Robots, Games, and Research: Success stories in USARSim

Since its genesis in 2003, USARSim has evolved into a full-blown robot simulator whose components have been downloaded more than 50,000 times. The spectrum of imagined applications has grown far beyond the initially envisioned search and rescue scenarios, and now encompasses a huge variety of robots, sensors and actuators. Examples of successful and effortless migrations of code from simulation to real robots abound; a clear indication of its value as a tool to ease the development and debugging of robot control software targeted for running on real robots. An active community of developers has contributed back a variety of additions to the open source project, and a fruitful exchange of ideas and tools has been established. This workshop brings together researchers and educators using this software in order to promote and exchange of ideas, results, and software components. The twelve papers presented illustrate a significant subset of the most significant developments, including educational games, Robocup, human-robot interaction, integration with third-party commercial software, and federally funded large research projects.

This timely event coincides with the beta release of the new version of the simulator that will provide enhanced physics and graphics realism and that will offer many renewed opportunities to current and new users. USARSim is freely available online at http://sourceforge.net/projects/usarsim/.

Stephen Balakirsky
Stefano Carpin
Mike Lewis

St. Louis, MO
October 15, 2009

Table of contents

A Humanoid-Robotic Replica in USARSim for HRI Experiments

Kyle Carter and Matthias Scheutz and Paul Schermerhorn
Human-Robot Interaction Laboratory
Cognitive Science Program
Indiana University
Bloomington, IN 47406, USA
{kylcarte,mscheutz,pschermeg}@indiana.edu

Abstract— **An important set of open questions in human-robot interaction research, and to some extent cognitive science, is centered around the difference in interactions humans have with real versus simulated robots or agents. The goal of this research is to understand the effects of the agent's embodiment on human perception and cognition.**

In this paper, we present our work on providing computational tools to facilitate research in embodied situated cognition and human-robot interaction. Specifically, we introduce a simulation model of our humanoid robot CRAMER which we developed in the UNREAL game engine using the USARSim control interface. We provide details on its development and the implementation of the control interface that allows it to work seamlessly with our existing robot control architectures. We also discus potential applications of the simulation model as well as future plan to extend it.

I. INTRODUCTION

Simulated environments like USARSim [1] have become important tools for the development, testing and debugging of robot control architectures in a variety of areas, including single and multi-robot setups with and without human-robot interaction (HRI). In addition to rapid prototyping of control software, however, simulated robots can also serve an important role in psychological research: they can be used for the study of important psychological phenomena related to *embodiment* and *situatedness* of agents, which are of critical importance for human-computer and human-robot interaction. Specifically, sufficiently accurate simulations of real robots will allow us to study any possible differences in how humans interact with real versus virtual agents. These differences, then, will have significant implications for the design, testing, and deployment of robots and robotic architectures. For example, we have demonstrated that the expression of affect in a robot's voice can motivate people to perform better at a joint task when the robot is physically co-present in the same environment as opposed to just shown on a video screen [6], [7], [8]. Similarly, a warning message from a virtual character might be *less believable* than that from a physical robot [3]. Hence, one important implication for the design of robotic architectures in simulation is that HRI mechanisms that work well with simulated robots might not work well or work at all with physical robots and vice versa.

In this paper, we describe our work developing a simulated replica of our physical robot that can be used for HRI studies, specifically to explore the effects of physical embodiment. We will describe the details of the simulated robot, how it was developed, how it can be controlled, and how it can be used for future psychological experiments.

II. BACKGROUND

Multiple simulation packages are available that can be used for the development of simulated robotic replicas (including USARSim, Gazebo, ODE, etc.). Some of the packages already come with robot models and APIs for software control architectures (e.g., USARSim or Gazebo), while others provide only a core physics engine within which both simulated robot models as well as software interfaces will have to be developed. Our selection of USARSim was based on the fact that there is a fairly substantial user community with increasing support for robot models and environments.

USARSim is a collection of modifications made to *UNREAL Tournament* (a commercial "first-person shooter" game) for the purpose of allowing a robotics control architecture to interface the game engine. UNREAL Tournament is a physics-based simulation, which supports rigid-body physics and interactions. The USARSim modifications remove all game-related elements of UNREAL, leaving only the physics engine and some client software. USARSim also includes GameBots, freeware from a 3rd party developer, which provides mechanisms for communication over a TCP socket. This means of communication forms the basis for the creation and control of agents within the simulated environment at runtime. Because GameBots uses sockets for communication, developers can construct their own client software to connect to USARSim. It is through this client that one can place agents in (or remove agents from) the simulation environment. And the client also allows robotic control architectures to send commands to the simulated robots (e.g., to control a robot's wheels or actuators).

While we are using a variety of robots in our HRI studies, we are particularly interested in human-robot interactions with human-like robots. For this purpose, we employ our robot CRAMER (the "Cognitive, Reflective, Affective, Mobile, Expressive Robot"), which consists of a humanoid upper torso (manufactured by the now defunct company RoboMotio) mounted on a mobile Pioneer P3DX platform. CRAMER has two firewire cameras mounted in its eyes

and a series of eight microphones mounted in its torso. Moreover, it has movable eyes, eye brows and lips to produce facial expressions. The challenge for the simulated robot in USARSim was thus to replicate all of CRAMER's effector capabilities as closely as possible, including their timing and their degrees of freedom. Our plan was for the replica to be displayed on life-size 63in monitor so that the simulated and real CRAMER, placed side-by-side, would exhibit (close to) identical motions if controlled by the same architecture.

The development effort consisted of the integration of three primary tasks. First, we created a simulation model for CRAMER with the help of a number of 3rd party programs and tools. Second, we produced the classfile structure and configuration that UNREAL Tournament uses to recognize CRAMER as a placeable agent. Finally, we wrote a client for USARSim building upon the basic functionality already provided by USARSim and GameBots, together with additional control software for our ADE control environment, to allow us to connect our robotic architecture to the simulated CRAMER in exactly the same way it is connected to the physical robot. In the following, we will describe all three parts in more detail.

III. SIMULATION MODELS OF HUMANOIDS IN USARSIM

A fully functional simulacrum of a robot requires both a model of the physical shape and behavior of the robot as well as specifications of how some of the movable parts can be controlled. Creating the physical model in USARSim consists of "modeling the robot's parts", "adding texture to the surfaces" to make it look like the original, "organizing various configuration files" that UNREAL requires for the parts to function as a whole, and "configuring the robot's joints to function properly". Once the simulated robot is assembled within UNREAL, software can be developed to control it. Our lab's USARSim client software currently consists of three main components: the code that initializes communication with USARSim, the parser which interprets all of the incoming data, and a set of methods that implement the translation of the platform-independent action commands of our robotic control architecture into platform-specific commands of USARSim.

A. Model Creation

The possible methods for creating the model of the simulated robot are varied, and the particular combination of methods presented here is only one of many. As part of our commitment to open-source operating systems and software, our lab chose to work as much as possible with free and open-source tools available for Linux, which turned out to be more complicated and time-consuming than what model development would have been otherwise using proprietary Windows-based software. To construct the three-dimensional models of CRAMER's parts, we used the free modelling program *Blender*. The texturing was originally accomplished using the raster image editor *the GIMP*, but later we moved to the scalable vector graphics image editor *Inkscape*. Using

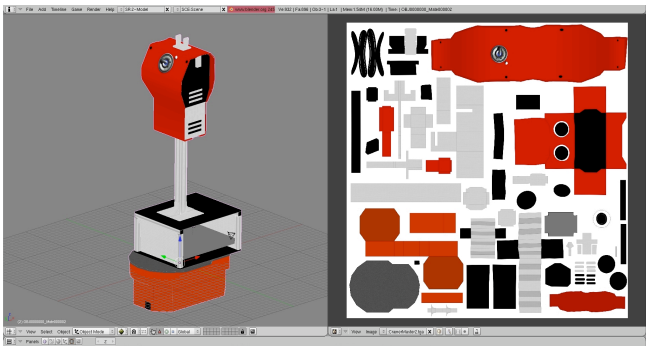


Fig. 1. Split shot of the model and texture in Blender, showing how the texture is mapped to the model.

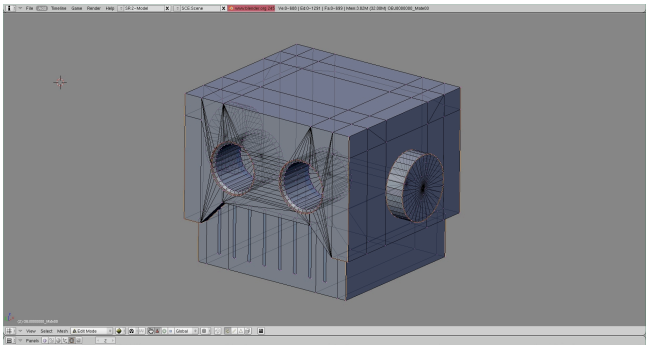


Fig. 2. Wireframe view of the CRAMER head model.

these programs (after a significant learning phase), a smooth workflow developed that proved quite efficient at producing robot models.

The modelling software, Blender, handled both the forming of the models, and their preparation for texturing, called "wrapping" (see Fig. 1 and Fig. 2). The wrapping process involves designating seams on the models in order to provide a smooth, relatively distortion-free means of wrapping a two-dimensional texture around a three-dimensional object. There do exist other open source modelling tools that perhaps could be used to create the 3D content for USARSim, but none seemed to be as mature and effective as Blender, and so their use was not considered further. Once the model

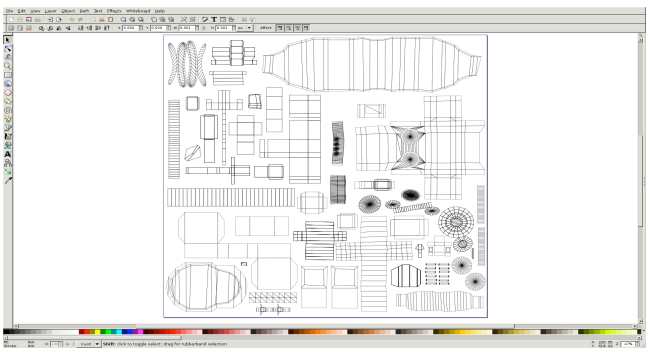


Fig. 3. The blank texture before it is filled in using Inkscape.

was prepared, the wrapping template was exported to an image editor, where it was colored and smoothened (see Fig. 3). One significant benefit that using Blender brought to this process was its ability to export the unwrapped texture directly to a file with which vector graphics software could work. Similarly to our work with Blender, there were other programs that could have been used to fill in the textures, such as Xara Xtreme LX, which is released under the Gnu Public License. Inkscape initially seemed to be more developed and robust than the alternatives, and so further exploration was deemed unnecessary. Both the model and the texture image were then exported into formats that the UNREAL Editor would accept (all content used by USARSim must be of in one of a few particular formats, according to type). In order for UNREAL to be able to use the robot models, it had to first be packaged by the UNREAL Editor. While we were able to use open-source software for physical modelling and texturing, only the proprietary UNREAL Editor can package up the collection of model, texture, and configuration files and class files, the latter of which define the robot as an entity and connect the pieces into controllable groups of joints (for details on this process, see the documentation in the USARSim manual). Configuration files give joints their limits, including range of motion, maximum speed, maximum torque, along with other parameters. With everything in place, the robot can now be initiated and manually controlled through GameBots through a simple telnet connection. For autonomous robot behavior, however, a robot control architecture is needed.

B. USARSim Client Software

In order to encapsulate USARSim's simulation capabilities for the purpose of using it with our robotic software environment ADE, we developed our own client software to provide functionality from a pre-established API that other robot interface components implement (when they interface physical robots). This functionality was achieved by wrapping the low-level joint command provided by USARSim into JAVA-based methods, such as `moveArm` or `moveHead`, that accept a number of arguments and forward the appropriate command together with the arguments to USARSim, which then performs the appropriate action.

As with all physical robots, however, the communication is two-way and sensory data needs to be received from the robot as well. In USARSim, the sensory feedback is continuously provided in special message packets, which need to be parsed accordingly and translated into a format that the robotic architecture can understand, another function performed by our USARSim client. Finally, the client is able to establish communication with USARSim and initialize the robot in the environment in right location in a specified configuration.

C. Control Implementation

The procedures of control are all built around the USARSim defined protocol. All commands are issued to GameBots over the established socket in raw line-based text format, designating which joint is to be moved as well as the magnitude and order of the movements. This order defines what type of command the message is, and can be a positional command, velocity command, or torque command. The order, therefore, determines the meaning of the magnitude, which is a number whose units are either radians, radians per second, or in a special unit used by UNREAL to measure torque, respective to the order. A significant challenge in implementing the control of the simulated robot is the inherent difference in methods of control between the simulated and real version. UNREAL Tournament accepts one of three orders of control for any joint, allowing the user to specify a joint's angle, velocity, or torque. However, in some physical robots, servo control allows for multiple orders to be issued simultaneously (e.g., specifying that a joint move to a certain angle at a particular velocity). Bridging this gap requires more sophisticated control than is natively provided by USARSim. The general solution to this problem is to have the server issue a velocity command, wait for a bit, and then send a command which sets the velocity to zero when the joint is at the correct angle. Two potential methods of ensuring correct timing for the stop command have been implemented to produce the highest fidelity between the physical and simulated robots' movements. While one method seems to be significantly more reliable than the other, there may be uses for which either is better suited.

The first method — "check to stop" — relies upon UNREAL Tournament's incoming information about the joints. Throughout the lifetime of a simulated robot, USARSim is constantly sending information about all of its joints—their respective positions, velocities, and torques. This method then waits for a return value from the parser for the appropriate joint that indicates that it is in the correct position, and so should be stopped from moving further. The method works reliably at lower velocities (0 to $\approx$ 1.2 rad per sec) and on movements that involve longer movements ($> \approx$ 1 to 1.5 rad). However, as the rotational velocity increases, this method's limitation begins to show. USARSim periodically updates its joint information every $\approx$ 0.2 seconds. When the velocity is too high, or the angle displacement is too small, the number of updates received about the joint's position during the movement is reduced to zero, frequently causing the joint to overshoot its intended angle. Ultimately, this limitation prevents the "check to stop" method from being the preferred method.

The second method relies instead on dead reckoning. Given a joint's current angle, the desired angle, and a rotational velocity, one can simply compute the time necessary between start and stop commands using the formula $dt = v/dx$. During initial testing, this method was thought to be flawed by complications in the physical simulation entailed in UNREAL, but further development saw that these errors were in fact being produced by limitations enforced by the USARSim configuration of the robot's joints. In the configuration, certain parameters are set for each joint, including limits on range of motion, as well as limits on the joint's velocity. Because these limitations are enforced internally in UNREAL, it was not initially clear that they

were what was interfering with the calculations. If the client were to order a movement that was faster than allowed by the configured restrictions, the command would be sent without any indication that it was being executed incorrectly except for the updates from UNREAL, which include data about the joint's velocity. The client software would make the calculation based on its anticipated velocity, which would produce a significant error due to the discrepancy of velocity. This problem was bypassed by increasing the maximum velocity of the joint in question. It was found that if the joint was permitted by USARSim to move at the correct velocity, the accuracy of the calculation was restored, allowing this dead-reckoning method to be used exclusively.

