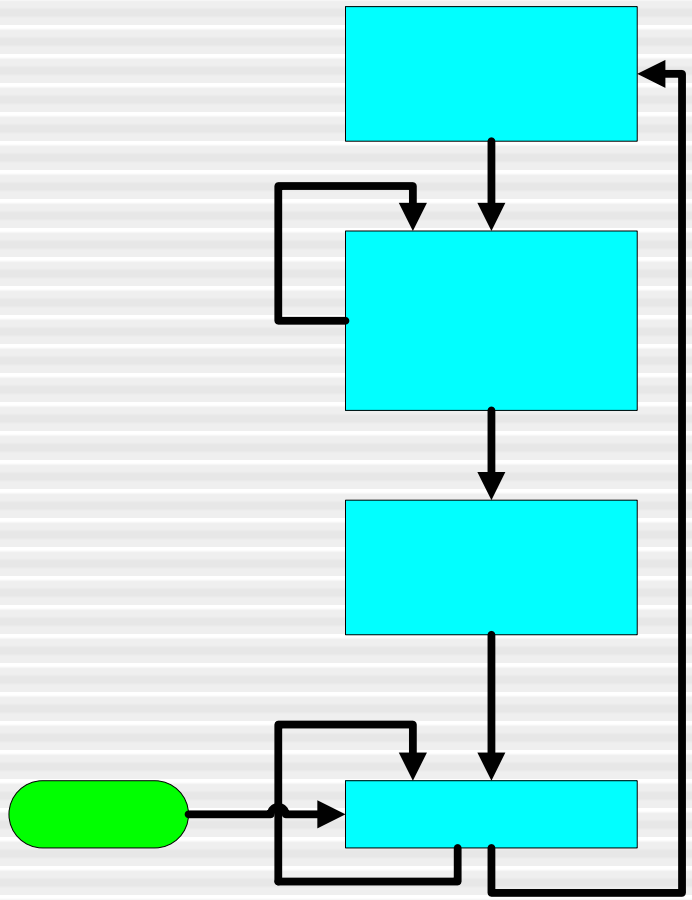
- Header file – a standard C++ header file that contains the message classes and definitions
- Configuration file – a text file that holds configuration information
- NML file – a file automatically generated from the configuration file that NML uses
- Buffer – a storage location with a fixed maximum size
- Process – a task that connects to a buffer
  - Task name – the name of the application
  - Process name – the name from the NML file

Creating A New Process

- We will now create a completely new process
- This process will require:
  - A definition of what the process will do
  - A state machine that carries out the procedures
  - Buffers for command input and status output
  - Buffers for subordinate data and status input
- **Please note** that code shown in this tutorial is only for illustrative purposes. For complete code details please see the sourceforge repository

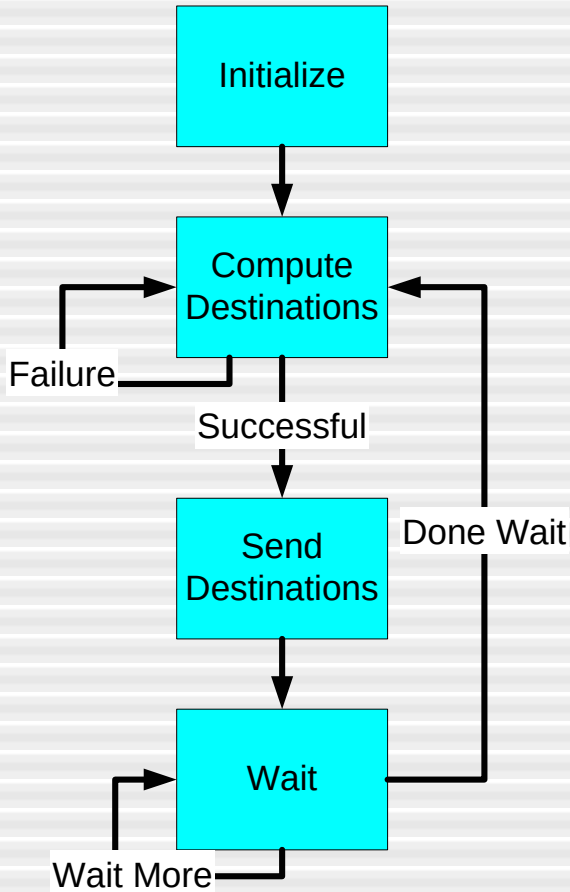
Section Echelon Process

- What this process will do:
  - Connect to some specified set of vehicles
  - Perform a mapping and exploration of an unknown area
  - Disconnect from the vehicles



Mapping & Exploration

- Very simple algorithm that generates constrained random destinations to send to each vehicle
- Constraints:
  - Destination must be within *x* m of vehicle
  - Destination must be unexplored
- System then waits some period of time before sending next destinations



Command & Status Buffers

- Command information
  - Will need to know which vehicles to command
  - How large an area each vehicle can explore
  - Constraints on the exploration
- Status information
  - In addition to standard status information, will want to provide the number of vehicles under control

Creating a Communications Buffer

- Skeleton communications channel provided in repository
- Consists of C++ header file with 3 main sections
  - General definitions
  - Class declarations
  - Convenience functions

General Definitions

Buffer names and command offsets

Commands defined in template

Unique command

Config. defined in template

Unique configuration

```
#define SECT_MOB_PL_CMD_NAME "sectMobPLCmd"
#define SECT_MOB_PL_STAT_NAME "sectMobPLStat"
#define SECT_MOB_PL_CFG_NAME "sectMobPLCfg"
#define SECT_MOB_PL_SET_NAME "sectMobPLSet"

#define SECT_MOB_PL_CMD_BASE        (SECT_MOB_PL_BASE)
#define SECT_MOB_PL_STAT_BASE       (SECT_MOB_PL_BASE + 100)
#define SECT_MOB_PL_CFG_BASE        (SECT_MOB_PL_BASE + 200)
#define SECT_MOB_PL_SET_BASE        (SECT_MOB_PL_BASE + 300)

#define SECT_MOB_PL_CMD_INIT_TYPE          (SECT_MOB_PL_CMD_BASE + 1)
#define SECT_MOB_PL_CMD_ABORT_TYPE         (SECT_MOB_PL_CMD_BASE + 2)
#define SECT_MOB_PL_CMD_HALT_TYPE          (SECT_MOB_PL_CMD_BASE + 3)
#define SECT_MOB_PL_CMD_SHUTDOWN_TYPE      (SECT_MOB_PL_CMD_BASE + 4)
#define SECT_MOB_PL_CMD_NOOP_TYPE          (SECT_MOB_PL_CMD_BASE + 5)
#define SECT_MOB_PL_CMD_EXPLORE_TYPE       (SECT_MOB_PL_CMD_BASE + 6)
#define SECT_MOB_PL_CMD_CONNECT_TYPE       (SECT_MOB_PL_CMD_BASE + 7)
#define SECT_MOB_PL_CMD_DISCONNECT_TYPE    (SECT_MOB_PL_CMD_BASE + 8)

#define SECT_MOB_PL_STAT_TYPE              (SECT_MOB_PL_STAT_BASE + 1)

#define SECT_MOB_PL_CFG_CYCLE_TIME_TYPE    (SECT_MOB_PL_CFG_BASE + 1)
#define SECT_MOB_PL_CFG_DEBUG_TYPE         (SECT_MOB_PL_CFG_BASE + 2)
#define SECT_MOB_PL_CFG_NOOP_TYPE          (SECT_MOB_PL_CFG_BASE + 3)
#define SECT_MOB_PL_CFG_CLEAR_WM_TYPE      (SECT_MOB_PL_CFG_BASE + 4)
#define SECT_MOB_PL_CFG_STORE_WM_TYPE      (SECT_MOB_PL_CFG_BASE + 5)
#define SECT_MOB_PL_CFG_LOAD_WM_TYPE       (SECT_MOB_PL_CFG_BASE + 6)
#define SECT_MOB_PL_CFG_DUMP_WM_TYPE       (SECT_MOB_PL_CFG_BASE + 7)

#define SECT_MOB_PL_SET_TYPE        (SECT_MOB_PL_SET_BASE + 1)

#define MAX_SECTION_VEHICLES 4
```

- Established communication buffer name convention
  - echelonModuleFunctionRole_vehId_subsysId
- Defines command numbers as offsets from a base number

Class Declarations

- Each command class:
  - Derived from RCS_CMD_MSG
  - Constructor includes command ID and size of message
  - Must include standard ‘update’ function
  - User defined class variables
  - Documented w/ Doxygen style comments
- Status class has same elements, but derived from RCS_STAT_MSG
- Dynamic array convenience function to limit transmitted data

```
/*!
  The SectMobPLCmdConnect NML message is used to connect to
  subordinate vehicles. The init flag is used to tell if we
  should initialize the subordinates
*/
class SectMobPLCmdConnect:public RCS_CMD_MSG {
public:
  SectMobPLCmdConnect():RCS_CMD_MSG
    (SECT_MOB_PL_CMD_CONNECT_TYPE,
     sizeof(SectMobPLCmdConnect)) {};
  void update(CMS *);

  //! IDs of vehicles that should be connected to
    /*!
     Variable length array of vehicles to command. The name of the array is
     "vehID", the type is int, and the maximum length is defined by
     MAX_SECT_VEHICLES.
     NML creates a variable named "vehID_length" that contains the number
     of vehicle IDs in the current message.
   */
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY(int, vehID, MAX_SECT_VEHICLES);
  //! should an init command be sent to subordinates (true is yes)
  bool doSendInit;
};
```

Convenience Functions

```
DECLARE_NML_DYNAMIC_LENGTH_ARRAY(int, vehID, MAX_SECT_VEHICLES);
```

```
extern int sectMobPL_format(NMLTYPE type, void *buf, CMS * cms);
extern const char * sectMobPL_symbol_lookup(long type);
```

- Dynamic length arrays limit amount of data transmitted
  - Parameters: data type, variable name, max length
  - Automatically generates variable “variable name”_length that user fills in with transmission array length
- Each header file must declare:
  - A format function that is used when opening the buffer and sets the buffer’s vocabulary
  - A symbol_lookup function that returns a human readable string when given a valid command id

Setting Up NML To Use The New Buffer

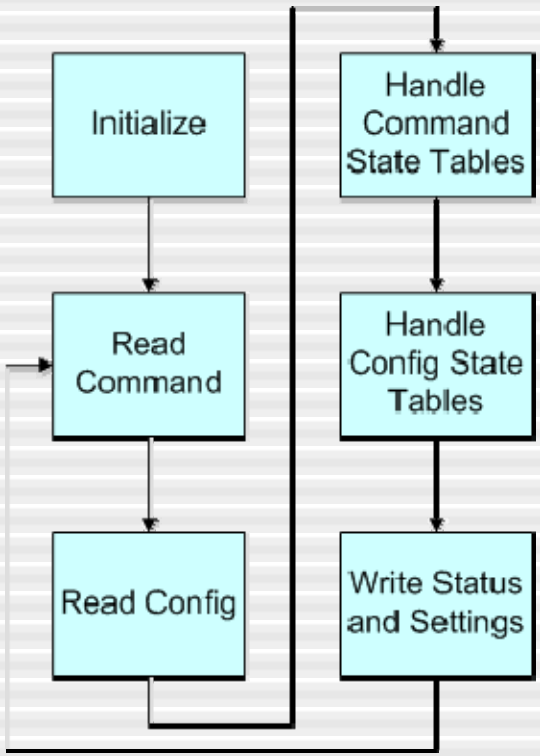
```
b bufname=sectMobPLCmd1 size=4096
```

```
p bufname=sectMobPLCmd1 name=sectMobPL1 master=0 server=0 proctype=local
```