It should be noted that in the process of simulating RoboMotio's Reddy robot, we strove for realism of the mechanisms. The facial expressions were a particularly significant point of interest to us, as the robot is intended for human interaction purposes. On the physical robot, the eyebrows consist of one degree-of-freedom (DOF) each, rotation in the plane parallel to the face. Recreating the eyes, likewise, was a relatively simple matter, as they consist of three DOF: individual left-right pan, and a single up-down tilt, linked to both eyes. A "dummy" object was used to link the tilt DOF to both eyes. This object was a massless object that was positioned inside of the head model, so as to be invisible during normal simulation. The mouth, however, presented a much more difficult problem. On the physical robot, the mouth consists of two "lips". Both lips were lengths of flexible rubber hose, bent into any given shape by two servos, one at each end. These four servos could then produce simple facial expressions, such as smiling, frowning, or a tilde-shaped "confused" look, by turning to particular positions, bending the rubber. The problem presented to the simulation is the difficulty of simulating flexible objects. We were also limited by the topology of the mouth, as each defined joint part in USARSim can have up to one parent part. This directly conflicts with the concept of the rubber hose mouth of the physical robot, as the shape of each lip directly depends upon not one, but two joints. To solve this, we broke each lip up into three visible components: Two end sections and one middle section. The four end sections could be rotated like normal joints, with the positions specified by the method "moveMouth(a, b, c, d)". The two middle sections actually each consist of one normal rotational joint and one prismatic joint, which has a linear up-down motion instead of rotational. The positions of these "hidden" joints could be calculated to provide the appearance of a seamless curve. These mechanisms led to the ability to simulate the robot's facial expressions through exactly the same interface of commands. For example, to frown, the robots both turn their eyebrows to tilt downward in the center of the face, and turn all four corners of the mouth upward, bending it into the appropriate shape. The only difference between physical and simulated expressions is that the physical mouth is made out of rubber, and the simulated mouth is made of extra rigid pieces which depend upon the hidden calculations to be put in the correct positions. The commands which are sent to the individual software components that control the physical and the simulated robot are exactly the same.

D. Calibration

Once the simulation was set up, the remaining task required to complete the control functions was to adjust the parameters in the simulated robot to match those on the physical robot, specifically to map the numbers used for velocity commands on the physical robot (using a PWM signal) to the numbers that USARSim uses (radians per second). The physical servos' necessary use of torque precludes a linear or simple calculation. Instead, measurements were taken on each joint, at varying speeds, to match the speeds of the physical and simulated robots, through visual similarity. These points, then, provided the data for a polynomial fitting function that can provide a smooth translation between the PWM signal and the corresponding radians per second. One problem with this strategy, of course, is that physical servos will degrade as they are used over time, and so the fitted function will become obsolete at some point in the future, and the function fitting will have to be repeated.[1]

E. Parsing Sensory and Joint Feedback

We implemented a simple parser to take the sensory output from UNREAL and translate it into a format that the robotic architecture can use. The parser runs in a separate thread from the rest of the client's operations, and primarily consists of a loop that decodes the incoming messages and attempts to ascertain which type of message it has received by checking for distinguishing tokens. For instance, a message that carries information from one of the robot's sensors will start with the token "SEN", while a message with information about the positional data of the robot's joints will start with "MISSTA". The information gleaned from these messages can be used for sensory processing, such as obstacle detection, or for motion control, as discussed in the previous section.

IV. THE USARSIM DIARC/ADE INTEGRATION

The USARSim model described above has been integrated into the *Agent Development Environment* (ADE), an infrastructure toolkit for constructing complex robotic architectures developed in our lab [5]. ADE allows developers to create modular components called *ADE servers* that can subsequently execute on any host with appropriate hardware resources. An *ADE registry* maintains information about all servers currently running in the infrastructure; whenever a new server starts, it checks in with the registry and provides information about its resource needs and the functionality it provides to the system. The registry is then able to provide a reference to that server when another server requests a server with that functionality. Some examples of other ADE servers available to architecture developers are:

- Goal Manager/Action Sequencer
- Speech Recognition

[1] Note that it is unclear how to best address this problem without a thorough model of motor degradation which is unlikely to be available to owners of physical robots.

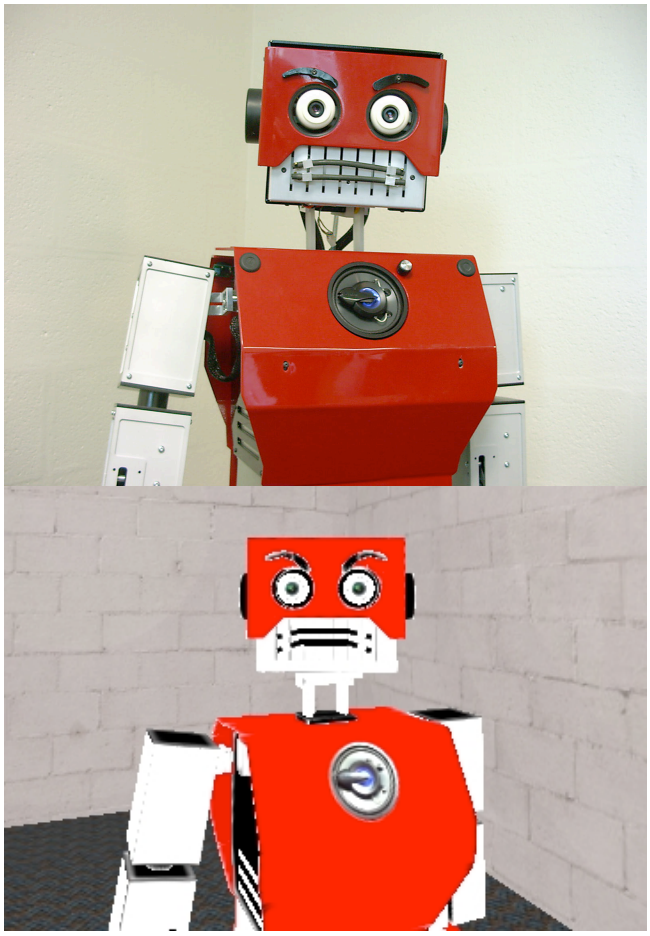


Fig. 4. Angry real CRAMER (Above), Angry virtual CRAMER (Below).

- Speech Production
- Natural Language Processing
- Planning
- Robot Base (e.g., Pioneer, Segway)

Of particular interest here is the *action manager*, ADE's goal management and action sequencing component, as it is the most frequent client of the USARSim server's services. The action manager uses the utility of goals along with information about goal deadlines to determine how resources should be allocated. This allows the system to pursue multiple goals simultaneously, so long as there are no resource conflicts (e.g., between a goal that requires the robot to remain stationary and a movement goal). When conflicts arise, the action manager gives precedence to the higher-priority goal. The action manager stores procedural knowledge in the form of action scripts that allow it to sequence multiple sub-actions together to accomplish a goal. For cases in which the action manager does not have a pre-defined script to achieve a goal, a planner component (based on the SapaReplan planner [4]) can construct scripts to achieve them.

The CRAMER server and USARSim server both implement the *HumanoidTorso* interface, which includes several methods for manipulating the arms, head, facial features, etc.

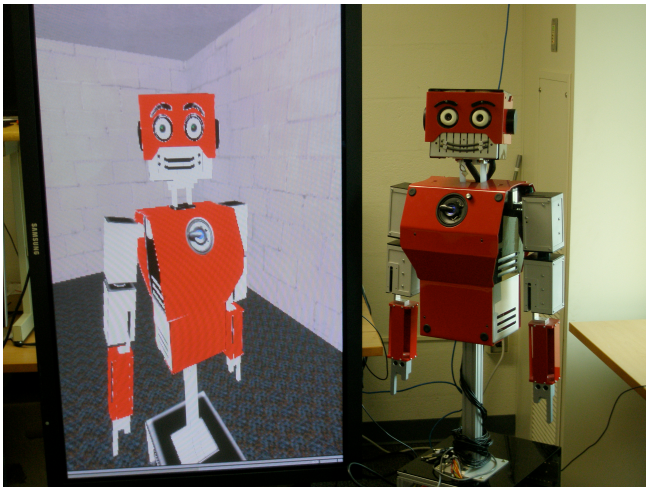


Fig. 5. Side-by-side view, as during the lab introduction described in the text.

Some of the methods are low-level, for example:

- `moveLeftArm/moveRightArm`
- `moveEyes`
- `moveHead`
- `moveEyeBrows`

whereas others are complex, higher level actions that build on the low-level interface, such as:

- `lookAt`
- `pointTo`
- `Frown`
- `Scowl`
- `Smile`.

Because the servers both implement the same interface, other ADE servers (e.g., the action manager server) need only request a reference to a *HumanoidTorso* server, and it can use whichever implementing instance (i.e., a server for the real or for the simulated robot) is returned. The USARSim server, in addition, implements the *PioneerServer* and *SICKLaserServer* interfaces, allowing access for ADE servers to the Pioneer and SICK laser range finder models included with USARSim.

V. REAL VERSUS VIRTUAL INTERACTIONS

The simulated version of the humanoid robot will allow us to explore important questions in human-robot interaction related to how people respond to simulated robots (see Section VI below). However, the validity of those experiments will depend, in part, on how closely the simulated robot models the behaviors of the real robot. We have tested the validity of the model in multiple contexts. Two of these scenarios are described below: a "dialogue" with a simulated and a real robot, and a "dance contest" in which the two robots employ the same algorithm to react (i.e., dance) to music. These scenarios allow us to evaluate the simulation by having the real and simulated robots side-by-side in the same environment; the simulated robot is displayed on a large

(63") plasma monitor, and hence can be displayed full-size, making it possible to eliminate the effect of size.

A. The Dialogue

In this scenario, the two robots perform the task of introducing the research done in the lab to visitors. The dialogue is fully scripted (very much in the way Disney animatorics work), so there is no geniune interaction with the people watching the robot. However, the two robots are programmed to respond to each other throughout their interactions. The real CRAMER begins the introduction by describing its own capabilities, explaining what its name stands for, and how it can be used in experiments in the lab. For example, because emotional expressions are very important for human-human interactions, familiar "emotion" expressions have been programmed for use by CRAMER; in the course of the dialogue, many of these emotional expressions are demonstrated. The robots behaviors are controlled by simple action sripts that are being executed by the action manager. A script for making the robot look "angry", for example, could look like this:

```
script lookAngry
moveLeftArm(5, 10, 90)
Scowl()
changeVoice(angry)
sayText("For example, I can look angry!")
```

Each of the script commands in this example invokes an action in a corresponding ADE server (the CRAMER or USARSim servers for the first two, the speech production server for the last two), producing a behavior in which the robot speaks in an angry voice while scowling and pointing angrily. `moveLeftArm` is a simple action in the robot servers, whereas `Scowl` is a compound action (also implemented in the robot servers) that builds on multiple lower-level simple actions to manipulate the robot's eyebrows, lips, and eyes.

While real CRAMER is introducing itself and the lab, virtual CRAMER "watches" it, nodding in (scripted) response to important points. When real CRAMER turns to and introduces virtual CRAMER, virtual CRAMER then takes over the introduction, demonstrating its own capabilities and elaborating on how having a virtual replica is useful for exploring the kinds of questions described in Section VI. The two robots then alternate back and forth for the remainder of the dialogue.

Note that the design of the servers plays an important role in the ease with which these behaviors can be scripted by the action manager. Because the USARSim server implements the same interface as the CRAMER server (as described in Section IV), the action manager can use the same scripts (such as `lookAngry` above) to evoke identical actions in the virtual agent. Hence, there is no need for the action manager to have any understanding of the difference between the two (or that there even is a difference).

B. The Dancing Robots

Another scenario used to demonstrate the functionality of the simulated version of CRAMER is the "dance contest." Once again, the robots are placed side-by-side (i.e., the large monitor is placed beside the read robot). The robot servers (CRAMER server and USARSim server) perform waveform analysis on the audio output of the selected song (Kraftwerk's *We Are the Robots*, naturally) to detect peaks. These are taken to approximate the beat of the music, and the servers generate random movements of various body parts to coincide with the peaks in the music. The effect is of two robots dancing to the music.

VI. DISCUSSION

The above examples of simple dialogues or synchronized behaviors between the real and the simulated robot are intended only as a proof of the functionality of the current interface, not as a demonstration of what its potential is for empirical studies of humans interacting with robots. Many empricial robot studies, particularly in the area of human-robot interaction, employ an experimental method that involves showing videos of real robots to subjects or having subjects interact with simulated robots, rather than having them interact directly with real robots. While there are certainly substantial benefits to the use of simulations in the development of complex robot architectures, it remains an open question whether experimental results obtained using simulations (or videos, images, etc.) are directly applicable to "best practices" in the design of architectures for human-robot interaction. In particular, the embodiment of the physically instantiated robot in the same physical space as the human subject is likely to have some effect on how the robot is perceived, how attention is allocated, where eye gaze focuses, etc.

For example, the simulated model of CRAMER will allow us to replicate an experiment that we conducted with CRAMER in a real environment, where the robot had to follow human eye gaze in real-time during a word learning task [2]. The interesting question then is whether the eye gaze patterns observed in humans interacting with the real robot will end up matching those to be observed in interacting with the simulated robot. If they match up, then we have learned something about eye gaze, namely that simulated robots do not necessarily have a different effect on attentional mechanisms in humans from real robots. If, however, they differ, then this effect will trigger a detailed investigation into the nature of attention and how it interacts with physical embodiment. And, of course, this is only one, immediate example of how the simulated version of CRAMER can be an invaluable research tool for psychology and HRI.

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented a simulation model of our humanoid robot CRAMER in the UNREAL simulation engine using the USARSim interface. We demonstrated how a completely actuated robotic simulation model can be developed with open-source tools and how the model can

be connected to a robotic architecture in such a way that from the architecture's perspective there is no difference between controlling a simulated versus a real robot. We also briefly discussed how such a simulated replica of a physical robot that attempts to be as faithful as possible to both the visual appearance as well as the physical behavior of the robot can be a very useful tool for human-robot experiments. In particular, we believe that such a tool is necessary for the systematic exploration of the effects that the physical embodiment of a robot has on humans interacting with it, compared to possibly different effects of a two-dimensional version of the same robot on a video screen.

We plan to expand the current USARSim interface in the future, in particular, the way in which the ADE USARSim server manages the sockets and lines of communication. Currently, we can only initialize one robot inside the UNREAL environment and control it through our ADE system. The plan is to generalize the interface so that a single USARSim client can handle multiple connections. We are also planning on developing "dummy objects", which are initialized in the same way as the robot agent, but with limited actuating capabilities. A dummy object may be a box with a single hinge, for example, on which an agent may perform such actions as "open" or "close". While such interactive objects are already possible within UNREAL, there is currently no way for a remote client to manage and control them. Yet, we believe that objects with limited behavioral and actuating capabilities will be of great use (e.g., as props) in HRI studies.

VIII. ACKNOWLEDGMENTS

This work was in part funded by ONR MURI grant #N00014-07-1-1049 to second author.

REFERENCES

[1] Steven Balakirsky, Chris Scrapper, Stefano Carpin, and Michael Lewis. Usarsim: Providing a framework for multi-robot performance evaluation. In *Proceedings of PerMIS*, 2006.

[2] You-Wei Cheah, Matthias Scheutz, Chen Yu, Paul Schermerhorn, and Ikhyun Park. A multi-modal real-time interaction framework and platform for studying natural human-robot interactions. In *Proceedings of the International Conference on Multimodal Interfaces*, 2009 (Under Review).

[3] Robert Rose, Matthias Scheutz, and Paul Schermerhorn. Empirical investigations into the believability of robot affect. In *Proceedings of the AAAI Spring Symposium*. AAAI Press, 2008.

[4] Paul Schermerhorn, J. benton, Matthias Scheutz, Kartik Talamadupula, and Subbarao Kambhampati. Finding and exploiting goal opportunities in real-time during plan execution. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[5] Matthias Scheutz. ADE - steps towards a distributed development and runtime environment for complex robotic agent architectures. *Applied Artificial Intelligence*, 20(4-5):275–304, 2006.

[6] Matthias Scheutz and Paul Schermerhorn. Affective goal and task selection for social robots. In Jordi Vallverd and David Casacuberta, editors, *The Handbook of Research on Synthetic Emotions and Sociable Robotics*. IGI Global, 2009.

[7] Matthias Scheutz, Paul Schermerhorn, James Kramer, and David Anderson. First steps toward natural human-like HRI. *Autonomous Robots*, 22(4):411–423, May 2007.

[8] Matthias Scheutz, Paul Schermerhorn, James Kramer, and Christopher Middendorff. The utility of affect expression in natural language interactions in joint human-robot tasks. In *Proceedings of the 1st ACM International Conference on Human-Robot Interaction*, pages 226–233, 2006.

Exploring Spoken Dialog Interaction in Human-Robot Teams

Matthew Marge, Aasish Pappu, Benjamin Frisch, Thomas K. Harris, and Alexander I. Rudnicky

Abstract We describe TeamTalk: A human-robot interface capable of interpreting spoken dialog interactions between humans and robots in consistent real-world and virtual-world scenarios. The system is used in real environments by human-robot teams to perform tasks associated with treasure hunting. In order to conduct research exploring spoken human-robot interaction, we have developed a virtual platform using USARSim. We describe the system, its use as a high-fidelity simulator with USARSim, and current experiments that benefit from a simulated environment and that would be difficult to implement in real-world scenarios.

I. Introduction

An enduring challenge in human-robot interaction is creating interfaces that are both effective, in that proper behaviors occur, and natural, in that humans can interact with robots on a level closer to goals at hand. We are interested in the scenario where humans need to interact with multiple robots at the same time, a situation that particularly stresses the need for effective communication. At the core of the problem is the management of human-robot teams and sub-teams, where each has different roles and responsibilities. Spoken language has the potential to reduce the complexity of this interaction, by allowing humans and robots to communicate on a more abstract task level rather than in terms of a more structured operator/device relationship.

This paper describes TeamTalk: a human-robot interface capable of interpreting spoken dialog interactions between humans and robots in virtual and real-world spaces [1]. This system currently manages goal-oriented dialog in a search domain (which we will refer to as the "Treasure Hunt"). TeamTalk was developed using the Olympus architecture for the dialog interface [2]. The system incorporates live map updates in virtual and real environments and allows the execution of complex action sequences, called "plays". The PlayManager component is taken from RoboCup-related work [3]. Plays can be either individual or team-based actions, such as searching an area for victims or treasures. Low-level task allocation is managed via the TraderBot component [4]. These components were integrated into a coherent single application as part of the TeamTalk system [5]. More details on these components can be found in Section IV.

The complexity of the full system (including major software components developed in separate sub-projects, as well as 3-4 robots) presented a logistical challenge that led us to implement a simulation-based analog that would allow us to experiment with the interaction component. Development and testing was facilitated through the use of USARSim as a virtual testing platform [6]. USARSim is a simulation platform designed for evaluating and conducting research with urban search-and-rescue robots. The simulated environment and robot models are rendered with the Unreal Tournament 2004 game engine. We run simulated robots using MOAST's "SIMware" software, which serves as a replacement for real-world robotics hardware [7]. MOAST is a mobile robot framework that interfaces directly with USARSim.

Simulation allows us to make the development process more streamlined. At the same time, we maintained the same interface layer that is used for the real system, allowing us to move between the two environments with relative ease. Moreover, we designed the virtual world to mirror the environment in which the full system was being run.

TeamTalk with USARSim has also served as a robust testbed for conducting user studies that explore linguistic aspects of human-robot dialog. This virtual platform has a relatively low development and maintenance cost as compared to using actual robots. Current research focuses on multi-participant human-robot dialog, and on the interpretation of spatial language in human-robot dialog. Practical benefits of the virtual platform include the ability to perform user studies without incurring the logistical costs of a real-world full trial, and also reducing the chance that studies will experience technical difficulties. With TeamTalk interfaced to the virtual platform, we have collected several datasets of real-time spoken language interactions with virtual robots. By varying the scenario and mode of interaction, this platform permits us to explore what people say when speaking to robots in the context of goal-oriented tasks (e.g., exploring a room, moving to another teammate's location).

The organization of this paper is as follows. Section II elaborates on previous work with speech interfaces in USARSim. Section III describes the Treasure Hunt task. TeamTalk system features are discussed in Section IV. The

Manuscript received July 16, 2009. This work was supported in part by the Boeing Company Grant CMU-BA-GTA-1. The content of this paper does not necessarily reflect positions or policies of the Boeing Company and no official endorsement should be inferred.

Matthew Marge, Aasish Pappu, and Alexander I. Rudnicky are with the Language Technologies Institute at Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: mrmarge@cs.cmu.edu, aasish@cs.cmu.edu, air@cs.cmu.edu).

Benjamin Frisch is currently in the Computer Science Department at University of Wisconsin-Madison, Madison, WI 53706 USA (email: bfrisch@wisc.edu).

Thomas K. Harris is currently with EDalytics, LLC, Pittsburgh, PA 15217 USA (email: thomas@edalytics.com).

TeamTalk architecture, including its connection to the virtual platform, is described in Section V. Research projects making use of the virtual platform are presented in Section VI. We offer concluding thoughts in Section VII.

II. Related Work

Speech interfaces for USARSim-based systems have been previously explored. For example, a speech interface has been proposed for the airport tug domain, where robot tugs holding cargo could be commanded to move to different locations [8]. USARSim robots were used in simulation as part of a preliminary testing phase for cargo-moving tasks.

Other speech-based simulations have robots working as assistants to people. With the LiSA platform, robots transport small lab equipment around well-structured, delicate biological lab settings [9]. LiSA-based robots are interfaced through speech and touchscreen interactions. USARSim has been used as a simulator for their platform, but the interaction was designed to be between just a single robot and a single human.

"Smart home" robots have been developed using the Agent Development Environment (ADE), where agents such as robots share room-related information about the user [10]. Natural language dialog is made possible in ADE through the use of the SmartKom system [11]. ADE has used USARSim as part of a research platform that explores differences between human-robot interactions with virtual robots and with real-world robots.

TeamTalk extends the use of speech with USARSim by addressing communications within teams of robots and humans performing goal-oriented tasks.

III. The Treasure Hunt Domain

In the Treasure Hunt, a team consisting of robots and humans is tasked with locating "treasures" (color-coded objects) that are scattered throughout an indoor area. At the start of the task, the treasure locations are unknown and the area unexplored. In addition, team members may have roles assigned before the start of the task, based on their capabilities, such as exploration or treasure retrieval (the act of picking up a treasure and returning it to base). Teams may or may not have knowledge about the number of treasures in the environment, and are not aware of the hazards associated with the area. This makes the exploration task most appropriate for robot teammates. At any time, a human may query a team member's location and status. A robot team member would respond with its current assigned task, if any, and location (e.g., "Alphie here. I am moving to home.").

When a teammate locates a treasure, robots become aware of the location through internal communication while humans may view the treasure on their live map interface (displayed on a tablet computer). The Treasure Hunt GUI is shown in Figure 1. Currently, the treasure retrieval task is performed by a sub-team of a human (operating from a distance of about ten meters away from the remainder of the team), a navigation robot, and a robot specialized in locating

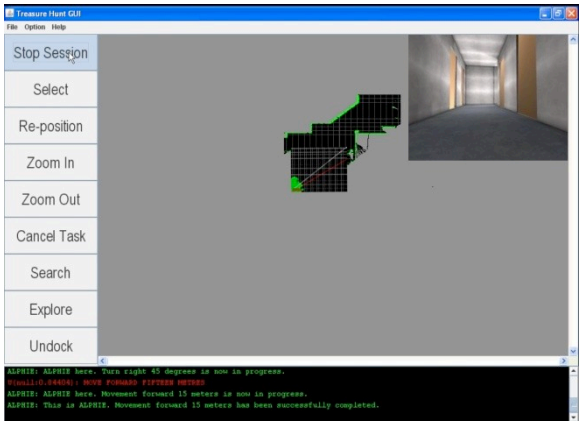


Fig. 1. The Treasure Hunt GUI, with a USARSim hallway overlaid in the upper-right corner.

the treasure. The team carrying a treasure must return to the home location to complete the task.

Humans have several roles: they must manage high-level goals to teams, query robot locations, and decide how robots should perform search tasks. At any time, a human can stop a current task or reassign a robot to a new task. Also, humans may also perform team-based subtasks, especially if they are most fit for the job, such as picking up a treasure from a safe location.

Robots have roles in the Treasure Hunt domain based on their capabilities. In both explorer and retrieval teams, Pioneer P2-DX robots traverse the area and collaboratively build an occupancy grid (using SICK laser range-finders) that notes obstructions, hallways, and doors. Currently, the Treasure Hunt scenario has Pioneers traverse a large open area used by several different robotics projects and thus a topography that varies over time. A low-level obstacle avoidance system prevents robots from colliding with other robots or with obstructions. The occupancy grid that teams of Pioneers build collaboratively is displayed on each human's Treasure Hunt GUI. This environment has been replicated in the TeamTalk virtual system using USARSim and Unreal Tournament map-building tools.

The treasure-hunting includes at least one Segway Robotic Mobility Platform (RMP). This robot is mounted with a high-resolution camera that allows it to follow a Pioneer robot and to spot treasures. As the Segway follows an exploring Pioneer, it shares its high-resolution images with the humans via the Treasure Hunt GUI. When a Segway locates a treasure, it notifies the team in two ways. First, it announces that a treasure has been found, via speech; the Treasure Hunt GUI also displays the treasure on a map. All of these roles are replicated in the USARSim-based virtual system.

IV. System Features

TeamTalk, the human-robot interface, is capable of interpreting speech, mouse clicks, and pen gestures from users. Although it is designed for tablet PCs, it may be run on any Windows-based platform. In the Treasure Hunt task, a human user operates TeamTalk with a tablet PC and an attached headset microphone. The user can instruct robots

and robot teams with speech or pen-based gestures. As an alternative, TeamTalk interprets typed text as a substitute for speech.

TeamTalk also displays a live representation of the robots' environment via its GUI. This includes an occupancy grid that is updated using information generated by the Pioneer robots as they traverse the environment. In the virtual platform, USARSim range scanner sensors are attached to the Pioneers and have access to the ground truth map. A live feed of high-resolution images from the Segway robot is also displayed on the GUI; this is replaced by USARSim images in the virtual environment. TeamTalk also displays the status of all robots involved in the task, along with a trace of the conversation history from the start of the task. Robot status information displayed on the GUI includes the robots' locations and orientations to the best of their knowledge and their current task assignment.

A user can instruct a robot or team of robots with varying levels of detail. At the high end of commands, robots can move to named locations (such as their starting point), explore an area, and search an area for treasure. In the exploring task, Pioneers traverse an area specified by pen gestures on the occupancy grid. As they traverse the area, the TeamTalk map is updated to account for obstacles, open spaces, and moving objects. The search task has Pioneer robots explore the area, with at least one Segway following, looking for treasure in the environment.

Robots are also able to process low-level "turn-by-turn" commands. At any time, a human can ask a robot to move a specified number of meters in any relative direction or cardinal direction (e.g., "Move forward five meters"). A human can also have a robot turn a specified number of degrees in any relative direction or cardinal direction (e.g., "Turn right ninety degrees"). TeamTalk is also capable of interpreting a natural language combination of moves and turns together in a single command. These may be used to get a robot out of a difficult situation or to navigate a delicate environment.

Consider a task where a robot must process a series of instructions. Such scenarios demand a robot's understanding of task division and input requirements for each component subtask. The robot instantiates a conversation to receive inputs from the user and acknowledge the in-progress subtask. If more information is necessary, such as if a user simply asked a robot to "move forward" without specifying a number of meters, the robot can prompt for this information.

We call this plan of tasks a "play," a term borrowed from the RoboCup domain. As defined in [12], a play, P, is a fixed team plan which consists of a set of applicability conditions, termination conditions, and N roles, one for each team member. Each role defines a sequence of tactics {T_1, T_2...} and associated parameters to be performed by that role in the ordered sequence. Assignment of roles to team members is performed dynamically at run time. Upon role assignment, each robot is assigned its tactic T_i to execute from the current step of the sequence for that role. Tactics, therefore, form the action primitives for plays *that influence the surrounding environment. PlayManager permits robots to perform these "plays" [13].*

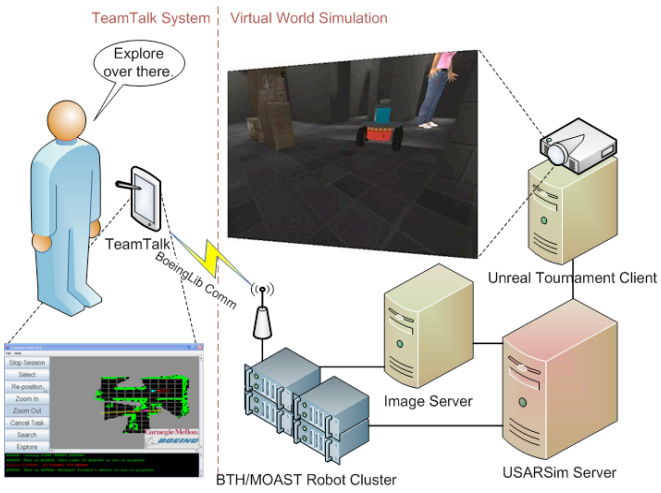


Fig. 2. Overview of TeamTalk with USARSim

TraderBot is a low-level robot team management system that dynamically assigns roles in plays to different robots, using an auction mechanism. For example, if the task is to explore a specified area, robots in the team "bid" on the task, with the highest bidder being the one that is closest to the goal location and relatively idle. This low-level management is not directly controlled by the human user (nor is it intended to be). In principle the TraderBot mechanism allows tasks to be automatically rebid in case a team member drops out, and more generally permits robots to dynamically adjust their level of autonomy according to the given task [13]. TeamTalk, TraderBot, and PlayManager communicate directly with USARSim and MOAST components in the virtual system.