- NML controlled by a configuration file
- Buffer lines define items such as the name, maximum size, and characteristics
- Process lines dictate who can connect to a buffer and the process's parameters

Setting Up Your Application

- Template provides basic functionality
  - Process flow
  - Connecting to buffers
  - Reading/writing buffers
  - Handling of administrative states
- You need to add your specialized commands



Process Initialization

- Allocate status and settings instances
- Initialize standard status message
  - heartbeat – beats every cycle
  - cycleTime – length of time previous cycle took
  - admin_State – the current administrative state
  - command_type – the current command
  - status – noop is always done
  - echo_serial_number – the serial number of the last command
  - message – used to report human readable information
- Initialize user provided status items
- Set channel pointers to NULL
- Next step: run basic process flow

```
rcsEngine::rcsEngine()
{
  int count;

  nmlStatus = new SectMobPLStat();
  nmlSettings = new SectMobPLSet();

  nmlStatus->heartbeat = 0;
  nmlStatus->cycleTime = DEF_CYCLE_TIME;
  nmlStatus->admin_state = ADMIN_SHUT_DOWN;
  nmlStatus->command_type = SECT_MOB_PL_CMD_NOOP_TYPE;
  nmlStatus->status = RCS_DONE;
  nmlStatus->echo_serial_number = 0;
  // user status initialization
  nmlStatus->numVehicles = 0;

  nmlSettings->echo_serial_number = 0;
  strcpy( nmlStatus->message, "" );

  // default cmd, stat, cfg, set buffers
  sectMobPLCmdNml = NULL;
  sectMobPLCfgNml = NULL;
  sectMobPLSetNml = NULL;
  sectMobPLStatNml = NULL;

  // user included buffers and command pointers
  for( count=0; count<MAX_SECT_VEHICLES; count++ )
    {
      navDataExtNml[count] = NULL;
      sensorDataNml[count] = NULL;
      vehicleStatNml[count] = NULL;
      vehicleCmdNml[count] = NULL;
    }
  vehicleSerialNumber = 0;
}
```

Connect To Buffers

- Open channels
  - Same signature for RCS_CMD_CHANNEL, RCS_STAT_CHANNEL, and NML
    - Format function
    - Buffer name – concatenation of name defined in header file with vehicle id
    - Process name – must match process line in NML file
    - File name –NML file to open
  - Check for success
- Open own channels, will wait to open subordinate's command channels
- Write out first messages (Noop)

```
bool rcsEngine::connect(const char *nmlFileName,const char *systemNMLName,
                        const char *nmlPostfix )
{
  SectMobPLCmdNoop *noopCmd;
  SectMobPLCfgNoop *noopCfg;
  char chanName[MOAST_NML_BUFFER_NAME_LEN];

  sprintf( chanName, "%s%s", SECT_MOB_PL_CMD_NAME, nmlPostfix );
  sectMobPLCmdNml = new RCS_CMD_CHANNEL( sectMobPL_format, chanName,
                                         (char*)systemNMLName,
                                         (char*)nmlFileName );
  if( NULL == sectMobPLCmdNml || !sectMobPLCmdNml->valid() )
    {
      rcs_print_error( "%s: Error - Can't open %s buffer\n",
                       systemNMLName, chanName );
      disconnect(1);
      return(false);
    }

  .
  .

  nmlStatus->admin_state = ADMIN_UNINITIALIZED;
  // stuff a noop cmd
  noopCmd = new SectMobPLCmdNoop();
  noopCfg = new SectMobPLCfgNoop();

  if( noopCmd <= 0 || noopCfg <= 0 )
    {
      printf( "Allocation error!!!\n");
      return false;
    }

  noopCmd->serial_number = nmlStatus->echo_serial_number+1;
  noopCfg->serial_number = nmlSettings->echo_serial_number+1;

  sectMobPLCmdNml->write(noopCmd);
  sectMobPLCfgNml->write(noopCfg);
  delete noopCmd;
  delete noopCfg;

  return true;
}
```

Read Input Buffers

- Run method performs basic housekeeping
  - Increment heartbeat, prepare for cycle time computation
- Reads command and configuration buffers
  - Simply uses the “read()” method
  - Captures serial number of incoming commands
- If necessary, reads subordinates’ data and status buffers

```
bool rcsEngine::runStateMachine(const char *systemNMLName)
{
  NMLTYPE cmdType, cfgType;
  int cmdSerialNumber, cfgSerialNumber;
  char newText[100];
  static double runTime = etime();
  int count;

  /* status housekeeping */
  nmlStatus->heartbeat++;
  nmlStatus->cycleTime = etime()-runTime;
  runTime = etime();

  // read command
  if( sectMobPLCmdNml != NULL )
    cmdType = sectMobPLCmdNml->read();
  else
    cmdType = 0;
  if( cmdType > 0 )
    cmdSerialNumber = sectMobPLCmdNml->get_address()->serial_number;

  // read configuration
  if( sectMobPLCfgNml != NULL )
    cfgType = sectMobPLCfgNml->read();
  else
    cfgType = 0;
  if( cfgType > 0 )
    cfgSerialNumber = sectMobPLCfgNml->get_address()->serial_number;

  // read data and status buffers
  // read each vehicle's location, sensor data, and status
  for( count=0; count<nmlStatus->numVehicles; count++ )
    {
      if( navDataExtNml[count] != NULL )
        updatePosition(count);
      if( sensorDataNml[count] != NULL )
        readVehicleMap(count);
      if( vehicleStatNml[count] != NULL )
        readVehCmdStatus(count);
    }
```

Input Buffer Housekeeping

- If a valid new command is received, the status command type must change to reflect this
- A new serial number on the command will change the status echo serial number, state, and status
- Use of serial number allows process to change parameters without resetting the command's state machine
- state and status use defined tags
  - state – NEW_COMMAND, S1, S2, …
  - status – RCS_EXEC, RCS_DONE, RCS_ERROR
- Notice use of symbol lookup function

```
// perform housekeeping on command and configuration
switch(cmdType)
  {
  case 0:
    // no new command
    break;
  case -1:
    // comm error
    disconnect(1);
    break;
  case SECT_MOB_PL_CMD_INIT_TYPE:
  case SECT_MOB_PL_CMD_ABORT_TYPE:
  case SECT_MOB_PL_CMD_HALT_TYPE:
  case SECT_MOB_PL_CMD_SHUTDOWN_TYPE:
  case SECT_MOB_PL_CMD_NOOP_TYPE:
  case SECT_MOB_PL_CMD_CONNECT_TYPE:
  case SECT_MOB_PL_CMD_DISCONNECT_TYPE:
  case SECT_MOB_PL_CMD_EXPLORE_TYPE:
    nmlStatus->command_type = cmdType;
    if(cmdSerialNumber != nmlStatus->echo_serial_number)
      {
        nmlStatus->echo_serial_number = cmdSerialNumber;
        nmlStatus->state = NEW_COMMAND;
        nmlStatus->status = RCS_EXEC;
      }
    break;
  default:
    fprintf(stderr, "%s: unknown command %s\n",
            systemNMLName, sectMobPL_symbol_lookup(cmdType));
    break;
  } // switch (cmdType)
```

Perform Command & Configuration

- Call the appropriate method to execute the command and configuration state tables
- doCmd*xx*() can perform complex actions
  - Must finish in one cycle's time or return after partial execution and be reentrant
  - Can implement non-state-based algorithms
    - Reentrant A* search has been implemented in the vehicle level

```
// perform command
switch( nmlStatus->command_type )
  {
  case SECT_MOB_PL_CMD_INIT_TYPE:
    doCmdInit();
    break;
  case SECT_MOB_PL_CMD_ABORT_TYPE:
    doCmdAbort();
    break;
  case SECT_MOB_PL_CMD_HALT_TYPE:
    doCmdHalt();
    break;
  case SECT_MOB_PL_CMD_SHUTDOWN_TYPE:
    doCmdShutdown();
    break;
  case SECT_MOB_PL_CMD_NOOP_TYPE:
    doCmdNoop();
    break;
  case SECT_MOB_PL_CMD_CONNECT_TYPE:
    doCmdConnect();
    break;
  case SECT_MOB_PL_CMD_DISCONNECT_TYPE:
    doCmdDisconnect();
    break;
  case SECT_MOB_PL_CMD_EXPLORE_TYPE:
    doCmdExplore();
    break;
  }

// perform configuration
switch( nmlSettings->command_type )
  {
  case SECT_MOB_PL_CFG_CYCLE_TIME_TYPE:
    doCfgCycleTime();
    break;
  case SECT_MOB_PL_CFG_DEBUG_TYPE:
    doCfgDebug();
    break;
  case SECT_MOB_PL_CFG_NOOP_TYPE:
    doCfgNoop();
    break;
  }
```

Write Out Buffers

```
  // write out buffers
  if( sectMobPLStatNml != NULL )
    {
      sectMobPLStatNml->write(nmlStatus);
    }
  if( sectMobPLSetNml != NULL )
    {
      sectMobPLSetNml->write(nmlSettings);
    }
  return true;
}
```

- Write out status and settings information
- Write out data (if any)
- Start cycle again…

What Happens Inside of Command

- Section of state table shown here
- Items to note:
  - Pick-up contents of command with “get_address”
  - Use of macros
  - Attention paid to time in each state to determine if next state should be executed

```
SectMobPLCmdExplore *sectExploreCmd =
  (SectMobPLCmdExplore*)sectMobPLCmdNml->get_address();

if( state_match(nmlStatus, NEW_COMMAND) ) // init
  {
    state_new(nmlStatus);
    state_default(nmlStatus);
    // check if we are allowed to execute this command
    if( nmlStatus->admin_state != ADMIN_INITIALIZED )
      {
        status_next(nmlStatus, RCS_ERROR);
        return;
      }
    // set wait counter to 0
    waitCounter = 0;
    // zero success flags
    for( count=0; count<nmlStatus->numVehicles; count++ )
      {
        // reset success flag
        vehicleSuccess[count] = false;
      }
    state_next(nmlStatus, S1);
  } // this state did not take long, so fall through
if( state_match(nmlStatus, S1) ) // compute destination
  {
    fail = false;
    for( count=0; count<nmlStatus->numVehicles; count++ )
      {
        if( vehicleSuccess[count] == false )
          {
            // computeDestination puts location in vehicleDestination[]
            if( computeDestination(count, sectExploreCmd->exploreDist) )
              vehicleSuccess[count] = true;
            else
              fail = true;
          }
      }
    if( !fail )
      state_next(nmlStatus, S2); // set all destinations
  } // this still did not take very long so fall through
```

Building The Buffers

- Communication buffers must have their C++ code autogenerated
- Autogeneration set in the file *$MOAST/devel/src/nml/Makefile.local*
  - Adding *myfile.hh* file to "**NML:**" list will cause *myfile.cc* to be automatically created
  - This file will be compiled and added to the *libmoast.a* library

Building The Process

```
bin_PROGRAMS = moastNmlSvr simpleSim usarSim servoShell primSPMain primMisMain primMobMain
primShell exampleUtmConvert gpsToSerial amSPMain amMisMain amMobMain amShell vehSPMain vehM
obPLMain vehShell sectShell sectMobPL exampleDijkTabs exampleDijkHeap exampleDijkHeapN exam
pleAStarHeapNA exampleAStarHeapNB spPlay nmlPrint inifind exampleVisGraph vehDrone cameraCn
trl nmlConvert
```

```
sectMobPL_SOURCES = ../src/section/sectMobPL.cc ../src/section/rcsSectMobPlEngine.cc
sectMobPL_LDADD = ../lib/libmoast.a -L$(RCSLIB_DIR)/lib -lrcs -lposemath -lm
```

- Two sections of the file *$MOAST/devel/bin/Makefile.am* must be modified
  - The first tells the makefile to build the process
  - The second specifies the process's dependencies.