V. System Architecture

The TeamTalk system consists of several major sub-systems, which we now describe. TeamTalk coupled with the virtual system is no different than TeamTalk in real environments since the MOAST robots have been extended by incorporating the TraderBot and PlayManager components. The MOAST robots have been extended to use the BoeingLib communications protocol.

A. Treasure Hunt Multimodal Interface

The front end for TeamTalk handles user input and displays the status of each robot involved in the Treasure Hunt task. The Treasure Hunt GUI displays all the controls necessary to manage the robot team. It also displays an occupancy grid of the robots' shared representation of the environment, the locations of the robots, and the recent conversation history. Commands may either be spoken through a headset microphone connected to the computer running the Treasure Hunt GUI or by typing directly into the GUI itself. Upon initialization, the Treasure Hunt GUI reads a configuration file that specifies the IP addresses of the robots (either real or virtual) and the map server. Once the session has been initiated, the robots involved in the task report their status and are ready to begin the Treasure Hunt

task. As part of the Olympus Spoken Dialog Framework, the Treasure Hunt GUI uses the Galaxy Communicator architecture for message communication with Olympus-based dialog components.

B. Olympus Spoken Dialog Framework

This subsection elaborates on the underlying spoken dialog framework as shown in Figure 3. The speech component is implemented using Olympus [2]. It consists of all the processes that are necessary to maintain a task-based dialog with a human user. Among the components involved are those that keep track of conversation state, record speech, decode it (using Carnegie Mellon's PocketSphinx speech recognition engine), annotate speech for confidence, parse the input to extract its semantics, generate language and produce synthesized speech or display elements.

Olympus contains several components that handle the acquisition of user input. An audio server performs voice activity detection, acquires user input, sends the acoustic data to the PocketSphinx decoder, and collects recognition results. The Logios language model-building component is used to automatically create a dictionary and language model based on the grammar specified by the developer [14]. The Phoenix server parses decoded speech using a context-free grammar.

TeamTalk contains multiple domain-specific instances of dialog managers, each associated with a robot. Each robot internalizes its own conversation state processes with an instance of the RavenClaw dialog manager [15]. This allows for multi-agent interaction. A *dialog manager* ***models the state of the conversation between interlocutors at each step of the interaction and tracks the next task that the robot should perform. For the Treasure Hunt domain, we declare a number of*** *agencies*, ***or tasks, that a robot can perform. A robot's capabilities are incorporated into a dialog manager, including the ability to report a location, move to a destination, or turn, or undertake an activity such as search. These abilities are hierarchically organized into tasks and subtasks.***

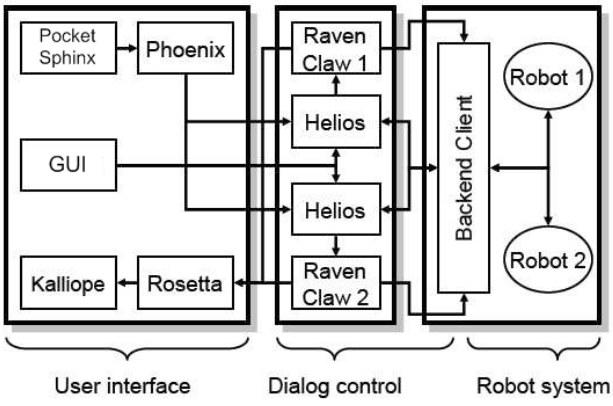


Fig. 3. TeamTalk system architecture, which uses the Olympus Spoken Dialog Framework.

Every task that is assigned to a robot has certain requirements and prerequisite conditions that need to be fulfilled [15]. For example, a robot may need to move a certain distance before it can turn to the right. Such requirements are part of the dialog task. A *dialog task* ***is an ordered list of agents that is used to dispatch inputs to appropriate subagents in the task. An*** *agent* ***is a particular type of handler to the ongoing conversation. For instance, a REQUEST agent asks and listens for certain user inputs that are relevant for the current task. Similarly, an INFORM agent generates follow-up prompts as an acknowledgement to user input. User input is bound to agree with a*** *concept type* ***(e.g., a*** `yes_no` ***question is bounded to a Boolean concept type; a*** `where_is`***-type request is bounded to a string concept type). Additionally, a RavenClaw instance can handle complex concept types like arrays, frames and structures that are necessary when we are not informed about the size of list-type inputs.***

Besides syntax-based bounding brought about by the speech recognizer's language model, RavenClaw allows us to restrict the scope of user input to certain semantic concepts using grammar mappings. Grammar mapping binds a particular user input as an instance of a semantic concept. For instance, if a user responded that his name is *John*, ***RavenClaw's grammar mapping binds user input to the semantic concept*** *name*. ***It is necessary to add an entry in the system vocabulary for the name*** *John* ***under the concept*** *name*. ***The TeamTalk backend is responsible for communicating with the robots involved in the task. When the dialog manager decides on the course of action that a robot should take, TeamTalk passes that message to the backend. The communication typically involves high- or low-level task instructions from the user. Messages are passed between TeamTalk and robot teammates via Boeing's communication libraries.***

At any point in the dialog, a robot may have a concept it needs to convey to the user. This happens most often when a RavenClaw instance decides on the next task to perform. Rosetta-based natural language generation produces natural language text from RavenClaw dialog concepts. This component is customized for the Treasure Hunt domain, and consists of a set of templates with variables. We believe that this can be adapted to other domains, as the range of communications that a robot needs to convey is limited. Once natural language text is produced by Rosetta, Kalliope, the text-to-speech controller, synthesizes the text into speech and plays it in the user's headset. All components involved in the spoken dialog interaction are integrated in the Olympus architecture.

C. USARSim Configuration

An overview of USARSim integration with TeamTalk is shown in Figure 2. While robots are part of a treasure hunt rather than an urban search-and-rescue, we were able to modify the existing victim object (USARVictim) to be displayed as a color-coded treasure, which may be displayed in the virtual system. The virtual human could then pick up the treasure and return the item to home base.

To simplify development of the TeamTalk component, we developed a map simulating the actual environment to use with USARSim. Digital blueprints from the CMU Robotics Institute High Bay along with pictures with the locations of windows, stairs, bridges, and tables were used to create the

map. An example replication is shown in Figure 4. We measured each wall from the blueprint and created the proper areas, appropriately scaled to the USARSim environment. We were able to use the already-existing models in Unreal Tournament to add vehicles and stairs, which we also scaled to fit the map. The map also includes the stairs to the walkway on the second floor of the High Bay in addition to all of the rooms on the High Bay level of the building, providing us with an additional environment for testing. We also included an area representing the exterior in order to leave open the possibility of exploring outdoor scenarios. We found the tutorials on map development that came with the Unreal Tournament 2004 Editors Choice Edition (UT) to be sufficient to create the map. We also found that even though the map has many blocks and was slow to render, UT was able to display it at normal speeds on machines satisfying the typical UT system requirements (Pentium 4 processor, video card with 128MB of memory).

By default, USARSim starts the server in spectator-only mode. We modified the default game configuration (USARGame) and created a customized TreasureHuntGame which inserts an instance of a character into the simulation. A subclass of UnrealPawn, UT's player scripting class, was created that enlarged the human player mesh to the appropriate scale in USARSim and set the player's identification texture. Each time a human joins the environment, the TreasureHuntGame spawns a virtual character that has one of five uniform colors in a rotating order. A human player is necessary to open gates, doors, or the garage for the robots. The doors remain open for about five seconds to simulate real-world actions, but this length of time is configurable. The robots can then be asked to proceed through the opening. To make the player in the environment more appropriate for this task, we removed weapons attached to the player and in the environment. There is work in progress to add a tablet PC to the virtual character's hands. We found character creation to be a challenge due to the need to replicate human-like movements; most characters available on the web are creatures suitable for gaming and not for our kind of work. A repository of reasonable human figures would be an asset to USARSim-based research.

D. MOAST Integration

MOAST is configured to launch a user-configurable number of USARSim P2-DX robots inside of the simulated High Bay as a default UT Map. We are able to then connect a UT client to the USARSim Server to observe the robots, which are controlled by TeamTalk's connection to MOAST. We modified the USARSim game to load the virtual character model at an appropriate scale relative to the map. As a result, we are able to explore the room with the robots like a human teammate in the real-world scenario. We have also started work on a model of the Segway RMP. The USARSim manual clearly explained how to create a new robot, and the physics engine was able to keep the robot always upright.

Since the Treasure Hunt task typically requires the use of several robots, system resources can be demanding, even in virtual simulations. Initially, TeamTalk's MOAST configuration required one robot per computer. This proved ineffective as the number of robots outpaced the number of computers we had available. Currently, we use a single computer running VMWare, allowing us to instantiate multiple robots, each with its own network identity, maintaining compatibility with the real robot environment. Thus far, three robots can be instantiated on our equipment. Since each human runs the USARSim game within a UT client, any number of humans can be added to the environment. Two humans have successfully been added to the environment at any given time, though experiments with more humans are part of future plans.

Fig. 4. Comparison view of the real-world environment with the USARSim environment.

VI. Current Research

A. Multi-Participant Dialog

Dialog systems capable of handling multi-participant dialog may be beneficial to human robot interactions in domains such as treasure hunting and urban search-and-rescue. In fact, they become necessary when multiple teams are involved and are working towards a common goal. Therefore, assumptions that traditionally work for single-user dialog systems will fail. It becomes necessary to construct a policy that supports the needs of dynamic and asynchronous conversation between interlocutors.

USARSim provides us a reliable and relatively inexpensive manner in which to explore this challenge. Managing such conversations requires explicit

representation of the contexts and reasons behind the current dialog. To understand the relevance of the dialog with respect to context and situation, additional information is needed. This additional information includes the topology of the environment and history of events in the task. This data can be acquired through simulation testing. With USARSim, we are free to include a series of robots and humans in the Treasure Hunt map.

As can be observed in everyday team-based tasks, the number of addressees for an utterance can vary from one teammate to many. Adding to this complexity are dialogs within sub-teams, which tends to happen in this domain. At this time, we are beginning to address this issue by studying how turn-taking processes occur in human-human dialog [16]. For instance, a person who wishes to barge into another dialog should be dealt with carefully, without disturbing the ongoing conversation.

Consider the following scenario. Alice and Robot A comprise one subteam, and Bob and robot B comprise another subteam. If Alice gives an instruction to robot B at any time, robot B has to decide whether to ignore the instruction by Alice or suspend the latest instruction by Bob to follow the command issued by Alice. A conversation policy for turn taking helps robot B in the above-mentioned decision-making problem. Table 1 presents a few simple strategies that could govern turn-taking in multi-participant dialog.

TABLE I
POTENTIAL MULTI-PARTICIPANT DIALOGUE STRATEGIES

Strategy	Comments
Current speaker chooses the next speaker	This strategy is simple, but it curbs the freedom of other participants.
FIFO styled turn-taking	This strategy follows a typical meeting-style conversation policy.
Priority-centric yet dictated by external party.	This strategy comes with an assumption of task ranking and participant ranking, so that it allows multiple possibilities for the next turn, ranging from a low profiled participant reporting a less critical task to high profiled participant reporting a highly critical task. Also, this strategy leverages on the fact that real-life conversation between people takes place with respect to a priority hierarchy among them.

As a fallback to the above turn-taking mechanisms, embodiment of the robots and humans in the USARSim environment allows them to signal their willingness to take the conversational floor. These simulated embodiments act like a fabric to the turn-taking policy at the behavioral level.

B. Spatial language

An advantage to using a USARSim-based version of the Treasure Hunt task is the ability to rapidly obtain language data from users. Given the robustness of simulation testing, we are currently exploring spatial perspective-taking in human-robot dialog with USARSim. We intend to learn more about how people produce spatial language in reference to members of a human-robot team and later incorporate this knowledge into the TeamTalk platform. Such commands could include, for example, “Mok [a robot] move 4 meters to the right of Aki [another robot].” This has led us to design an experiment to assess how humans give simple dialog commands in reference to members of a human-robot team. We are in the process of extending spatial language processing to objects with associated ontologies.

In the study, the goal was to learn how people commanded a robot to move to a location in the 2-dimensional world relative to another robot using spatial language (e.g., “right” “left” “around”). Participants spoke their requests into a headset microphone and their speech was transcribed. Key findings from this study dealt with the varying configurations of the two robots in each scenario and the location of the goal point for the robots. The number and type of commands people gave varied based on the goal location. "Mok” was the robot that was commanded by participants to move to a location that was near “Aki,” the second robot. We found that people generally spoke in terms of 90° (e.g., “turn right”, “turn left”) or 180° turns (e.g., “turn around”). Also, we found that the orientation of Mok, the robot that needed to move, mattered most when it was facing right. This was also when it was directly facing two of the four potential goal locations.

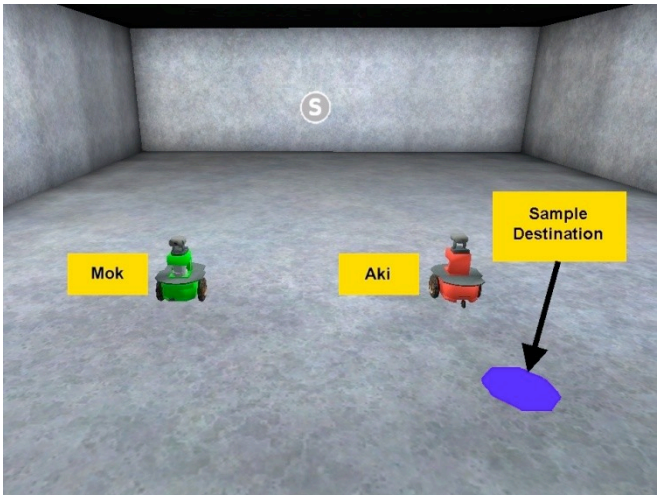


Fig. 5. Example stimuli from spatial language study. The goal destination is in purple.

Mok's other orientations required speakers to exert more effort, in terms of thinking time and the number of discrete steps spoken to move the robot to the destination. Here, we define a *discrete step* as one that causes or intends to cause a robot to move. Also, we found that Aki's orientation does not vary the commands; people treated it as a landmark. An analysis of the speech transcriptions from the study suggests that humans may use some form of internal grid to organize spatial-movement instructions. This is because participants use the length of Mok as a unit called a “step”.