Demonstration of Exploration

Resources

- USARSim on sourceforge
  www.sourceforge.net/projects/usarsim
  - Software, maps, documentation
- USARSim mailing list
  https://mailman.cc.gatech.edu/mailman/listinfo/robocup-rescue-v
- Virtual Robots competition homepage
  www.faculty.iu-bremen.de/carpin/VirtualRobots
- Additional & Updated tutorial material
  www.faculty.iu-bremen.de/carpin/ICRAtutorial

References

- Most of the USARSim technical details can be found in the *USARSim 2.0 reference manual*, by Jijun Wang, downloadable from sourceforge
- Some validation results appeared in

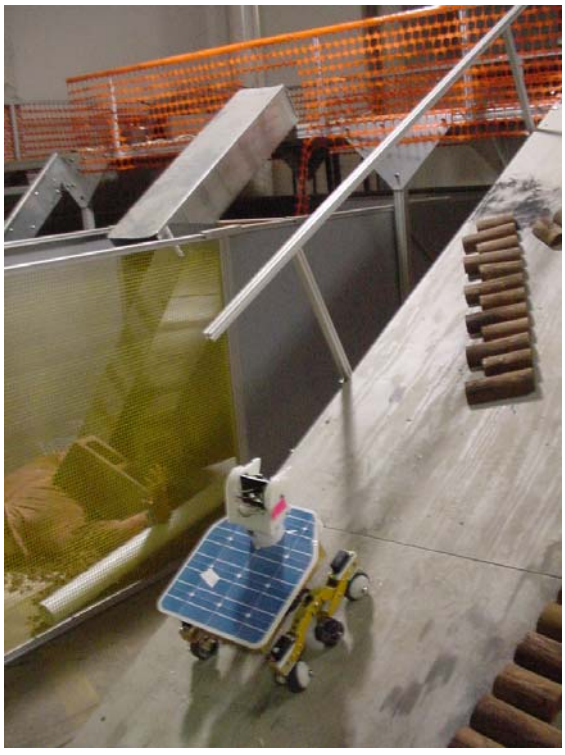
  J. Wang, M. Lewis, M. Koes, S. Carpin. "Validating USARSim for use in HRI Research". Proceedings of the 49th meeting of the Human Factors and Ergonomics Society, pp. 457-461, 2005
- More USARSim related research papers are available on http://usarsim.sourceforge.net/usar-pub.htm

Credits

- MOAST was funded in part by the DARPA MARS project, Doug Gage program manager and ARL, Chuck Shoemaker program manager
- Ravi Rathnam developed the Omnidrive robot model
- Denver Serrao provided validation results

USARSim V2.0.1

A Game-based Simulation of the NIST Reference Arenas



Prepared by Jijun Wang

USARSim

Contents

USARSim iii
Contents iii
1 Introduction 1
1.1 Background 1
1.2 What is USARSim 1
2 System Overview 2
2.1 System architecture 2
2.1.1 Unreal engine 3
2.1.2 Gamebots 3
2.1.3 Controller 3
2.2 Simulator components 4
2.2.1 Environment simulation 4
2.2.2 Sensor and effecter simulation 8
2.2.3 Robot simulation 9
3 Installation 9
3.1 Requirements 9
3.2 Install UT2004 9
3.2.1 Windows 9
3.2.2 Linux 9
3.3 Install USARSim 10
3.4 Install the controller 10
3.4.1 MOAST 10
3.4.2 Pyro 12
3.4.2.1 Windows 12
3.4.2.2 Linux 12
3.4.3 Player 12
4 Run the simulator 14
4.1 The steps to run the simulator 14
4.2 Examples 15
4.2.1 The testing control interface 15
4.2.2 MOAST 16
4.2.2.1 Configuration 16
4.2.2.2 Running MOAST 17
4.2.3 Pyro 17
4.2.4 Player 18
4.2.5 SimpleUI 19
4.2.5.1 Using SimpleUI by locally capturing pictures 19
4.2.5.2 Using SimpleUI by remotely receiving pictures 20

5 Coordinates, Units and Scale 21
5.1 Coordinates 22
5.2 Units and scale 23
6 Mission Package 23
7 Communication & Control (Messages and commands) 25
7.1 TCP/IP socket 25
7.2 The protocol 25
7.3 Messages 26
7.4 Commands 34
8 Sensors 38
8.1 State Sensor 39
8.1.1 How the sensor works 39
8.1.2 How to configure it 39
8.2 Range Sensor 39
8.2.1 How the sensor works 39
8.2.2 How to configure it 39
8.3 Range Scanner Sensor 40
8.3.1 How the sensor works 40
8.3.2 How to configure it 41
8.4 Odometry Sensor 42
8.4.1 How the sensor works 42
8.4.2 How to configure it 42
8.5 INU Sensor 43
8.5.1 How the sensor works 43
8.5.2 How to configure it 43
8.6 Encoder Sensor 44
8.6.1 How the sensor works 44
8.6.2 How to configure it 44
8.7 Touch Sensor 45
8.7.1 How the sensor works 45
8.7.2 How to configure it 45
8.8 RFID Sensor 45
8.8.1 How the sensor works 45
8.8.2 How to configure it 46
8.9 Victim RFID and False Positive Sensor 46
8.9.1 How the sensor works 46
8.9.2 How to configure it 46
8.10 Sound sensor 47
8.10.1 How the sensor works 47
8.10.2 How to configure it 47
8.11 Human-motion sensor 48
8.11.1 How the sensor works 48
8.11.2 How to configure it 48
8.12 Robot Camera 48
8.12.1 How the sensor works 48
8.12.2 How to configure it 50

9 Effecters 50
9.1 RFID Releaser 50
9.1.1 How the effecter works 50
9.1.2 How to configure it 50
9.2 Headlight 51
10 Robots 51
10.1 P2AT 51
10.1.1 Introduction 51
10.1.2 Configure it 53
10.2 P2DX 54
10.2.1 Introduction 54
10.2.2 Configure it 55
10.3 ATRVJr 55
10.3.1 Introduction 55
10.3.2 Configure it 56
10.4 PER (Rover) 57
10.4.1 Introduction 57
10.4.2 Configure it 57
10.5 Corky 57
10.5.1 Introduction 57
10.5.2 Configure it 58
10.6 Four-wheeled Car 59
10.6.1 Introduction 59
10.6.2 Configure it 59
10.7 Papagoose 59
10.7.1 Introduction 59
10.7.2 Configure it 60
11 Controller 60
11.1 MOAST 60
11.2 Pyro 63
11.2.1 Simulator and world 63
11.2.2 Robots 64
11.2.3 Services 65
11.2.4 Brains 66
11.3 Player 66
11.3.1 Simulation and device configuration 67
11.3.2 Device Drivers 68
11.3.2.1 us_bot 68
11.3.2.2 us_position 68
11.3.2.3 us_position3d 69
11.3.2.4 us_sonar 69
11.3.2.5 us_laser 69
11.3.2.6 us_ptz 70

12 Advanced User 70
12.1 Build your arena 71
12.1.1 Geometric model 71
12.1.1.1 Import an existing model *72*
12.1.1.2 Build it with Unreal Editor *72*
12.1.2 Special effects 72
12.1.3 Obstacles and Victims 73
12.2 Build your sensor 75
12.2.1 Overview 75
12.2.2 Sensor Class 76
12.2.3 Writing your own sensor 77
12.3 Build your effecter 77
12.3.1 Overview 77
12.3.2 Effecter Class 77
12.3.3 Writing your own effecter 78
12.4 Build your robot 78
12.4.1 Step1: Build geometric model 78
12.4.2 Step2: Construct the robot 78
12.4.2.1 Create the part/wheel class *78*
12.4.2.2 Create the robot class *79*
12.4.2.3 Prepare the attributes and objects used for your robot *79*
12.4.2.4 Connect the parts/wheels *80*
12.4.2.5 Mount the auxiliary items *83*
12.4.3 Step3: Customize the robot (Optional) 83
12.5 Build your controller 84
12.5.1 Embedding Unreal Client 85
12.5.2 Capturing Unreal Client 85
12.5.3 Using The Image Server 87
13 Bug report 87
14 Contributors 87
15 Acknowledgements 88

1 Introduction

This manual is written for version 2.0 of the USARSim. The files may be found at the files releases 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 source tar ball. Maps and tools are also available on the website.

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 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.