Using the results from the exploratory analysis, we are currently conducting a follow-up study in a three-dimensional virtual setup with USARSim P2-DX robots, as shown in Figure 5. This required developing a USARSim

map that could permit participants to make decisions about moving Mok around a virtual environment while referring to Aki. The purpose of this second study is twofold: (1) to validate the results of the previous study and (2) to look into how the inclusion of metric units affects people's production of spatial language. We are in the process of conducting this follow-up study. The results of this study will provide us direction for developing a spatial reasoning component for TeamTalk that will be applicable to both real-world and USARSim-based situations.

VII. Conclusions and Future Work

By integrating a virtual system component into TeamTalk using USARSim, its usability as a research platform has improved. Additionally, the TeamTalk project development strategy has shifted since the integration of the USARSim virtual testing platform. Enhancements to TeamTalk are more easily tested by using the USARSim-based virtual simulation, as compared to testing with real robots. The simulation still maintains all the communication protocols that are used by robots –in the field". Furthermore, we can still elicit real-time human interactions with the virtual system. In addition, Unreal Tournament permits us to take the perspective of any robot or a bird's eye view of the scene in –Spectator Mode". We can also test outdoor scenarios with the virtual platform. Another benefit to using the virtual platform is that it facilitates collaboration with remote colleagues. Despite using the virtual system as our primary research platform, interesting results can still be validated in real-world studies, if necessary.

Future work in spoken language interaction with robots will involve using USARSim in Wizard-of-Oz (WoZ) experiments and ontology-driven robot navigation. Given the results from the spatial language acquisition studies, we anticipate developing varied forms of system responses, and testing these tuned responses in WoZ user studies. A researcher in these experiments will be controlling each robot's interactions with the user. Similarly, the aim of incorporating a Treasure Hunt ontology is to generate spatial representations that allow a human robot team to refer to objects using common sense in an environment. It is necessary to have a symbolic representation of the objects and their relationships with other objects. Conceptualization of these objects avails the opportunity for the robot to infer complex queries, such as –Face the door and walk until you find a window". Furthermore, we will assign attributes to each object to keep track of their status and update the sensory map, i.e., a robot's view of the world with the help of robot's sensory inputs. On the basis of detected objects (in the environment) and topological partitioning of the environment, a robot's knowledge about the world will be maintained.

Performing routine TeamTalk system checks is both necessary and relatively straightforward with USARSim. The transparency and flexibility of the USARSim project leads us to believe that we can expand the types of tasks associated with the TeamTalk project (e.g., collaborative problem-solving, robot learning by demonstration as done in [17]).

For more information on the TeamTalk project, please refer to the wiki, `http://wiki.speech.cs.cmu.edu/teamtalk`***. The maps associated with the Treasure Hunt task are located at the TeamTalk project's Subversion repository,***
`http://trac.speech.cs.cmu.edu/repos/teamtalk/trunk/usarsim`

Acknowledgments

The work described in this paper was supported by a university research grant from the Boeing Company. Bernadine Dias, Brett Browning, Brenna Argall, E. Gil Jones, Marc Zinck and Balajee Kannan collaborated with us on the overall Treasure Hunt project. We would like to thank Satanjeev Banerjee for comments on earlier drafts of this paper.

References

[1] T. K. Harris and A. I. Rudnicky, "TeamTalk: A platform for multi-human-robot dialog research in coherent real and virtual spaces," in *The Twenty-Second AAAI Conference on Artificial Intelligence*, 2007.

[2] D. Bohus, A. Raux, T. K. Harris, M. Eskanazi, and A. I. Rudnicky, "Olympus: an open-source framework for conversational spoken language interface research," in *HLT-NAACL 2007 workshop on Bridging the Gap: Academic and Industrial Research in Dialog Technology*, 2007.

[3] E. Jones, et al., "Dynamically Formed Heterogeneous Robot Teams Performing Tightly-Coordinated Tasks," in *International Conference on Robotics and Automation*, 2006, pp. 570-575.

[4] M. B. Dias, R. M. Zlot, M. B. Zinck, J. P. Gonzalez, and A. Stentz, "A Versatile Implementation of the TraderBots Approach for Multirobot Coordination," in *International Conference on Intelligent Robots and Systems*, 2004.

[5] (2008, Dec.) Treasure Hunt Project. [Online]. http://www.cs.cmu.edu/~treasurehunt/

[6] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "USARSim: a robot simulator for research and education," in *International Conference on Robotics and Automation*, 2007, pp. 1400-1405.

[7] S. Balakirsky, C. Scrapper, and E. Messina, "Mobility open architecture simulation and tools environment," in *International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, 2005, pp. 175-180.

[8] A. M. Olney, "Multi-robot dispatch," in *International Joint Conference on Artificial Intelligence 5th workshop on Knowledge and Reasoning in Practical Dialogue Systems*, 2007, pp. 42-45.

[9] E. Schulenburg, et al., "LiSA: A Robot Assistant for Life Sciences," *Lecture Notes in Computer Science*, vol. 4667, pp. 502-505, 2007.

[10] P. Schermerhorn and M. Scheutz, "Natural Language Interactions in Distributed Networks of Smart Device," *International Journal of Semantic Computing*, vol. 2, no. 4, 2008.

[11] W. Wahlster, N. Reithinger, and A. Blocher, "SmartKom: Multimodal Communication with a Life- Like Character," in *7th European Conference on Speech Communication and Technology*, Aalborg, Denmark, 2006.

[12] B. Browning, J. Bruce, M. Bowling, and M. Veloso, "STP: Skills, tactics and plays for multi-robot control in adversarial environments," *IEEE Journal of Control and Systems Engineering*, vol. 219, pp. 33-52, 2005.

[13] M. B. Dias, et al., "Dynamically Formed Human-Robot Teams Performing Coordinated Tasks," in *AAAI Spring Symposium: To Boldly Go Where No Human-Robot Team Has Gone*, 2006.

[14] W. Ward and S. Issar, "Recent improvements in the CMU spoken language understanding system," in *ARPA Human Language Technology Workshop*, Plainsboro, NJ, 1994, pp. 213-216.

[15] D. Bohus and A. Rudnicky, "The RavenClaw dialog management framework: Architecture and systems," *Computer Speech & Language*, vol. 23, no. 3, pp. 332-361, 2009.

[16] H. Sacks, E. A. Schegloff, and G. Jefferson, "A Simplest Systematics for the Organization of Turn-Taking for Conversation," *Language*, vol. 50, no. 4, pp. 696-735, 1974.

[17] B. Argall, B. Browning, and M. Veloso, "Learning by Demonstration with Critique from a Human Teacher," in *Second Annual Conference on Human-Robot Interactions (HRI 2007)*, Washington, D.C., 2007.

USARSim and HRI: from Teleoperated Cars to High Fidelity Teams

Michael Lewis, *Member, IEEE*

***Abstract*— USARSim began as a human-robot interaction (HRI) research tool but has since found use in a much wider community and for purposes we had never envisioned. This paper describes a six year HRI research program at the University of Pittsburgh using the simulation. Our original work involved teleoperated control of single robots and primitive simulations. In the most recent experiment teams of operators were controlling 24 robot teams in a high fidelity environment. In between we developed and tested measures of coordination demand, tried out news ways for managing video generated by teams, and investigated scaling effects as operators controlled increasing numbers of robots. This paper provides a brief chronology of this research summarizing their designs and findings.**

I. Introduction

In 2000 in response to rapidly rising costs of academic virtual reality software, we began experimenting with game software as an alternative. After a review of the most suitable engines we chose Unreal Tournament [1] over Quake [2] because of its object-oriented design and convenient java-like scripting language. Our first game engine-based application, CaveUT [3], software for creating multi-projector cave-like displays, was completed in 2001 and reported in a special issue of Communications of the ACM [4] we organized to highlight research groups who had independently begun working with game engines. We developed UTSAF [5], software using the game engine as a stealth viewer (3D visualization) for the ModSAF [6] military simulation shortly thereafter.

Work on USARSim began in late 2002 under an NSF ITR grant to study Robot, Agent, Person (RAP) teams in Urban Search And Rescue (USAR). Because our primary research interest was in human-robot interaction and most USAR robots rely on teleoperation from camera video, accurate simulation of video was our primary concern. As work was beginning, Epic games released Unreal Tournament 2003 which included the Karma physics engine [7] relieving us from simulating behavior manually and dramatically strengthening the engine as a simulation tool.

Manuscript received August 15, 2009. This work was supported in part by AFOSR grants FA9550-07-1-0039, FA9620-01-0542 and ONR grant N000140910680

Michael Lewis is with the School of Information Sciences, University of Pittsburgh, Pittsburgh, PA 15260 phone: 412-624-9426; emal: ml@sis.pitt.edu.

II. Teleoperation with single robots

Our initial HRI research using USARSim explored two areas: situation awareness for attitude and camera and viewpoint control. A review of findings such as McGovern's [8] observation that all recorded robot rollovers at Sandia had involved teleoperation using an onboard camera led us to suspect that camera geometry was leading to what we called the "fixed camera" illusion. You will notice this effect if you drive a robot up a ramp. Because the camera is fixed to the robot chassis when the robot mounts an incline the camera will remain perpendicular to the surface making the ramp appear flat and level. Even if attitude data is displayed nearby on an artificial horizon or other analog display it remains difficult for the operator to integrate that data with the camera video being used to drive the robot. As a consequence operators controlling from a fixed camera are prone to driving robots onto dangerously slanted surfaces risking rollovers and other problems. One potential solution is to reference the camera to gravity rather than the robot's chassis. Now when the robot moves onto a slanted surface it looks slanted rather than flat. Because standard cameras do not come with gimbals for gravity referencing and delays associated with attitude sensing and servos might introduce even greater errors this potential solution would be expensive and difficult to test using real robots. In simulation by contrast it was easy to program the viewpoints to reference the chassis or true vertical. This first USARSim experiment reported in [9,10] compared 26 participants controlling a robot using either a fixed (FC) or gravity referenced camera (GRC). The robots were driven across irregular outdoor and indoor environments toward target beacons that could be seen from anywhere in the map. The GRC led to less extreme roll/distance traveled, lower times to completion, and less backing-up (needed to retreat from impassable terrain). An examination of reported cues highlighted the importance of including some part of the chassis within the GRC view to help gauge robot orientation relative to the scene.

The viewpoint control experiments were an outgrowth of earlier studies [11] of "attentive navigation", a technique for automating viewpoint control. As an actor moves through a (virtual) environment he/she may look straight ahead in the direction of travel or pan from side to side to capture a fuller understanding of what is visible from that location. Conventionally this process is automated by planning the

agent's path so the viewer becomes a sight-seeing passenger able to look about freely. In attentive navigation, control of gaze is automated instead. Now the agent plans his own path through the world but control over where he looks is automated. This can make a lot of sense in virtual environments where the author may wish to direct a user's attention to some particular object or area but is less generally applicable to robotics. There are, however, similar issues related to coupling camera views to the direction of motion (straight ahead), pan-able views, and object tracking views. A long standing difficulty in mobile robotics involves moving cameras that are inadvertently left off axis when the robot is moved. The path appears clear in the side-pointing camera so the operator drives directly into an obstacle [12]. If the camera isn't movable, however, the operator may need to execute an elaborate dance often losing sight of the target in order to maneuver to obtain a desired viewpoint on an object.

An initial experiment [13] compared 65 operators in five conditions.

- Single Fixed Camera, No orientation indicator
- Single moving Camera , No orientation indicator
- Single moving Camera, orientation indicator
- Multiple moving Cameras, No orientation indicator
- Multiple moving Cameras, orientation indicator

For the first three conditions comparing single cameras the fixed camera eliminates the problem of off axis driving but at the cost of making it difficult to obtain different perspectives. The moving camera makes it easy to inspect the environment but at the risk of off axis driving. The orientation indicator provides an aid for restoring the moving camera to straight ahead before driving. In the multiple camera conditions the operator has the option of keeping one camera pointed in the direction of travel for driving while using the other to search the environment. The orientation indication provides assistance for returning a camera to straight ahead for driving if the operator chooses to use both cameras to search the environment.

Operators had a two step search task. First they needed to locate red cubes scattered throughout an indoor and an outdoor environment. After finding the cube they needed to maneuver closer in order to locate and read a letter on a side of the cube. The experiment found no advantage for the orientation indication. Both the moving camera and the two camera conditions led to identifying more targets suggesting that the ability to visually search the environment was more important to task performance than accurate driving.

A follow on experiment [14] investigated object tracking. Object tracking, called orbiting by [15] is a variant of attentive navigation in which the viewpoint remains fixed on an object as its platform moves through the environment. So, in moving around an object, that object remains in view without panning or other effort by the operator. This method is easy to write in simulation and could plausibly be implemented using laser or other ranging data to localize the object. The experiment compared two groups of 13 participants each performing the "lettered-cube" search task. In the control group operators used two moving cameras as in the earlier experiment. In the experimental group one of the cameras assisted operators by initiating object tracking for nearby cubes. Assisted operators identified more cubes and spend substantially less time maneuvering to read the letter. These viewpoint control experiments support the use of multiple cameras and show that for task relevant assistance such as object tracking, automated control of view point is accepted and benefits operators.

These first USARSim experiments addressed general issues in HRI and teleoperation that did not depend on the fidelity of the simulation. All three studies investigated the teleoperation-from-camera-video which required accurate reproduction of video but only approximate fidelity in other dimensions. Simple car models based on the vehicle class were used in these studies and little attention was given to scaling of the platform or environment.