2 System Overview

2.1 System architecture

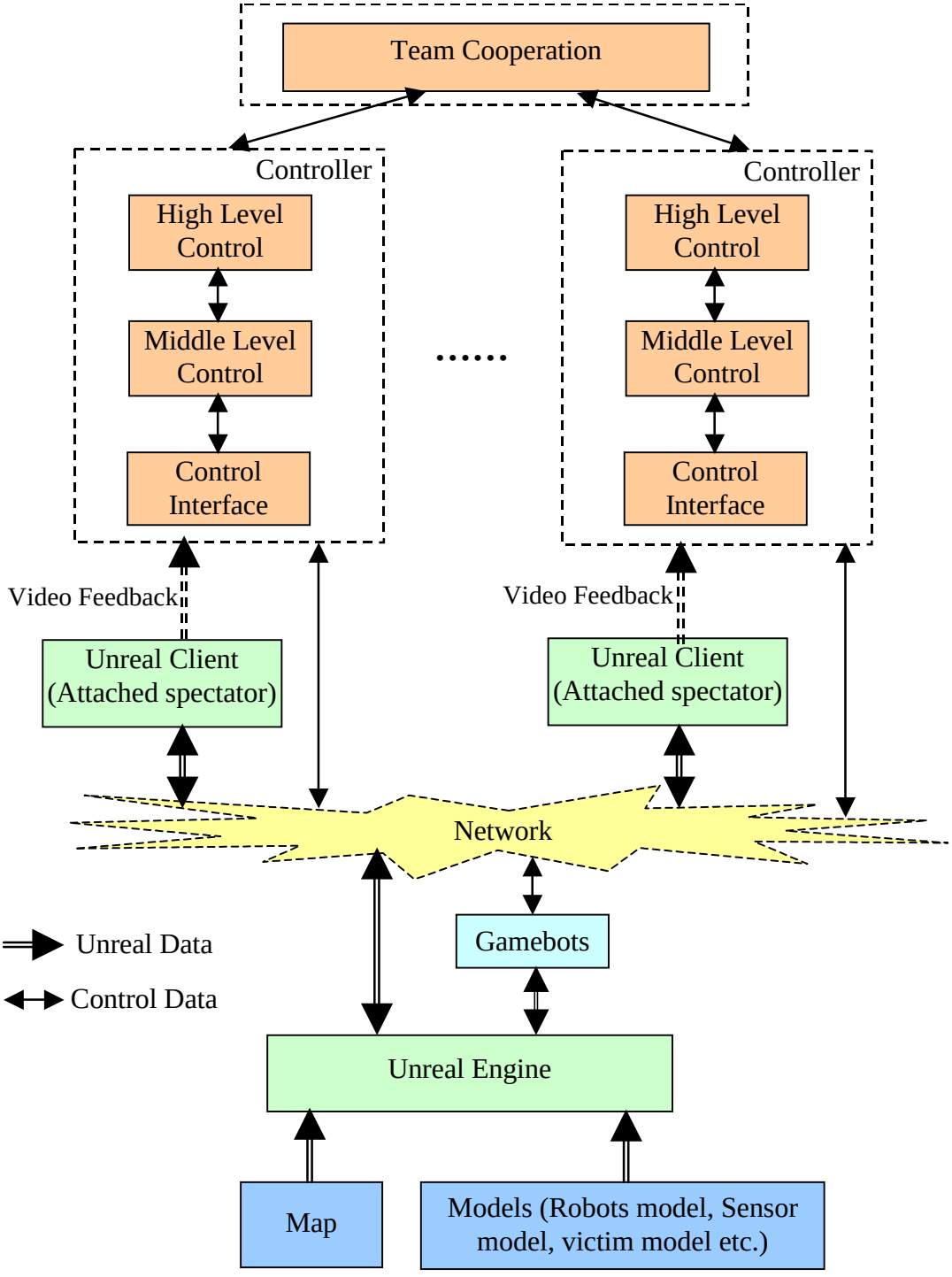


Figure 1 System Architecture

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.

2.1.1 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.

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 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

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's 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 11.1.

Besides MOAST, USARSim also supports two other popular robot controllers, Pyro 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 11.2.

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 11.3.

2.2 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.

2.2.1 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. 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 12.1. 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.

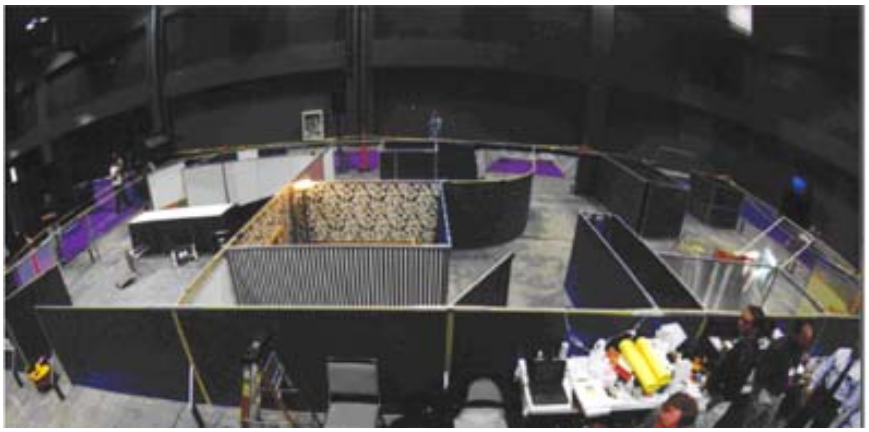


Figure 2 Yellow arena

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

2.2.2 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 8. For details of mounting a sensor on the robot please see section 10.

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 9. How to equip an effecter is explained in section 10.

2.2.3 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 12.4.

In USARSim, six robots are provided for you. They are Pioneer robots: P2AT and P2DX, iRobot ATRV-Jr, 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 in section 10.

3 Installation

3.1 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 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.

3.2 Install UT2004

3.2.1 Windows

1) Install UT2004.
2) Get the latest patch (http://www.unrealtournament.com/ut2004/downloads.php) at UT2004 official website. And then double click the file to install the patch.

3.2.2 Linux

1) Install UT2004
2) 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
        $ ut2k4-patch --help

3.3 Install USARSim

The installation is simple. You just need to download it from SourceForge http://sourceforge.net/projects/usarsim/ and then unzip it to the UT2004 installation directory. The base code set is located under the package usarsim-2004-core. Maps of different arenas are located under the Maps package.

> NOTE: In addition to the arena maps, the file *MapBasefiles-x.x.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.

3.4 Install the controller

This step is optional. Install it only when you want to use MOAST, Pyro or Player to control robot in USARSim.

3.4.1 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-compilied 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

3.4.2 Pyro

3.4.2.1 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.

3.4.2.2 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.

3.4.3 Player

Player is primarily used on Linux system and other POSIX platforms. But it does **NOT** run on Windows. To install Player 1.4 and our USARSim Player drivers:

1) Download Player 1.4rc2 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. The file "configure" and "configure.in" in the player.tar.gz should overwrite the original files in the ../player-src-1.4rc2 directory.
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>
```

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.

4 Run the simulator

4.1 The steps to run the simulator

Basically, running the simulator requires three steps.

1) Start the Unreal Server

   Go to UT2004/system directory, and then execute:

   ucc server *map_name*?game=USARBot.USARDeathMatch?TimeLimit=0 –ini=USARSim.ini –log=usar_server.log

   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.

2) Start the Unreal Client

   Go to the UT2004/system directory, and then execute:

   ut2004 ip_address*?spectatoronly=1?quickstart=true –ini=USARSim.ini*

   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.

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

3) Start the Controller

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

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 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.

4.2 Examples

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

4.2.1 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 4.1 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 USARCar, USARBc, P2AT, P2DX, ATRVJr and Rover. 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.

4.2.2 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.

4.2.2.1 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.

4.2.2.2 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).

4.2.3 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.[1]
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 11.2.

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.

4.2.4 Player

Player is a device server. So before you use Player, you need to start USARSim. Please follow step 1 and 2 in section 4.1 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 11.3.1. 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.

[1] 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 10.

4.2.5 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 8.12 and 12.5. In SimpleUI, we simply display the video pictures without any image processing. How to run SimpleUI in local or remote mode is introduced below.

4.2.5.1 Using SimpleUI by locally capturing pictures

The steps of running SimpleUI are :

1) Start Unreal Server (see step 1 in section 4.1).
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.
    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. ‘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.
    c. ‘Wheels’ group
    This group controls how the robot moves. The arrow buttons control the robot in the corresponding direction with a fixed speed. The ‘Speed Up’ and ‘Speed Down’ buttons speed up or slow down the robot’s moving speed. The ‘Stop’ button stops the robot.
    d. ‘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.
    e. ‘Light’ group
    The ‘Turn On’/‘Turn Off’ button turns the headlights on or off.

4.2.5.2 Using SimpleUI by remotely receiving pictures

To run SimpleUI in remote mode, we need to start image server before we launch SimpleUI. The steps of starting imager server are:

1) Start Unreal Server (see step 1 in section 4.1).
2) Execute ImageSrv.exe, which is under the ut2004/Tools/ImageSrv/Release directory, to fire the Image Server. This may be downloaded from the tools section of the sourceforge files area and should be installed as indicated.
3) On the user interface, specify the following parameters:
    a. ‘Command’
    ‘Command’ is the command used to launch Unreal Client. For details please read step 2 in section 4.1.
    b. ‘Image’ group
    This group sets the video properties. ‘Format’ is the picture’s format. It can be raw data or jpeg data with different qualities. ‘Resolution’ is the picture’s size. ‘Frame Rate’ is the maximum video frame rate.
    c. ‘Network’ group
    In this group, we set the socket port number used to transferring pictures. The ‘Client’ text field shows all the connected clients. As a simple client manager, you can select one or more clients and use the ‘Kill Selected Client(s) button to disconnect them.

4) On the interface, click the ‘Start’ button. This will launch the Unreal Client and a message box that instructs you how to switch to the Unreal Client to set the viewpoint and switch back to the image server. After you set the viewpoint, click the ‘OK’ button on the message box which will trigger the image server to hide the Unreal Client and try to capture and send out the pictures.

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 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 5.2

5.1 Coordinates

Unreal Engine uses a left-hand coordinate system (Figure 8). 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.

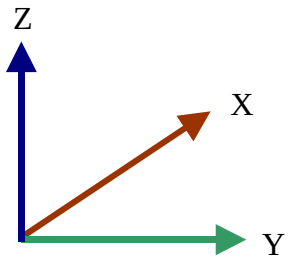


Figure 8 Left Hand Coordinate System

In the application interface, we use the right hand SAE J670 Vehicle Coordinate System (Figure 9). The only difference from the Unreal Engine coordinate system is that the positive Z-axis points straight down and not up.

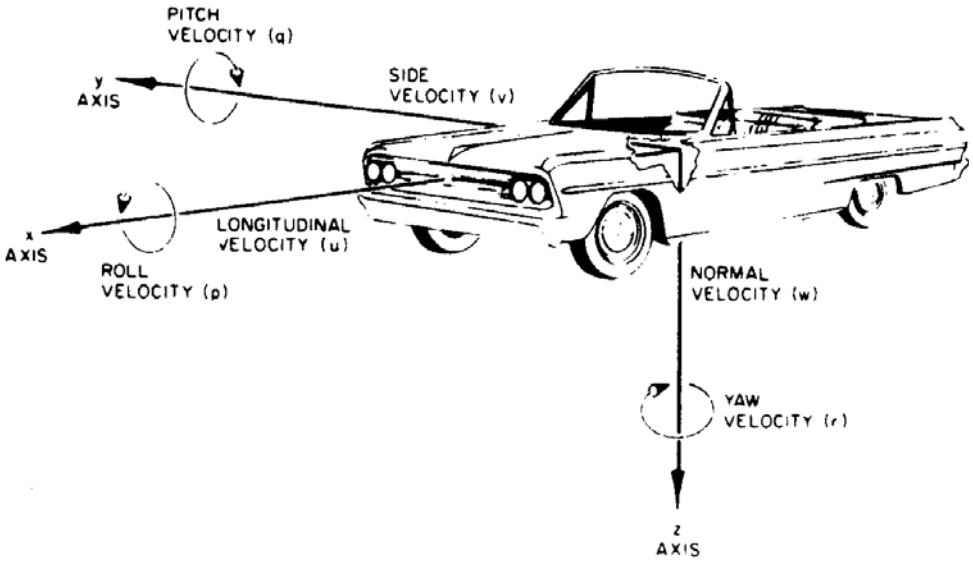


Figure 9 SAE J670 Vehicle Coordinate System

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.

5.2 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:

256 UU = 487.68 cm = 16 feet
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.

6 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 10. 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 11 is a more general example that depicts an arm mission package.

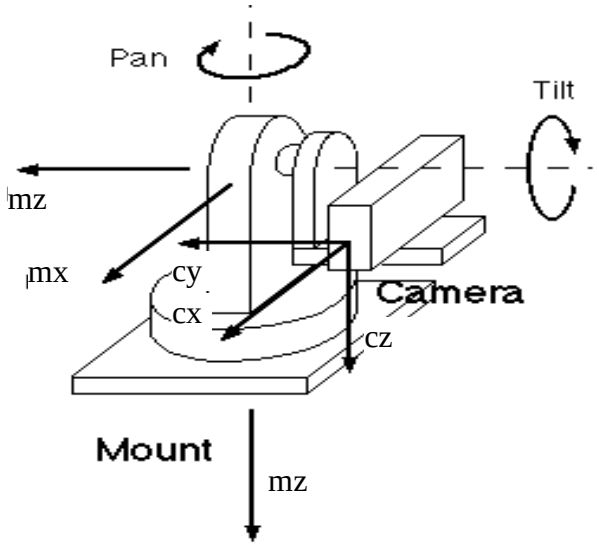


Figure 10 Mission package and its coordinates

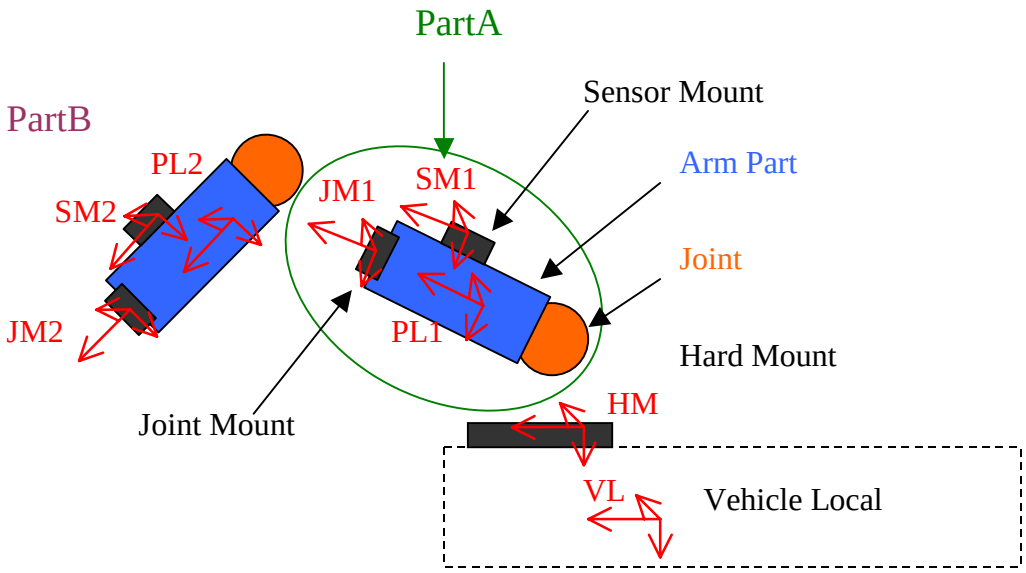


Figure 11 Arm mission package

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 10, we will give detailed instructions. A mission package's state is described as a series of the part's relative location and orientation to its mount. We use the MIS message to deliver this information. This message is explained further in the next section.

To facilitate mission package control, we classify mission packages into mission package types according to the package's structure and how we control it. For example, a PanTilt type mission package is a package that is constructed of two parts and controlled by panning the first part and tilting the second part. A Flipper type mission package is a package constructed of one part and controlled by tilting the part.

7 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.

7.1 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.

7.2 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.

7.3 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 sensor, effecter, or mission package's geometry information. Configuration messages give the sensor, effecter, or mission package's configuration information, and the response message provides the sensor/effecter's response status to a command.

- State and Mission Package message

  A robot state message looks like:

  STA {Time *float*} {Location *x,y,z*} {Orientation *x,y,z*} {Velocity *x,y,z*} {LightToggle *bool*} {LightIntensity *int*} {Battery *int*}

  Where:

{Time *float*}	'float' is the UT time in seconds. It starts from the time the UT server starts execution.
{Location *x,y,z*}	Position of the robot. The values are positions in the SAE J670 coordinate system in meters.
{Orientation *x,y,z*}	The attitude of the robot. The values are the rotation angle around x, y and z-axis in radians.
{Velocity *x,y,z*}	The velocity of the robot. The values are speeds in x, y, z axes direction in m/s.
{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 {Time 62.01} {Location 4.4976,1.9000,1.8527} {Orientation 0.0000,-0.0003,0.0000} {Velocity -0.0000,0.0000,-0.0000} {LightToggle False} {LightIntensity 0} {Battery 3564}

  A Mission Package state message reports ALL the mission packages' state. Every mission package state includes a Type segment, a Name segment and several

part pose segments that list the parts that construct the mission package and their poses relative to their mount bases. A mission package state message looks like:

MIS {Type *string*} {Name *string*} {Part *string* Location *x,y,z* Orientation *x,y,z*} … {Type *string*} {Name *string*} {Part *string* Location *x,y,z* Orientation *x,y,z*} …
Where:

{Type *string*}	'*string*' is the mission package type. For example it can be "PanTilt" which means the mission package is a pan-tilt style package.
{Name *string*}	'*String*' is the name of the mission package.
{Part *string* Location *x,y,z* Orientation *x,y,z*}	This segment describes a part that constructs the mission package. The '*String*' after Part is the part name, and the '*x,y,z*' after Location is the part's location relative to its mount base in meters. The '*x,y,z*' after Orientation is the part's orientation relative to the mount base in radians.

Example: MIS {Type PanTilt} {Name Cam1} {Part CameraBase Location -0.0000,-0.0000,0.0000 Orientation 0.0000,0.0000,0.0000} {Part CameraPan Location -0.0000,-0.0000,0.0280 Orientation -0.0001,0.0003,0.5693} {Part CameraTilt Location -0.0000,-0.0000,0.0000 Orientation 0.0000,0.0000,0.0000} {Type ARM} {Name Arm1} {Part BarA Location -0.0952,-0.0015,-0.0000 Orientation 0.0000,0.0002,0.0000} {Part BarB Location 0.0952,0.0015,0.0000 Orientation 0.0000,0.0000,0.0000}

- 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 8.

  - 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}

- INU sensor

  SEN {Type INU} {Name *string*} {Orientation *x,y,z*}

  Where:

{Name *string*}	'*string*' is the sensor name.
{Orientation *x,y,z*}	'*x,y,z*' is the orientation relative to the start pose.

  Example: SEN {Type INU} {Name INU} {Orientation 0.0000,0.0302,-0.5051}

- 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. The tag ids are set automatically, however this is optional. This option can be turned on or off by manipulating the bAutoId variable at the USARDeathMatch class. 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.
{ID *int*}	'*int*' is the detected RFID tags' id.
{Location *float, float, float}*	'*float, float, float*' is the x,y,z triplet of the location of the tag in sensor coordinates.

Example: SEN {Type RFID} {Name RFID} {ID 0} {Location 1.23, 2.23, 0.12} {ID 3} {Location -2.23, -1.23, 0.12}

- Victim sensor

Victim tags are simulated by implementing the class USARBot.VictRFIDTag class in the UnReal Editor when editing a map. This class has the strings victim_name, and victim_state in addition to the integer id and the boolean bSingleshot. By default the victim tags are multi-shot tags. They are deployed the same way as the RFID tags, however, the victim names and states have to be set manually. The ids do not need to be specified unless they are used to simulate false positive tags. In this case the ids have to be set to -1. If the robot is closer than the FPMinRange and if the victims are in the robot's field of view then the server sends "None" as the ID and state. If the robot gets closer than VictMinRange, the server sends the same message with the ID filled in. If the robot gets closer than VictStatusRange, the server sends the same message with the state filled in.

SEN {Type VictRFID} {Name *string*} {ID *string*}{State *string*} {Location *float, float, float* } …

Where:

{Name *string*}	'*string*' is the sensor name.
{ID *string*}	'*string'* is the detected victims ID or FalsePositive if within an identification range or None if within detection range but not within id range.
{State *string*}	'*string*' is the condition of the victim, e.g. 'unconscious' , 'responsive', …
{Location *float, float, float}*	'*float, float, float*' is the x,y,z triplet of the location of the tag in sensor coordinates.

Example: SEN {Type VictRFID} {Name VictRFID} {ID None} {State None} {Location 1.2, -2.20, 0.21} {ID None} {State None} {Location 2.1, 2.20, -0.21}

If the robot now moves closer…
SEN {Type VictRFID} {Name VictRFID} {ID Jim} {State None} {Location 1.2, -2.20, 0.21} {ID Sarah} {State None} {Location 2.1, 2.20, -0.21}
If the robot now moves even closer to 'Jim'…
SEN {Type VictRFID} {Name VictRFID} {ID Jim} {State Responsive} {Location 1.2, -2.20, 0.21} {ID Sarah} {State None} {Location 2.1, 2.20, -0.21}

False positive tags are simulated by implementing the class USARBot.VictRFIDTag. In order to distinguish them from victim tags, their id have to be set to -1, otherwise they will act as victim tags. The messages that are sent are described above.

- 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 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 state message includes the geometry information of all the mission packages that are of the same type. The 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 *string*} {Name *string* Location *x,y,z* Orientation *x,y,z*} {Part *string* Location *x,y,z* Orientation *x,y,z*} {Part *string* Location *x,y,z* Orientation *x,y,z*} … {Name *string* Location *x,y,z* Orientation *x,y,z*} {Part *string* Location *x,y,z* Orientation *x,y,z*} …

Where:

{Type *string*}	'*string*' is the mission package's type.
{Name *string* Location *x,y,z* Orientation *x,y,z*}	'*string*' is the mission package's name. The '*x,y,z*' after 'Location' and 'Orientation' are the item's relative location and orientation to the mounting base. The mounting configuration information is located in the CONF message.
{Part *string* Location *x,y,z* Orientation *x,y,z*}	This segment provides information about the part that constructs the mission package and it's location, orientation relative to the mounting base. Again, the mounting information can be found in the CONF message.

Example: GEO {Type PanTilt} {Name Cam1 Location 0.1200,0.0000,-0.0826 Orientation 0.0000,-0.0002,0.0000} {Part CameraPan Location -0.0000,0.0000,0.0279 Orientation 0.0000,0.0000,0.0000} {Part CameraTilt Location -0.0000,0.0000,-0.0000 Orientation 0.0000,0.0000,0.0000}

- Configuration Information

  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 8 about how to configure the sensor.

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

For a mission package, the configuration information is about the connection relationship among the element parts. All the mission packages in the specified type will appear in the message. The message looks like:

CONF {Type *string*} {ScanInterval *float*} {Name *string*} {Part *string* Parent *string* Mount *string* Location *x,y,z* Orientation *x,y,z* Mount *string* Location *x,y,z* Orientation *x,y,z* …} … {Name *string*} {Part *string* Parent *string* Mount *string* Location *x,y,z* Orientation *x,y,z*}

Where:

{Type *string*}	'*string*' is the mission package's type.
{ScanInterval *float*}	'*float*' is the time interval in seconds that the MIS message will be sent.
{Name *string*}	'*string*' is a mission package's name that has the specified type.
{Part *string* Parent *string* Mount *string* Location *x,y,z* Orientation *x,y,z* Mount *string* Location *x,y,z* Orientation *x,y,z* …}	This segment shows the element part's configuration information. It includes the element's name and its parent's name which is also a part. Then all the mount bases on the element are listed one by one. For every mounting base, the information includes the mounting base's name and its location and orientation relative to the element. If the element has no mounting base, the segment will only include the part and parent information. The mounting bases' name are generated by M_*A*_*B*, where *A* is the element's name and *B* is the part's name that will mount to A through the mounting base.

Example: CONF {Type PanTilt} {ScanInterval 0.20} {Name Cam1} {Part CameraBase Parent None} {Part CameraPan Parent None Mount M_CameraPan_CameraTilt Location 0.0000,0.0000,-0.0719 Orientation 0.0000,-0.0002,0.0000} {Part CameraTilt Parent CameraPan}

- Response Message

  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's 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 8 and 9.

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

7.4 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.USARCar, USARBot.USARBc, USARBot.P2AT, USARBot.P2DX, USARBot.Rover, 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 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*} {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 {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

- Control the camera:

CAMERA {Rotation *x,y,z*} {Order *int*} {Zoom *float*}
Where:

{Rotation *x,y,z*} '*x,y,z*' is the rotation angle of the camera in radians. It could be a relative or absolute value. This is set in the robot configuration file.

{Order *int*} It's the same as the order parameter of the DRIVE command. The possible values are 0 and 1. Without specifying the parameter, the system treats it as '0'.

{Zoom *float*} '*float*' is the camera's field of view in radians. Smaller fields of view gives a the zoom-in effect. You can also use SET command to adjust a camera's FOV.

Example: CAMERA {Rotation 0,0.175,0} will tilt the camera 10 additional degrees if the robot uses relative values. If the robot uses absolute values, it will tilt the camera to 10 degrees.
CAMERA {Zoom 0.35} will zoom in the camera until its FOV is 20 degrees.

- Control the 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 *p1,p2,…*}

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, may have different opcodes. For example, it can be "RESET", "ZOOM" etc.

{Params *p1,p2,…*} '*p1,p2,…*' are the parameters associated with the operation command.

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

- Control the 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**'s pose and let USARSim control every element's joint for us. If the camera is mounted on a PanTilt mission package, we can use this command to control the camera's pose. And in this case, this command is the same as the CAMERA command without the ZOOM parameter[2]. Using mission package control commands, we can have multiple cameras and control them separately. The command's format is:

MISPGK {Name *string*} {Rotation *x,y,z*} {Translation *x,y,z*} {Order *int*}

Where:

{Name *string*}	'*string*' is the mission package's name.
{Rotation *x,y,z*}	'*x,y,z*' is the rotation parameter that controls the terminal's rotation position, speed or acceleration. The Order parameter decides whether it's position or speed or acceleration.
{Translation *x,y,z*}	Same as Rotation except here it's a translation parameter.
{Order *int*}	'*int*' indicate the control mode. '0' means position control, '1' means speed control, and '2' means acceleration control.

Example: MISPKG {Name Cam1} {Rotation 0,0,0.57} {Order 0}
MISPKG {Name Arm1} {Translation 0.1,0,-0.1} {Order 0}

TIPS: With PanTilt mission packages, we can have multiple cameras and control them separately.

- Query the 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}

[2] The CAMERA command will be discarded in the future and may be replaced by a MISPKG and SET command.

- 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}

- 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 '{Camera *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.

8 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.

[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.

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 12.2.

8.1 State Sensor

8.1.1 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. All of the data in the state sensor data package should be treated as ground truth data.

8.1.2 How to configure it

None. We needn't configure it.

8.2 Range Sensor

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

8.2.2 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,Out
Val=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,Out
Val=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.

8.3 Range Scanner Sensor

8.3.1 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.

8.3.2 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=1000.000000,
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.
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.

8.4 Odometry Sensor

8.4.1 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.

8.4.2 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.

8.5 INU Sensor

8.5.1 How the sensor works

This sensor implements an Inertial Navigation Unit. USARSim simulates it by calculating the current rotation relative to the robot's start rotation and then adds random noise. We can use the SET command to reset the sensor and force it use the current rotation as the reference for the orientation estimation. The Opcode, Params and returned response Status are listed below:

Table 5 INU 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.

8.5.2 How to configure it

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