III. MrCS and High Fidelity Simulation

A. MrCS

Our initial studies satisfied us that the game-based simulation could provide a sound research tool but our project's goal was to investigate much larger systems involving multiple robots and humans. Such systems require a high degree of automation and to model that accurately required paying more detailed attention to robot configuration, sensors, and environmental models. USARSim first took on a recognizable form in this revision, reported in Wang [16] which replaced the original agent-based [17] architecture with a more conventional organization, added conventional APIs ,and developed detailed models of existing platforms and sensors. Jijun Wang [18] developed the MrCS (Multirobot Control System) around the same time to integrate USARSim with Machinetta, a proxy-based coordination infrastructure, and a GUI for interacting with the system. The robot proxy provides low-level autonomy such as guarded motion, waypoint control and middle-level autonomy in path generation. It also communicates between the simulated robot and other proxies to enable the robot to execute the cooperative plan they have generated. The user interacts with the system through the user interface which sends

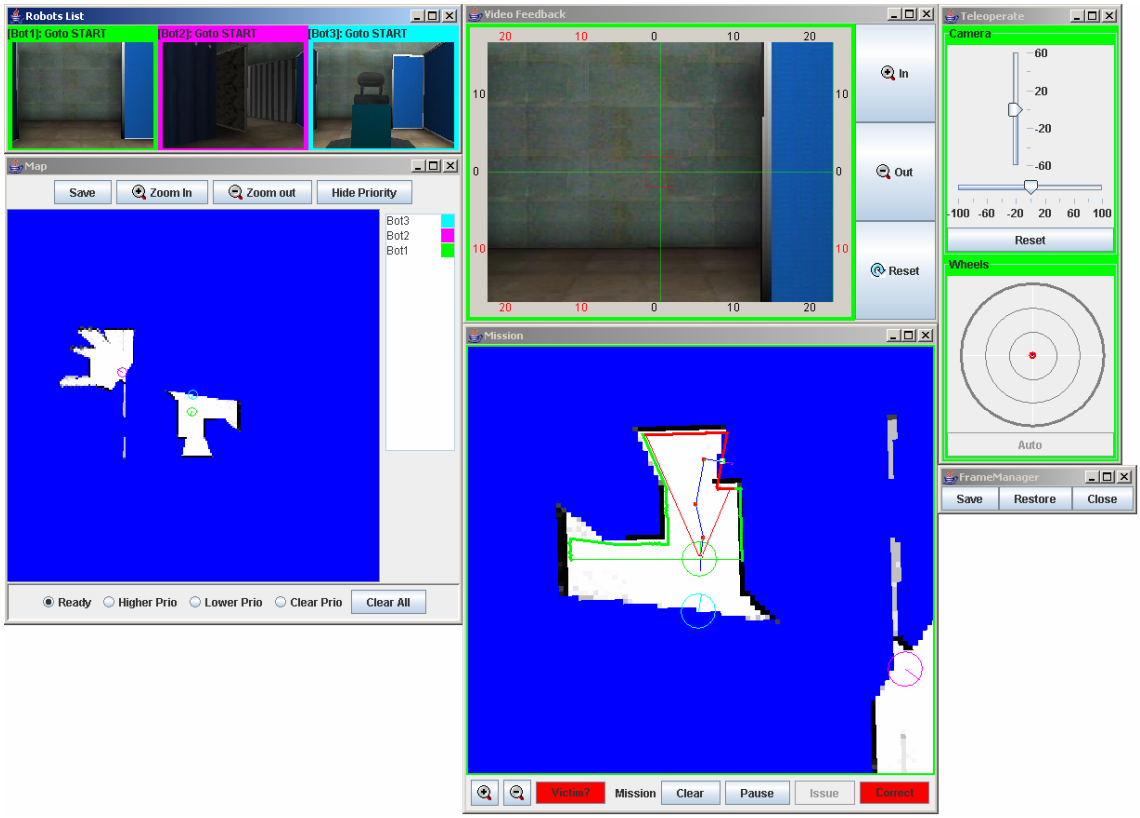


Figure 1. Multirobot Control System (MrCS) user interface

messages to robot proxies and reacts to their responses. Sensor outputs from the camera and laser go directly to the interface without passing through any proxy. A typical interface configuration is shown in Figure 1. On the left side are the global information components: the thumbnails of the individual's camera view (clicked to bring into focus); and the global Map (the bottom panel) that shows the explored areas and each robot's position. In the center are the individual robot control components. The upper component displays the video of the robot being controlled. The bottom component shows the controlled robot's local situation. The local map is camera up, always pointing in the camera's direction. Three increasingly sophisticated versions of MrCS can be downloaded from www.robocuprescue.org in the VR competition listings.

A. *Coordinating Teams*

The first experiment conducted using MrCS [18] compared manual and mixed-initiative control of 3 robots performing a USAR task followed shortly by an additional fully automated condition [19] to ensure that good mixed-initiative performance had not hiding superior automated performance. In the mixed-initiative condition operators could either teleoperate or assign waypoints. If a robot became idle it chose a waypoint at the nearest frontier and continued exploring.

In the experiment 14 participants searched for victims in both manual and mixed-initiative conditions in a counterbalanced repeated measures design. More area was explored and victims found in the mixed-initiative condition. Interestingly, switches in focus among robots was found to be correlated with good performance and operators switched attention more often in the mixed-initiative condition.

B. *Coordination Demand*

A theoretical controversy over the equivalence of computational complexity and human difficulty motivated the next set of experiments. Foraging tasks such as USAR allow robots to act more or less independently and we would expect increases in difficulty to be additive. Where robot actions are more interdependent, however, more frequent control might be needed making the task more difficult. Simply assigning robot roles in a plan, for instance, has been shown by Gerkey & Mataric [20] to be O(mn). If the complexity of choosing actions computationally approximates the difficulty of the task for a human, then it could be used to guide decisions about automation. Conversely, a human might be able to solve such problems heuristically making computational difficulty a bad estimator.

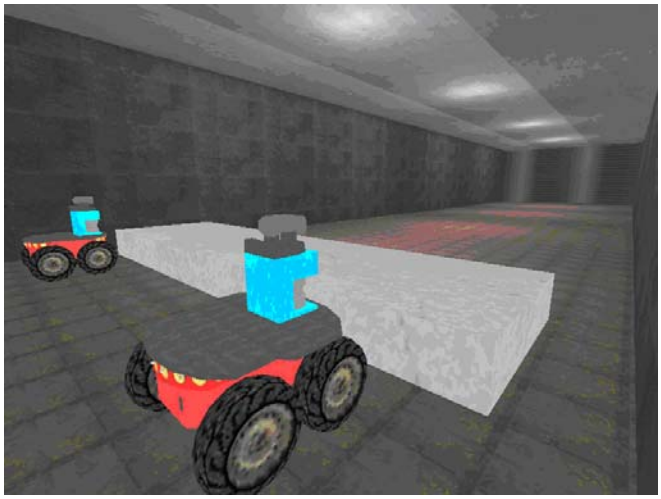


Figure 2. Tight coordination for box pushing

Figure 3. Explorer and Scout robots

These experiments were designed to evaluate coordination demand (CD) a proposed measure of the demand one robot's action(s) place on another. The measure is intended to extend Crandall's [21] neglect tolerance model to coordinating robot teams. The first experiment examined control of robots performing a box pushing task [22] (Figure 2). Fourteen participants controlled pairs of simulated P2-ATs using teleoperation or waypoint control and in the third condition a P2-AT paired with a P2-DX. As predicted CD=1 in the teleoperation condition as operators did not have time to do anything else. The heterogeneous pair showed higher CD, also as predicted. In a follow on experiment [23] seeking measures for less tightly constrained coordination a new definition of CD based on robot types was tested. The measure is based on the premise that CD involves marshalling the resources needed to perform a cooperative task. Since resources are held in common by robots of a particular type demand may be more accurately expressed and measured between types. Operators in this experiment controlled teams of robot pairs consisting of laser equipped explorer robots and camera carrying scouts (Figure 3). The operator needed to mark victims found using the scout's camera on the map generated by the explorer. Operators searched in three conditions with a 20 m explorer scan range (loosely coupled), a 5 m scan range (tightly coupled), or cooperative (explorer automatically follows the scout). Performance was as expected with the 20 m range leading to more victims and better performance with automated coordination.

C. *Scaling to Larger Teams*

While the question of fan-out (how many robots can an operator control?) is of general interest the question of how effects grow with team size offers greater promise for identifying bottlenecks and aspects of control best suited for automation. In a series of studies we have been investigating control of 4, 8, and 12 robot teams and in the most recent study teams of 24 robots. An experiment using a standard USAR task for 4, 8, and 12 robots [24] found a sharp decline

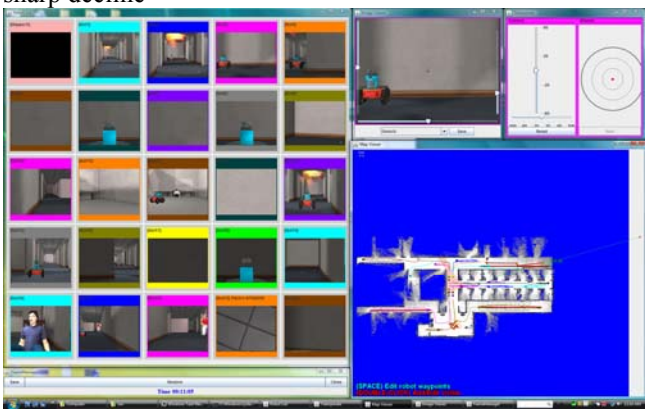


Figure 4. MrCS interface for 24 robots

in victims found and a slight decline in area explored between 8 and 12 robots. Two additional conditions [25] subdividing the operator's task into exploration (navigation) and perceptual search (scanning for victims) showed that effort involved in exploration accounted for most of the difficulty of the task and that victim finding performance was maintained by the perceptual search participants.

A similar investigation of scaling effects for use of static panoramas was less successful. In an earlier study [26] we compared use of streaming video from a team or 4 robots with still panoramas taken by robots at their terminal waypoints. The panoramas were marked on the map and could be accessed asynchronously as the operator found time to search them for victims. In the streaming video condition operators found slightly more victims and marked them with somewhat greater accuracy. We speculated that with more robots we might find an advantage for panoramas because of the greater moment-to-moment demand of monitoring streaming video. An experiment comparing these conditions for 4, 8, and 12 robot teams, however, replicated our earlier findings and showed a small but persistent advantage for streaming video.

In our most recent experiment [27] pairs of participants controlled 24 robots (Figure 4) in either a dedicated condition in which each was assigned control over 12 or a call center condition in which they were jointly assigned

control over 24. Results showed roughly comparable performance with slightly more area explored and victims found by participants in the dedicated condition. This experiment was intended as a control for studies in which we predict increased automation will alter the relative advantages.

IV. Discussion

USARSim was developed and remains an excellent platform for conducting HRI research. In this paper we have described 12 experiments conducted over 6 years using the USARSim platform. We are currently contributing to the UE3 port which we hope will lead to an even more effective experimental platform. While the choice of game engines was relatively easy in 2000 there is now a much broader range to choose from including open or inexpensive source alternatives. The advantages USARSim brings have shifted from the engine to the community. The true value of the simulation now lies in the substantial collection of models and validation data and our ability to share and maintain this common infrastructure.

References

[1] J. Gerstmann, "Unreal tournament: Action game of the year, 1999. GameSpot; www.gamespot.com/features/1999/p3_01a.html

[2] M. Abrash, "Quake's game engine: The big picture", Dr. Dobb's Journal (Spring 1977)

[3] J. Jacobson and Z. Hwang, "Unreal tournament for immersive interactive theater," *Communications of the ACM*, 45(1), 39-42, 2002.

[4] M. Lewis and J. Jacobson, "Game Engines in Scientific Research," *Communications of the ACM*, 45(1), 27-31, 2002.

[5] P. Prasithsangaree, J. Manojlovich, S. Hughes and M. Lewis, "UTSAF: A Multi-Agent-Based Software Bridge for Interoperability between Distributed Military and Commercial Gaming Simulation, Simulation", 80(12), 647-657, 2004.

[6] R. Calder, J. Smith, J. Courtemarche, J. Mar, and A. Ceranowicz. "ModSAF behavior simulation and control," *Proceedings of the Second Conference on Computer Generated Forces and Behavioral Representation*, STRICOM-DMSO, July, 1993.

[7] Karma Mathengine Karma User Guide http://udn.epicgames.com/Two/rsrc/Two/KarmaReference/KarmaUserGuide.pdf Accessed: 2009, 2002.

[8] D. McGovern, "Teleoperation of Land Vehicles". In Stephen Ellis (Ed.) *Pictorial Communications in Virtual and Real Environments*. New York: Taylor and Francis, 182-195, 1991.

[9] M. Lewis, J. Wang, J. Manojlovich, S. Hughes and X. Liu, X. "Experiments with Attitude: Attitude Displays for Teleoperation,". *Proceedings of the 2003 IEEE International Conference on Systems, Man, and Cybernetics*, IEEE, 1345-1349, 2003.

[10] M. Lewis. and J. Wang, J. "Gravity referenced attitude display for mobile robots: Making sense of what we see," *Transactions on Systems, Man and Cybernetics Part A*, 37(1), 94-105, 2007.

[11] S. Hughes and M. Lewis, "Partially guided viewpoint control in virtual environments," *Human Factors*, 47(3), 630-643, 2005.

[12] H. Yanco, J. Drury, and J. Scholtz, "Beyond usability evaluation: Analysis of human-robot interaction at a major robotics competition," *J. Human-Comput. Interact.*, vol. 19, no. 1–2, pp. 117–144, 2004.

[13] S. Hughes and M. Lewis, M. "Robotic camera control for remote exploration," *Proceedings of the 2004 Conference on Human Factors in Computing Systems* (CHI 2004), Vienna, Austria, 511-517, 2004.

[14] S. Hughes and M. Lewis, "Assisted viewpoint control for telerobotic search," *Proceedings of the 48th Annual Meeting of the Human Factors and Ergonomics Society*, pp. 2657-2661, 2004.

[15] S. Desney, G. Tan,G. Robertson and M. Czerwinski: "Exploring 3D navigation: combining speed-coupled flying with orbiting," *Proceedings of the 2004 Conference on Human Factors in Computing Systems* (CHI 2001): 418-425, 2001.

[16] J. Wang, "USARSim: A game-based simulation of the NIST reference arenas," unpublished, 2004.

[17] J. Wang, M.Lewis, and J. Gennari, "A Game Engine Based Simulation of the NIST USAR Arenas," *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, (1) 1039-1045, 2003.

[18] J. Wang, and M. Lewis, "Autonomy in human-robot control," *Proceedings of the 50th Annual Meeting of the Human Factors and Ergonomics Society*, pp. 525-529, 2006.

[19] J. Wang and M. Lewis, "Human control of cooperating robot teams," 2007 Human-Robot Interaction Conference, ACM, 9-16, 2007.

[20] B. Gerkey and M. Mataric. A formal framework for the study of task allocation in multi-robot systems. International Journal of RoboticsResearch, 23(9):939–954, 2004.

[21] J. W. Crandall, M. A. Goodrich, D. R. Olsen, and C. W. Nielsen. Validating human-robot interaction schemes in multitasking environments. IEEE Transactions on Systems, Man, and Cybernetics, Part A, 35(4):438–449, 2005

[22] J. Wang and M. Lewis, "Assessing coordination overhead in control of robot teams," *Proceedings of 2007 IEEE International Conference on Systems, Man, and Cybernetics*, 2645-2649, 2007.

[23] J. Wang, H. Wang and M. Lewis,."Assessing Cooperation in Human Control of Heterogeneous Robots," *Proceedings of the Third ACM/IEEE International Conference on Human-Robot Interaction* (HRI'08), 9-16, 2008.

[24] P. Velagapudi, P. Scerri, K. Sycara, K., H. Wang, M. Lewis and J. Wang, "Scaling effects in multi-robot control," *2008 International Conference on Intelligent Robots and Systems* (IROS'08), 2121-2126, 2008.

[25] H. Wang, M. Lewis, P. Velagapudi, P. Scerri and K. Sycara K. "How Search and its Subtasks Scale in N Robots," *Proceedings of the Forth ACM/IEEE International Conference on Human-Robot Interaction* (HRI'09),141-148, 2009

[26] P. Velagapudi, J. Wang, H. Wang, P. Scerri, M. Lewis, M., and K. Sycara, "Synchronous vs. Asynchronous Video in Multi-Robot Search," *Proceedings of first International Conference on Advances in Computer-Human Interaction* (ACHI'08), 224-229, 2008.

[27] H. Wang, S. Chien, M. Lewis, P. Velagapudi, P. Scerri, and K. Sycara, "Human teams for large scale multirobot control, *IEEE International Conference on Systems, Man, and Cybernetics* (to appear), 2009.