```
[USARBot.INUSensor]
HiddenSensor=true
bWithTimeStamp=False
Weight=0.4
ScanInterval=0.0
Noise=0.05
```

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 estimates of orientation. If we set it to 0, the sensor will use the robot's message sending time interval.
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.

8.6 Encoder Sensor

8.6.1 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 6 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.

8.6.2 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.

8.7 Touch Sensor

8.7.1 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.

8.7.2 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.

8.8 RFID Sensor

8.8.1 How the sensor works

RFID sensor simulates a RFID reader. It iterates all the RFID tags registered in the virtual world. When a tag is within the sensor's detection range, it will report the tag's id.

We can either statically add a RFID tag in the virtual world from UnrealEd or dynamically drop it to the world from a RFID releaser. When a RFID tag is spawned in the world, it will automatically register with USARSim. When it is destroyed in the world, it will unregister from USARSim.

8.8.2 How to configure it

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

```
[USARBot.RFIDSensor]
HiddenSensor=true
bWithTimeStamp=False
Weight=0.4
MaxRange= 0.28
```

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 detection range in meters.

8.9 Victim RFID and False Positive Sensor

8.9.1 How the sensor works

The Victim RFID and False Positive Sensor simulates a victim location sensor. When pointed at a victim within the maximum range of the sensor, a report is generated with an ID of "None". In order to differentiate between a false alarm and a real victim, the robot must move closer to the tag than the minimum range. At that point, the message will include a valid ID of either the victim's name or "FalsePositive" for a false alarm.

8.9.2 How to configure it

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

```
[USARBot.RFIDVictSensor]
HiddenSensor=true
bWithTimeStamp=False
Weight=0.4
FPMinRange= 1.33
VictMinRange=0.5
VictStatusRange=.25
RFIDFOV=0.52
```

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.
FPMinRange	The maximum detection range in meters.
VictMinRange	The maximum identification range in meters (range at which can tell if false positive).
VictStatusRange	The maximum state determination range in meters (range at which you can asses the health of a victim).
RFIDFOV	The field of view of the sensor in radians.

8.10 Sound sensor

8.10.1 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 12.1.3 about how to associate a sound source to a victim.

8.10.2 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.

8.11 Human-motion sensor

8.11.1 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.

8.11.2 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=1000.000000,
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.
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.

8.12 Robot Camera

8.12.1 How the sensor works

The Camera is the most special sensor in USARSim. The scenes viewed from the camera are captured by attaching the viewpoint to the camera in the Unreal engine. USARSim surrpots multiple cameras. It treats the first camera on the robot in the .ini file

as the main camera. We can use the CAMERA command[4] to control the main camera's pose and its FOV. For other cameras, we use SET command directly adjust its FOV and MISPKG command to control the camera's pan-tilt frame. The SET command's Opcode, Params and returned response Status are listed below. Every camera has a default, maximum and minimum FOV of the viewpoint. If the FOV in the SET command is out of the camera's FOV range, it will adjust it to the minimum or maximum FOV range.

Table 7 Robot camera control command

Opcode	**Params**	**Status**
Zoom	float number: the FOV of the camera	float number: the camera's current FOV. "–1" means failed to change FOV. This may be caused by an invalid command.

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 12.5.1.

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 12.5.2.
   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

[4] In the future, CAMERA will be discarded. For all the cameras, we must use MISPKG and SET command.

server are listed in section 4.2.5.2. How to communicate with the server and use the pictures are explained in section 12.5.3.

8.12.2 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.

9 Effecters

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.

9.1 RFID Releaser

9.1.1 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 to make sure that all the IDs are unique. The releaser generates the ID by using the virtual world's time in 1000 ms pluses. In other words, the IDs from 0 to 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 8 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.

9.1.2 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.

9.2 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.

10 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 the DRIVE command. For the drive command, the wheels' spin speed can be set as an absolute value in radians/second or as a normalized speed in which 100 means maximum translate speed or rotation speed. For details about the DRIVE command, please go back to section 7.4.

10.1 P2AT

10.1.1 Introduction

The 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: 50 cm x 49 cm x 26 cm
- Wheel diameter: 22 cm
- Weight: 14 kg
- Payload: 40 kg

a) Real P2AT

b) Simulated P2AT

Figure 12 P2AT robot

By default, in our simulation it's equipped with

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

The specification is:

Dimension: Length x Width x Height = 50 cm x 49 cm x 26 cm
Wheel: Diameter x Width = 22 cm x 7.5 cm
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 },
```

Maximum translate speed: 700 mm/s
Maximum rotating speed: 140 deg/s

10.1.2 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 12.4.2.

```
[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.

10.2 P2DX

10.2.1 Introduction

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

a) Real P2DX b) Simulated P2DX

Figure 13 P2DX robot

In summary, P2DX has:

- Two wheels
- Differential steering
- Size: 44 cm x 38 cm x 22 cm
- Wheel diameter: 16.5 cm
- Weight: 9 kg
- Payload: 20 kg

In our simulation, it's equipped with

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

The specification is:

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
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 }
```

Maximum translate speed: 1800 mm/s
Maximum rotating speed: 300 deg/s

10.2.2 Configure it

It’s the same as P2AT.

10.3 ATRVJr

10.3.1 Introduction

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

In summary, ATRV-Jr has:

- Four wheels
- Differential steering
- Size: 55 cm x 77.5 cm
- Weight: 50 kg
- Payload: 25 kg
- Sonars: 17 (5 front, 10 side, 2 rear)

In our simulation, it’s equipped with

- PTZ camera
- 17 sonars
- Sick Laser Scanner LMS200

a) Real ATRVJr

b) Simulated ATRVJr

Figure 14 ATRVJr robot

The specification is:

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)
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 }

Maximum translate speed: 1000 mm/s
Maximum rotating Speed: 120 deg/s

10.3.2 Configure it

It’s the same as P2AT.

10.4 PER (Rover)

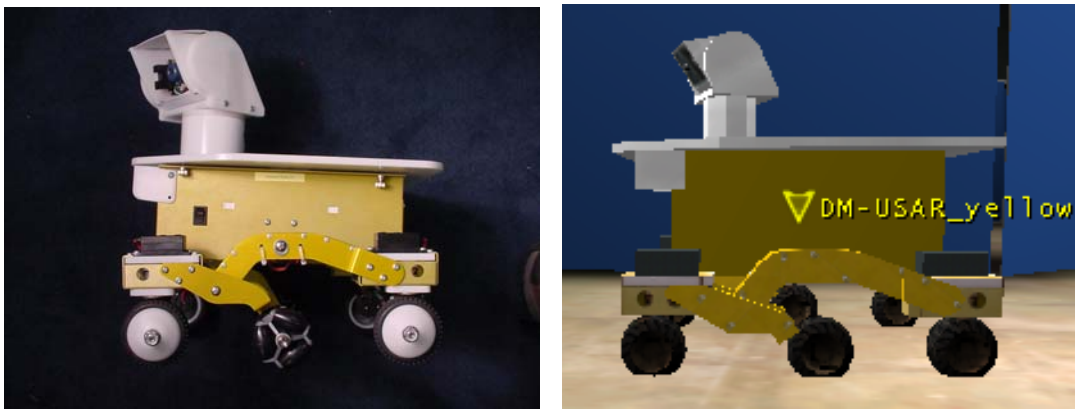
10.4.1 Introduction

The PER is the Personal Exploration Rover built by CMU for education and demonstration purpose. The robot uses a rocker-bogie suspension system to adapt to terrain. It has a pan-tilt camera mounted on it. For details about PER please visits 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 a 360 degree panorama

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



a) Real PER b) Simulated PER

Figure 15 PER robot

10.4.2 Configure it

It's the same as P2AT.