Validating a Real-Time Word Learning Model Tested in USARSim and on a Real Robot

Richard Veale and Paul Schermerhorn and Matthias Scheutz
Human-Robot Interaction Laboratory
Cognitive Science Program
Indiana University
Bloomington, IN 47406, USA
{riveale,pscherme,mscheutz}@indiana.edu

***Abstract*— In this paper we present a novel vision-based word-learning model that was developed and first tested using the USARSim simulation environment before being validated in the real world. We describe the learning architecture and the steps we took to integrate USARSim into the system. Models that were trained in the simulator evolve similarly to those trained on input from cameras, and perform comparably to their real-world counterparts on a subsequent *real-world* color recognition task.**

I. INTRODUCTION

Robot control architectures are becoming increasingly complex; prototyping and testing these systems can be time-consuming and expensive. Simulated environments like USARSim [5], [2] can facilitate the process by allowing faster and easier debugging. They can also be invaluable tools for developing and testing real-time models of embodied situated cognition. Specifically, sufficiently accurate simulations of real-robots will allow us to develop and test cognitive models effectively, without the need for the physical robot to be available throughout the process. Rather, the robot will only be needed for the final validation of the already fully developed and tested simulation model.

This paper describes work in our lab integrating USARSim into a robot development environment for the development of cognitive models, specifically to explore the effects of physical embodiment and situatedness. We will describe the details of the modeling setup using a specific neural network model of word learning that is currently under development in our lab. Using this model, we show the sequence of model development and testing in USARSim, followed by a final validation on the physical robot.

II. BACKGROUND

Cognitive modeling has a long tradition in cognitive science, going back to the late 70ies when the main architectures ACT and SOAR emerged ([1], [8]). Subsequently, various other symbolic architectures were introduced (including EPIC [9] and Prodigy [4]) as well as different kinds of neural network architectures. Typically, cognitive models developed for these architectures are only run in simulation, i.e., with simulated inputs and outputs, instead of being connected to real sensors and effectors. Yet, for studying embodied cognition, it is important to be true to the real-time real-world nature of sensory input and motor outputs. Hence, the model will either have to be run on a robot or in a physical simulation environment that can faithfully simulate the important physical aspects of sensors and actuators. Connecting a model to a real-time system (robot or simulation), however, is challenging because cognitive architectures have typically not been designed to support it.[1] What is needed is an infrastructure that can connect the inputs and outputs of the architecture with the outputs and inputs of the sensors and actuators. We will describe our ADE system that has been successfully connected to the USARSim simulation environment as well as several robots and embedded devices.

III. A BRIEF OVERVIEW OF THE ROBOT AND THE SETUP

We integrated the USARSim environment into our robot development infrastructure ADE, the *Agent Development Environment* [12]. ADE allows users to construct agent architectures using modular components (called *ADE servers*) that provide services (e.g., access to sensors or effectors) to other ADE servers, and can be distributed across multiple hosts. An *ADE registry* serves as a "yellow-pages" service for ADE servers to connect them with the other servers they require. An ADE server submits a request for a service (represented by an RMI interface) to the ADE registry, and the registry checks the credentials of the requesting server and forwards the information about the new resource. From that point on, all communication between those two servers is direct, without having to go through the registry (see [11] for more details).

As noted, ADE servers interact via pre-defined interfaces. Servers have been defined for many sensors (e.g., speech recognition, laser rangefinder, and GPS localization) as well as many effectors (e.g., robot bases such as the Pioneers and Segways, and a humanoid RoboMotio torso). The ADE USARSim server fits nicely into the system by instantiating models of many of these sensors and effectors (some predefined by USARSim, others constructed in our lab [6]) and

[1] However, note there are examples of both SOAR and ACT-R controlling robots, and, of course, neural network architectures (e.g., [15]).

replicating their interfaces, allowing other ADE servers to utilize those resources as if they were physically instantiated. The USARSim server communicates with the simulation environment via a socket connection to the GameBots engine, which allows the server to instantiate models and monitor and manipulate their states. In addition to an interface for the provided Pioneer robot and SICK laser rangefinder models, we have constructed in our lab a full model of the RoboMotio Reddy humanoid torso, including the head with manipulable eyes, eyebrows, and lips, movable head, and arms with three degrees of freedom. The model was created using a combination of Blender, Inkscape, and the Unreal editor (for details, refer to [6]).

The ADE vision server has been configured to work with a variety of cameras (including IIDC Firewire and USB cameras) and can provide information about visually detected faces, color blobs, and environmental conditions (e.g., darkness) to other ADE servers as requested. Using the vision server in the USARSim environment required us to extend the server to analyze frames from the UT virtual camera instead of a real camera. Our solution is based on the "SDL Hook for USARSim on Linux," which modifies the SDL library to intercept the frames and redirect copies of them to a socket in addition to displaying them on-screen.[2] When the vision server is directed to use USARSim instead of a camera, the only difference is in the initialization (e.g., open a socket instead of initialize a camera) and the method used to grab frames. All subsequent operations (i.e., analysis performed on the frames) is performed in the same way it would be if the frames were coming from a camera; the analysis code does not need to be modified to operate differently than it normally would. Similarly, because other ADE servers only need to know the interface exported by the vision server, whether the vision server is operating in the real world or the UT environment is completely opaque outside the vision server; other servers neither know nor care from which environment the visual information comes any more than they care what kind of camera provides the frames in a real environment.

The tests described below (see Sec. V) were conducted using the ADE vision server configured to use two "cameras." The tests conducted in the real world use a Unibrain Fire-i IIDC Firewire camera, whereas the tests in the Unreal environment use the virtual camera defined by the SDL hook. Identical vision processing was performed in each case before the results (in this case information about the color, size, and relative location of each blob detected) was passed on to the word learning model component, also encapsulated as an ADE server.

IV. MODEL DEVELOPMENT: THE WORD LEARNING MODEL

The word learning model is an associative, incremental model implemented in several interconnected artificial neural networks. The desired behavior is to learn word-reference associations in an unsupervised fashion. It has been hypothesized that human infants learn such associations in a statistical manner, tabulating co-occurrences, usually in a batch-fashion (e.g., [14]). However, it has been shown that such simple models are not sufficient to fit the human data ([13], [7], [16]). Humans are embodied agents whose experiences are inextricably situated in time and space [3]. In light of this, temporal aspects and environmental context must be taken into account in the learning model.

We introduce a learning model in which learning is *incremental*, i.e., the system is modified by every experience, and then it is the modified system which goes on to encounter further experiences. This type of model affords important qualities, perhaps most importantly the ability to learn based on co-occurrences in a non-linear fashion. In other words, the association strength learned from experiencing some $\mathbf{A}$ and $\mathbf{B}$ co-occuring two times will not be simply double the association strength of experiencing them co-occurring once.

The model can be deconstructed into its two constituent networks, one of which learns words (as phoneme-strings), and the other which learns colors. Learned types (words or colors) are represented by a set of "concept nodes". Lower-level perception is represented by a three-dimensional feature-map on the color side, and as a set of nodes which react to different phonemes on the word side. The primary learning rule for associations based on co-occurrences is a non-linear Hebbian-type rule, which can be generally stated as:

$$w_{xy}^{t} = w_{xy}^{t-1} + S \cdot \mathrm{sig}(a_x a_y) \cdot M e^{B e^{C w_{xy}^{t-1}}} - (w_{xy}^{t-1} \cdot D)$$

where S is a scalar variable[3] which scales with the magnitudes of the activations of nodes x and y, w_{xy} represents the association weight between nodes x and y, and a_x is the output activation of node x. sig is a sigmoid soft-thresholding function.[4] The doubly-exponentiated term is an application of a Gompertz function to the previous weight, and produces a horizontally asymptotic exponential growth to the weight based on the previous magnitude of the weight, but on a logrithmic scale. This is the primary factor causing the non-linear incremental learning described earlier. The coefficients B (-4) and C (-0.66) modulate the growth rate and scale of the function, and M provides the upper bound. D is a decay constant (0.02).

A. THE COLOR NETWORK

The color network is the simpler of the two. It is composed of a three-dimensional feature map (with each dimension accounting for one dimension of a color in RGB format), with each node connected to concept nodes. Color concept nodes are created when appropriate. In the case of our

[2]The SDL hook is made available by Can Kavaklıoğlu at `http://cankavaklioglu.name.tr/shul.html`.

[3]$S = \mathrm{sig}(a_x + a_y)$

[4]$\mathrm{sig}(x) = \frac{1}{1+e^{-(x-C)}}$, C is the displacement determining where the soft threshold is set, (6.0, as tested) is a parameter determining how quickly (steeply) the function grows to its asymptote.

composite system, this is when a sufficiently new word-color pair is experienced. Input comes into the color network in the form of activation given to nodes in the color map. Nodes to be activated are determined by finding the node in the map which minimizes the distance between its feature weights (i.e., R, G, B values) and the RGB values of the incoming color. Nodes surrounding the winning node then receive smaller amounts of activations based on their distance from the winner.

For these experiments, we use a feature map of size $10 \times 10 \times 10$, with each node in the feature map initialized to evenly-spaced places between 0 and 255 in each dimension. We find this an acceptable compromise between speed and space concerns, while maintaining a sufficiently fine granularity for discriminating different colors.

After a winner has been selected, activation of all other nodes is set:

$$\text{xyz CN, xyz} = e^{\frac{-d_{xyz}^2}{2^2}}$$

xyz is the activation of node with coordinates x, y, z in the feature map. is a parameter, which we have set to 0.8. CN refers to the RGB feature map. d is a *distance function* which determines the distance between the winner and the node in question, and is defined as:

$$d_{xyz} = \frac{\sqrt{(r_{xyz} - r_{win})^2 + (g_{xyz} - g_{win})^2 + (b_{xyz} - b_{win})^2}}{d_{max}}$$

where *r, g, b* are the feature value of subscripted node to that feature (*r* for red, *g* for green, *b* for blue), and d_{max} is a constant, maximum distance used to normalize the values into the desired range of activation [0, 1], which is determined by the size of the color feature network, among other things:

$$d_{max} = \frac{\sqrt{255^2}}{10} \cdot 3$$

10 is the width of our network in a given dimension (i.e., there are ten nodes), and 3 is the number of dimensions.

Energy then flows from the winning node to any connected concept nodes, modulated by the weight of that connection, i.e.,

$$\text{n WN, } {}^{t}_{n} = {}^{t-1}_{n} + \text{win} \cdot w_{win,n}$$

where, again, is the activation of the subscripted node at the superscripted time (with time as measured by the number of sounds heard since input began).

B. THE WORD NETWORK

The word network is a recurrent network, with a layer representing the reactive activations to experienced aural input, and a recurrent layer representing the activations of the previously experienced input. These two layers are connected to the (word) concept layer via an array of soft-thresholding interneuron nodes. This network effectively recognizes words based on the sequence of phonemes they contain. It does this by incrementally pooling activation into word concept nodes as phoneme-strings contained in that word-concept are experienced.

The network is assumed to know when a word ends and a word begins. Sounds enter the system as deconstructed probabilities, one for each phoneme. These are the probabilities that the uttered sound was an instance of that phoneme. Thus, if a perfect /a/ is uttered, the probability for the /a/ phoneme will be very close to one, while others will be close to zero. In the case of more ambiguous sounds, such as /b/ and /p/, it may be that each receives relatively high activation, especially if the environment is noisy.

The network is instantiated with the full score of these "phoneme nodes", one representing each salient phoneme present in the language and dialect. Input thus enters the system and induces an activation in each of those phoneme nodes. There is also an equally sized recurrent layer of phoneme-nodes, which hold the activation of the phoneme layer from the previous sound. The activation spreads along efferent links to arrays of "interneurons" (N for each word node, where N is the number of phonemes). Each phoneme node is connected to one interneuron node in each word cluster, and each recurrent phoneme is connected to *all* interneuron nodes. These interneuron nodes apply a soft threshold to their activation (using a sigmoid as in footnote 4) and pass that output on to a layer of "word nodes". The only weights which are currently trained are those of the connections between the recurrent phoneme layer (nodes notated p_o), and the interneuron layer. This weight is updated using a Hebbian rule based on three values. For a recurrent layer node p_o and an interneuron node i, the weight between p_o and i will update according to the following rule:

$$w^{t}_{p_o,i} = w^{t-1}_{p_o,i} + S \cdot \quad \cdot \text{sig}(\; i \; p_o \; p_n) - w^{t-1}_{p_o,i}$$

is a learning rate (0.1), and S is (again) a scalar to reduce undesired decay of weights when the connected nodes are not activated, calculated as:

$$S = 0.01 + \text{sig}(\; i + \; p_n + \; p_o)$$

where p_n is the normal phoneme node corresponding to the recurrent phoneme in question p_o.

Throughout a word experience, there is a rising threshold value which is applied to the activations of the word nodes. This threshold depends on the number of sounds experienced since the word began. The output activation of word nodes (as used in learning rules involving links between word nodes to other nodes) is also determined by the application of a sigmoidal soft-thresholding function based on this threshold. When a word ends, a winner is chosen from among the word nodes based on highest activation. If the activation of this winner surpasses the threshold, then the experienced sequence of sounds is considered recognized. If it does not, then the experienced sound sequence does not sufficiently match any remembered words, and so a new word node is added to the network (trained on the sound sequence).

C. THE INTEGRATED SYSTEM

We integrated the system by removing the color concept nodes and simply connecting the word nodes directly to the RGB feature map. This is justified since the color modality

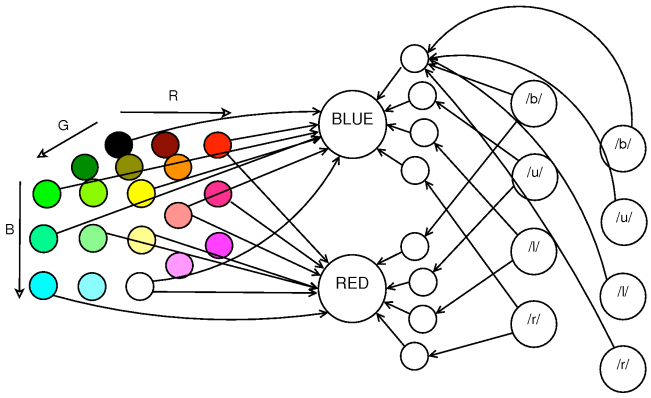


Fig. 1. The integrated word-learning system. On the left is the 3-d feature map representing red, green, blue dimensions of RGB color space. The layered network on the right is the recurrent network recognizing phoneme-sequences. Not all links are shown (only the top interneuron node of the top word node has all of its afferent connections displayed).