10.5 Corky

10.5.1 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. . It's designed for this specified robot. To archive speed control, PID controllers are built for both wheels of Corky. Please note the model is obsolete.

In USARSim, we use the classname USARBot.USARBc to represent Corky.

10.5.2 Configure it

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

a) Real corky

b) Simulated corky

Figure 16 Corcy robot

```
[USARBot.USARBc]
msgTimer=0.200000
bSpeedControl=True
bAbsoluteCamera=False
Sensors=(SensorClass=class'USARBot.RangeSensor',SenName="Front",Position=(
X=-80,Y=0,Z=50),Direction=(Pitch=0,Yaw=32768,Roll=0))
…
Sensors=(SensorClass=class'USARBot.RangeSensor',SenName="Right",Position=(
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	The time interval between sending two messages.
bSpeedControl	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	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	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 direction the sensor is facing relative to the robot.

Kp	The proportional parameter of the PID control. Both wheels 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.

10.6 Four-wheeled Car

10.6.1 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. The model is obsolete.

In USARSim, we use the classname USARBot.USARCar to represent the car.

Figure 17 Simulated Four-wheeled Car

10.6.2 Configure it

It's the same as Corky.

10.7 Papagoose

10.7.1 Introduction

The Papagoose (as shown in Figure 18) is a rescue robot that was built at the International University Bremen (http://robotics.iu-bremen.de). It is a six wheel differential drive and is equipped with the following sensors:

- Pan tilt camera
- Odometry
- INU
- 6 sonar sensors (3 on the front and 3 on the back)
- a range scanner on the front
- RFID sensor
- RFID victim sensor

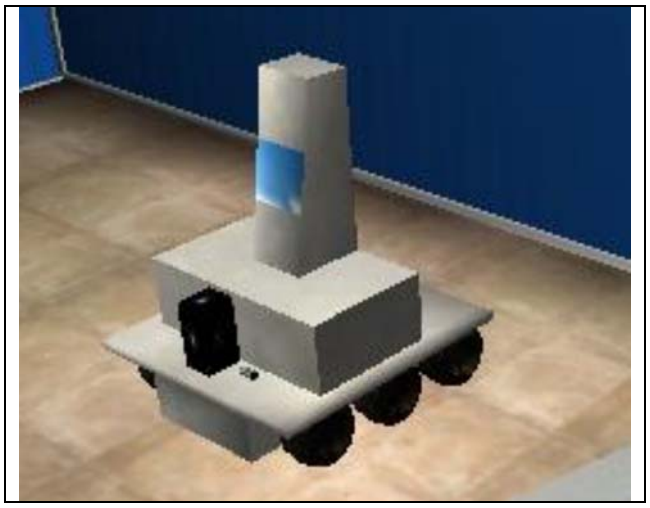


Figure 18: Papagoose robot

10.7.2 Configure it

Please see the section on the P2AT.

11 Controller

11.1 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 4.2.2. 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 19 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.

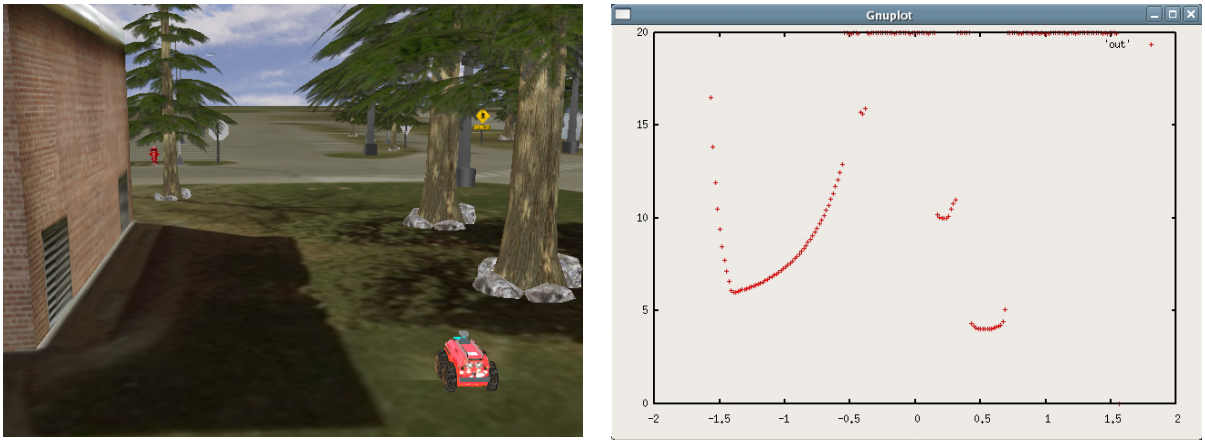


Figure 19: Image from the unreal Client and resulting graphical display of Sick LMS readings provided by nmlPrint and gnuplot. The building and trees are clearly visible in the plot.

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.

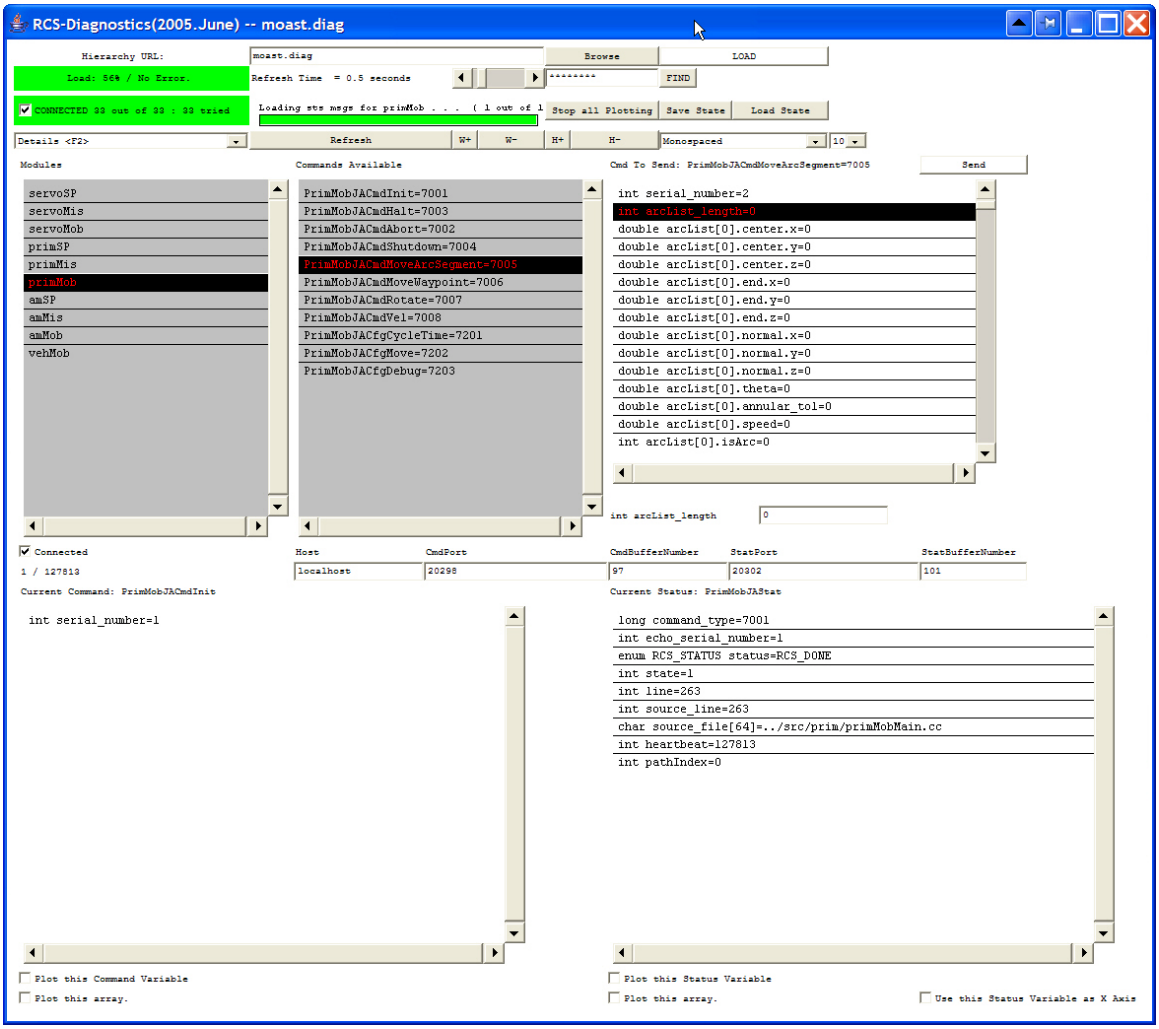


Figure 20: MOAST Diag display

Figure 20 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.

11.2 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.

11.2.1 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 7.1).
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.

11.2.2 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.

11.2.3 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

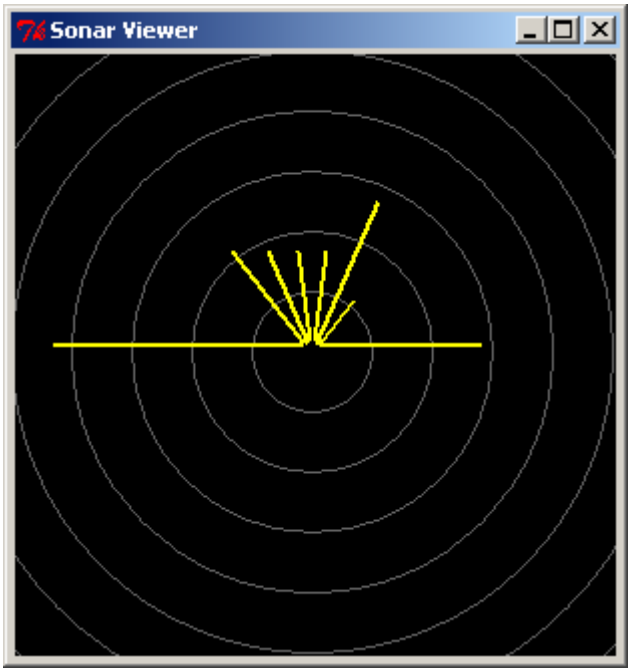


Figure 21 Sonar visualization

- Laser

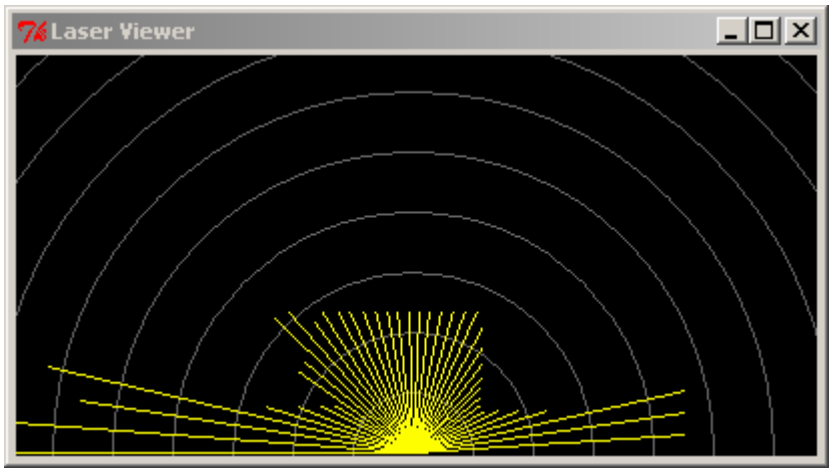


Figure 22 Laser visualization

- PTZ Camera

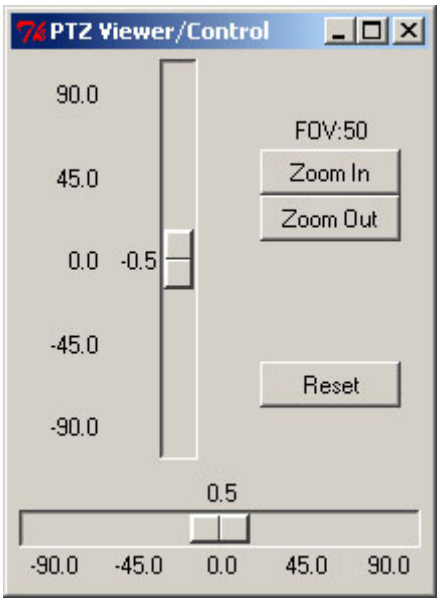


Figure 23 PTZ Camera viewer and controller

11.2.4 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.

11.3 Player

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 Player User Manual: http://playerstage.sourceforge.net/doc/doc.html.

11.3.1 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 11.3.2.1.

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 11.3.2.2 to section 11.3.2.6.

The following is an example configuration:

```
simulation:0
(
 driver "us_bot"
 host "127.0.0.1"
 port 3000
 pos "4.5,1.9,1.8"
 bot "USARBot.ATRVJr"
)

position:0
(
 driver "us_position"
 simulation_index 0
)

laser:0
(
 driver "us_laser"
 simulation_index 0
)
```

The USARSim robot is defined in ‘simulation:0’. It’s an ATRV-Jr robot. The ‘position:0’ and ‘laser:0’ are USARSim devices. The ‘simulation_index 0’ specifies that the devices are located on ‘simulation:0’ that is an ATRV-Jr robot.

11.3.2 Device Drivers

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

11.3.2.1 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	string	0,0,0	The initial spawn position
bot	string	P2AT	The robot type

11.3.2.2 us_position

Synopsis:

The us_position driver is used to control the robot’s movement.

Interfaces:

Supported interfaces:

- position

Required devices:

- None.

Configuration file options:

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

			located.

11.3.2.3 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:

- None.

Configuration file options:

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

11.3.2.4 us_sonar

Synopsis:

The us_sonar driver is used to access **all** the robot's sonar sensors.

Interfaces:

Supported interfaces:

- sonar

Required devices:

- None.

Configuration file options:

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

11.3.2.5 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:

- None.

Configuration file options:

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

11.3.2.6 us_ptz

Synopsis:

The us_ptz driver is used to control the robot's ptz camera. The camera should use absolute pose control.

Interfaces:

Supported interfaces:

- ptz

Required devices:

- None.

Configuration file options:

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

NOTE: The camera should use absolute pose control.

12 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=BotAPI
ServerPackages=USARBot

[Editor.EditorEngine]
EditPackages=BotAPI
```

EditPackages=USARBot

[UnrealEd.UnrealEdEngine]
EditPackages=USARBot

NOTE: You need to modify ut2004.ini before you build your own models.

12.1 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.

12.1.1 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 24 Some NIST facilities

12.1.1.1 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).

12.1.1.2 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.

12.1.2 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

12.1.3 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 25.

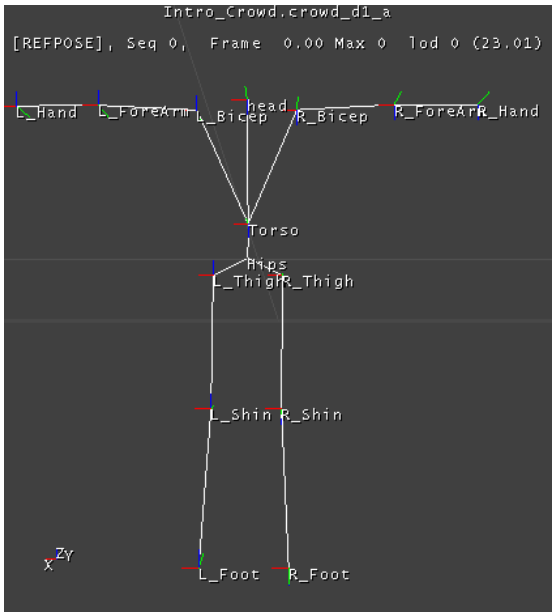


Figure 25 Skeletal bones name

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.

12.2 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)

12.2.1 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 showed below.

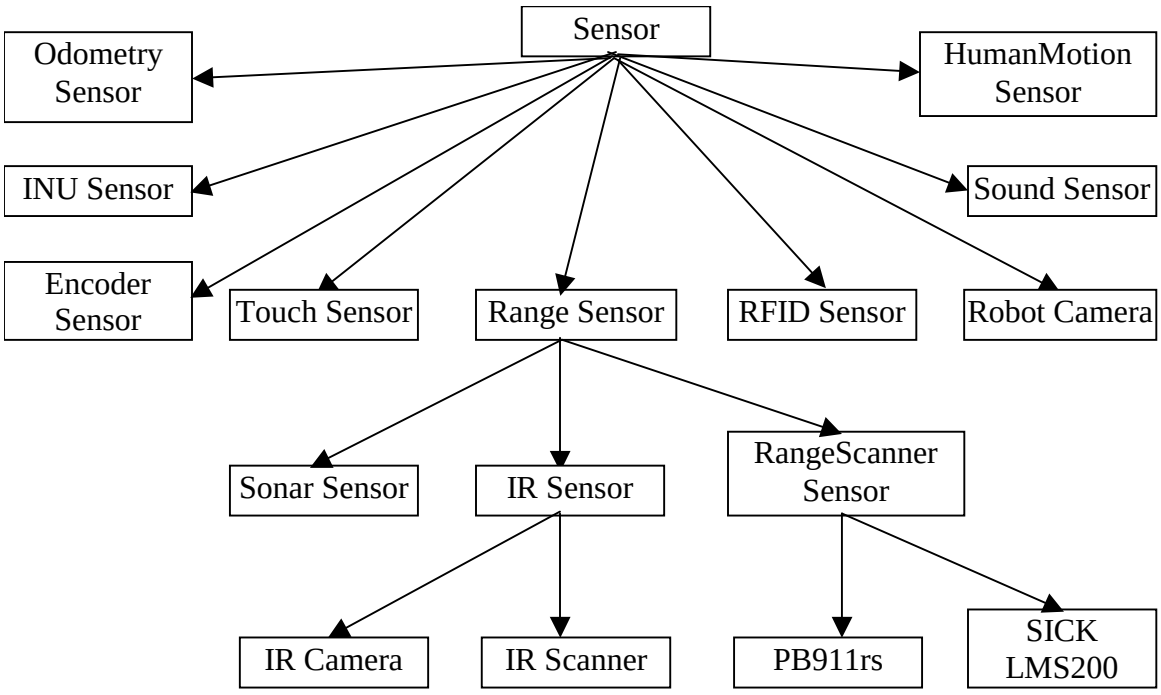


Figure 26 Sensor Hierarchy Chart

12.2.2 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

12.2.3 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.

12.3 Build your effecter

12.3.1 Overview

In USARSim, all effecters are inherited from the Effecter class. We use the hierarchical architecture to build the effecters. Right now, we only have one effecter, RFIDRelease. As we mentioned before, the effecter is very similar to a sensor and we can treat an effecter as a dumb sensor.

12.3.2 Effecter Class

The effecter class extends from the Item class. The new methods added in the Effecter class are:

Methods:

function String GetGeoHead() // the interface that sends the effecter's geometric information HEAD to the robot

function String GetGeoData() // the interface that sends the effecter's geometric data to the robot

function String GetConfHead() // the interface that sends the effecter's configuration information HEAD to the robot

function String GetConfData() // the interface that sends the effecter's configuration data to the robot

12.3.3 Writing your own effecter

Same as section 12.2.3

12.4 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.

12.4.1 Step1: Build geometric model

Essentially, this step is the same as building your own arena. Please refer to section 12.1.1 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.

12.4.2 Step2: Construct the robot

12.4.2.1 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.

12.4.2.2 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.

12.4.2.3 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.

12.4.2.4 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 27) that the part can spin around. Another is the steering and suspension axis (Steering Axis in Figure 27) that the part can steer around and travel along. A hinge joint connects two parts by one axis.

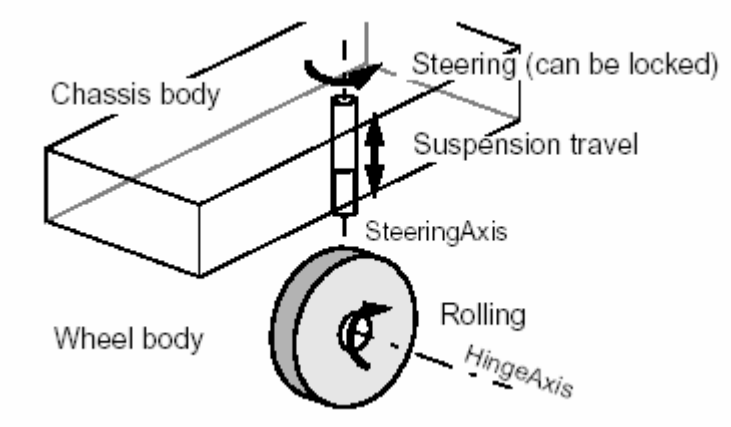


Figure 27 Car wheel joint

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;
  //Mission Package
  var() name                    PackageName;
  var() name                    PackageType;
};
```

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
PackageName	If this part-joint pair belongs to a mission package, here we give the mission package's name.
PackageType	If this part-joint pair belongs to a mission package, here we give the mission package's type.

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.

12.4.2.5 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.

12.4.3 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 and Rover as examples 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.

12.5 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 8.12, 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:

12.5.1 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

12.5.2 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.

12.5.3 Using The Image Server

The Image Server simulates a web camera. How to run it is described in section 4.2.5.2. 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.

13 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.

14 Contributors

The primary author:

- Jijun Wang at University of Pittsburgh, USA

Management and Coordination:

- Stephen Balakirsky at National Institute of Standards, USA
- Michael Lewis at University of Pittsburgh, USA
- Stefano Carpin at International University Bremen, Germany

USARSim drivers for Player 1.6:

- The first version from Erik Winter at Uppsala University, Sweden
- The updated version from Stefan Markov & Ravi Rathnam at International University Bremen, Germany

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
- Stereo P2AT from Giuliano Polverari at University of Rome "La Sapienza", Italy

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

15 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.