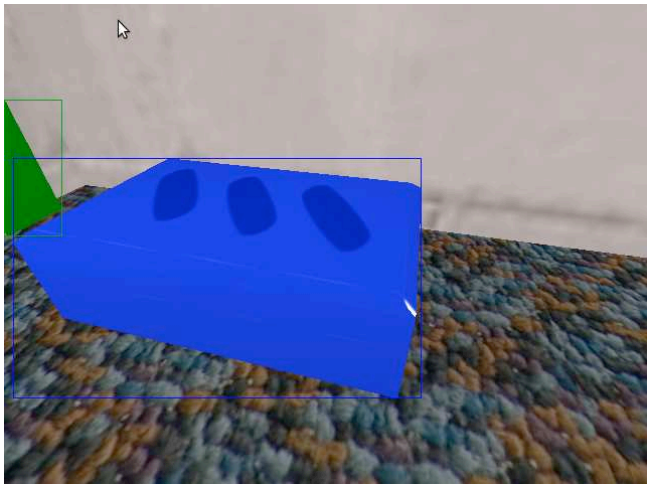


Fig. 2. Blob detector's view in the simulator of a typical bluebox situation.

is in a sense supervised by the word modality. It will only carve out an area of color space and associate it with a word when a word is presented.

The easiest way to understand how the composite system functions is to observe that the word-learning side will behave as it would alone, except that it will receive *additional* activation energy from the visual modality. This energy flows from feature map space, representing those colors which were previously experienced simultaneous to a word being experienced.

This additional information will have two interesting effects we would like to focus on. First, it will make word-recognition more robust, providing sufficient activation to a word-node if the associated color is present, even in situations where all of the contained phonemes were not sufficiently recognized to exceed the threshold for recognizing that word (a form of "perceptual co-modulation"). Second, since the learning rules contain terms representing the activations of the respective connected nodes, higher word node activations will result in higher connection weights. which translates to stronger memories. Stronger memories will be forgotten more slowly and will be easier to recall (recognize) later than memories of words which were less strongly associated with visual color experiences.

D. SETTING UP THE SYSTEM FOR EXPERIMENTS

For the testing and experiments presented in this paper, the system was set up to receive the necessary information from both its modalities in a relatively simultaneous fashion. The system is implemented as a C++ library, with functions which allow the system to be fed input (causing updates to the internal networks based on that input). To integrate this C++ library with the ADE framework, it was necessary to create an ADE server which calls the library. JNA was used to allow access to the C++ library's necessary function calls from Java.

The color map side was then updated every 150ms by querying the vision server for detected color blobs, their RGB values, and their areas. The area was normalized into the range [0,1]. This value was used to determine how much to activate a winning RGB map node. Phoneme information was fed from pre-configured files representing the output the phoneme-deconstruction program described above would produce on perfect or near-perfect input. These files were fed into the system on cue, with one "sound" fed every 150ms.

For the simulated training sessions, the agent was navigated to a block of the appropriate color and arranged so that the block took up a significant portion of the visual field (**> 55%**). In the real world training sessions a block of the appropriate color was placed in front of the robot so that it took up a similar portion of the visual field. For testing, after the system had been trained, it was shown real-world objects. To do this with a simulation-trained agent, it was necessary to disconnect the ADE server running the network from the vision server in the simulator, and reconnect to a vision server connected to the real-world robot. This was accomplished using ADE's recovery abilities–by simply killing the simulation server and having the ADE system recover by connecting to a server operating on the real-world robot.

V. MODEL VALIDATION: COMPARISON OF SIMULATION AND ROBOT DATA

Our evaluation tests are intended to examine how closely matched the behavior of the model is between the simulated environment and the real world. The learning task involved showing the robot a colored object during 10 presentations of a 4-phoneme word. Each phoneme is presented for 150ms, and there is a random (1–5s) time interval between presentations. Although the camera was allowed to move between training runs, it was held stationary while training. To prevent possible speech recognition confounds, the phoneme input was presented in perfect form so as to elicit the maximal recognition probability for each phoneme. The target object was a blue box in both the real world and USARSim. Fig. 2 shows the robot's view of the simulated box, while Fig. 3 shows a frame from a real-world training session.

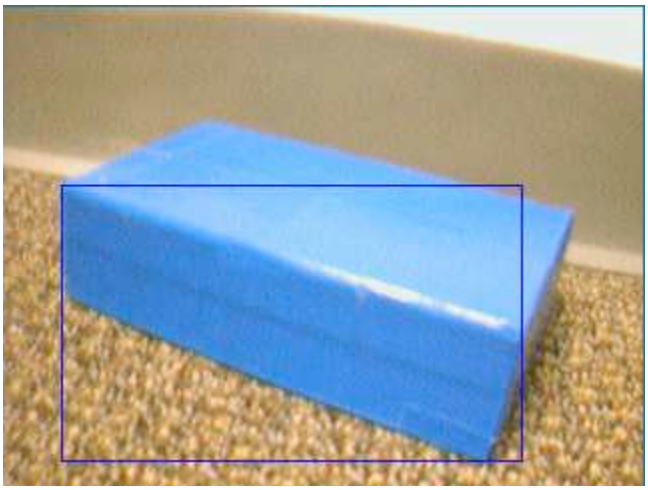


Fig. 3. Blob detector's view in the real world of a typical bluebox situation.

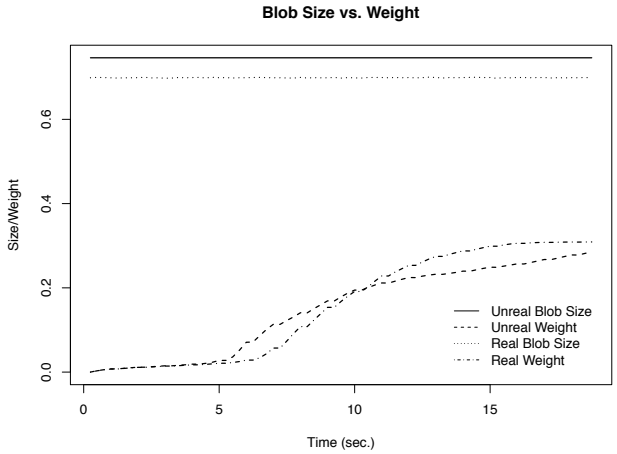


Fig. 4. Plot of blob size and color-word association weight over time

The training results are presented in Fig. 4. These results are averaged over multiple runs (16 in Unreal, 21 real-world). The results demonstrate that the learning behavior of the system is comparable in the simulator and in the real world. Because the camera is held stationary, there is little change in the average blob size as the training sessions progress. However, note the horizontally asymptotic exponential growth behavior of the weight between the color and the associated word node, caused by the learning rule proposed above. The regular "dips" in association strength are actually an effect of the implementation of word-recognition in the system. They occur because the initial phoneme of a word will not cause any major activation in a word node (since it is not a sequence). Thus, at these times, the word and color may actually be *de*-associated, but only by a small amount.

Regarding the real-virtual comparison, the growth of the weights is very strongly correlated in the two environments (Pearson's $\mathbf{r = .982, p < .001}$). There are minor differences in the overall pattern of growth, caused by the difference in average blob size and a greater blob size variance in the simulated environment. However, the learning is clearly very similar in the two environments. Moreover, as an additional validation, we tested the "recognition" ability of the trained systems in a real-world test. That is, we performed the training as described above, then after a short interval (2s) to allow the system to settle (due to decay), we presented input from the *real* camera to both those trained in the real world and those trained in USARSim. When presented with the blue blob input, without any phoneme input, real-world trained systems' word activation rose from an average of .0001 to .88, while the USARSim trained systems' average activation rose from .0001 to .77. That is, the activation of the word associated with the color increased substantially in virtue of being presented with that color. The increases are quite similar, as confirmed by the lack of a significant difference found by the t-test ($\mathbf{t = -1.1346, p = 0.2677}$).

It was difficult to get the blob sizes in the real world versus the simulated tests to match up exactly, and this explains some of the small differences in the growth rate data. In the controlled situations used in the experiments, and with only pertinent aspects of the network extracted and reflected in the results presented above, the effects of noise in the blob-detection algorithm are not obvious. However, this is justified. More so than visual noise or failures of the blob-detection algorithm (which would often result in additional, superfluous blobs being detected), it was differences in the size of the blob that adversely affected the results. Since the blob size has a doubly-exponential effect on the learning rate of the system, we see an odd comparison in our data. On the simulated side, blob size ranged over a larger variance of values then in the real world, where control over the positioning of the camera is not so coarse. This variance resulted in a more awkwardly-shaped graph (i.e., differently shaped than the curve that would be generated by the learning algorithm discussed above) for the simulated side, since functions with drastically different growths and shapes are being averaged together with no account for the blob size, an exponent. These small differences sometimes had the effect of the system falling on two sides of a bifurcation, resulting in relatively close blob sizes causing two drastically different learning curves–one which grew to maximum within the allotted time, and another which did not even come close to doing so.

It is also important to recognize that the differences observed in our results are *not* due to noise in the real world and the lack thereof in the simulated world. Even if the simulated image was somehow intentionally degredated to account for noise, the effect would be more failures of the blob detection algorithm in ambiguous or noisy visual situations, which we endeavored to avoid in the experimental setup presented in this paper and which are not reflected in the results. It is entirely possible, however, that given a different experiment (for example, one that examined the robustness of recognition, as is discussed below) the differences in noise in the real world versus the simulated one would be salient in explaining the differences.

VI. CONCLUSION AND FUTURE WORK

This paper presented our integration of a vision-based word learning architecture with the USARSim simulation environment. The existing vision component of the ADE robotic architecture infrastructure was extended to allow input from Unreal, allowing us to use the simulated environment as a drop-in replacement for real-world testing. We demonstrated the validity of the integration by comparing tests of the learning mechanism conducted using the simulator with tests conducted in the real world. The results showed that the training progressed comparably in the two environments. Moreover, on the real world color recognition task, the system performed similarly regardless of whether it was trained in the simulated environment or in the real world.

Future work will involve further development of the word learning system presented in this paper in multiple directions. For these experiments, parameter values and even function types were chosen using trial-and-error. In future work, a genetic algorithm may be used to optimize these parameters for the task at hand. The complexity of the word-learning side of the network leaves a lot to be desired, only recognizing contained phoneme-sequences. Modifications to the network will also be performed to allow for other effects on word-recognition, such as a phoneme being present in a word. Eventually, we hope to match the word-learning behavior of the system to data from real-world children's word-learning behaviors. Observations from that domain may give important clues towards optimizing the parameters and learning functions to match that behavior. Finally, an exciting next step will be to move from only recognition to also initiating action. Hearing a word or seeing a color could actually cause the agent to respond, perhaps by refocusing its attention. We expect this to result in an interesting dynamical feedback loop: as attentional focus on an object increases its proportion in the visual field, activations for that will increase, causing faster learning to take place. The fast learning would in turn cause higher activations the next time around, resulting in more probable, stronger attention shift towards that object. Research suggests that children may also learn words in this fashion [10].

VII. ACKNOWLEDGMENTS

This work was in part funded by ONR MURI grant #N00014-07-1-1049 to second author. Thanks to You-Wei Cheah for his work on the USARSim integration into the ADE vision server.

REFERENCES

[1] J.R. Anderson, D. Bothell, M.D. Byrne, and C. Lebiere. An integrated theory of the mind. *Psychological Review*, 11:1036–1060, 2004.

[2] Steven Balakirsky, Chris Scrapper, Stefano Carpin, and Michael Lewis. Usarsim: Providing a framework for multi-robot performance evaluation. In *Proceedings of PerMIS*, 2006.

[3] Randall Beer. Dynamical approaches to cognitive science. *Trends in Cognitive Science*, 4(3):91–99, 2000.

[4] Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton, and Manuela Veloso. Prodigy: an integrated architecture for planning and learning. *SIGART Bull.*, 2(4):51–55, 1991.

[5] Stefano Carpin, Michael Lewis, Jijun Wang, Steven Balakirsky, and Chris Scrapper. Usarsim: A robot simulator for research and education. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-2007)*, pages 1400–1405, April 2007.

[6] Kyle Carter, Matthias Scheutz, and Paul Schermerhorn. A humanoid-robotic replica in usarsim for hri experiments. In *IROS Workshop on Robots, Games, and Research*, St. Louis, MO, October 2009.

[7] George Kachergis, Chen Yu, and Richard Shiffrin. Temporal contiguity in cross-situation statistical learning. In *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, 2009.

[8] John Laird, Allen Newell, and Paul Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[9] D.E. Meyer and D.E. Kieras. A computational theory of executive cognitive processes and multiple-task performance. part 1. *Psychological Review*, 104(1):3–65, 1997.

[10] Larissa Samuelson and Linda B Smith. Memory and attention make smart word learning: An alternative account of akhtar, carpenter and tomasello. *Child Development*, 69:94–104, 1998.

[11] Paul Schermerhorn and Matthias Scheutz. Natural language interactions in distributed networks of smart devices. *International Journal of Semantic Computing*, 2(4):503–524, 2008.

[12] Matthias Scheutz. ADE - steps towards a distributed development and runtime environment for complex robotic agent architectures. *Applied Artificial Intelligence*, 20(4-5):275–304, 2006.

[13] Richard Shiffrin and Mark Steyvers. A model for recognition memory: REM–retrieving effectively from memory. *Psychonomic Bulletin and Review*, 4(2):145–166, 1997.

[14] Joshua B. Tenenbaum and Fei Xu. Word learning as bayesian inference. In *Proceedings of the 22nd Annual Conference of the Cognitive Science Society*, 2000.

[15] J. Trafton, N. Cassimatis, M. Bugajska, D. Brock, F. Mintz, and A. Schultz. Enabling effective human-robot interaction using perspective-taking in robots. *IEEE Transactions on Systems, Man and Cybernetics*, 25(4):460–470, 2005.

[16] Chen Yu and Linda B. Smith. Rapid word learning under uncertainty via cross-situational statistics. *Psychological Science*, 18:414–420, 2007.

Integrating Automated Object Detection into Mapping in USARSim

Helen Flynn, Julian de Hoog and Stephen Cameron
Oxford University Computing Laboratory
July 2009

Abstract— **Object recognition is a well studied field of computer vision, and has been applied with success to a variety of robotics applications. However, little research has been done towards applying pattern recognition techniques to robotic search and rescue. This paper describes the development of an object recognition system for robotic search and rescue within the USARSim simulator, based on the algorithm of Viola and Jones. After an introduction to the specifics of the object recognition method used, we give a general overview of how we integrate our real-time object recognition into our controller software. Work so far has focused on victims' heads (frontal and profile views) as well as common objects such as chairs and plants. We compare the results of our detection system with those of USARSim's existing simulated victim sensor, and discuss the relevance to real search and rescue robot systems.**

I. INTRODUCTION

The primary goal of robotic search and rescue is to explore a disaster zone and provide as much information as possible on the location and status of survivors. While the development of advanced and robust robot platforms and systems is essential, high-level problems such as mapping, exploration and multi-agent coordination must be solved as well. Development of such high-level techniques is the goal of RoboCup's Virtual Robots competition. This competition uses USARSim as a basis for its simulations due to this simulator's high quality image data and rendering.

Since the primary goal of robotic search and rescue is finding victims, a simulated robot rescue team must be able to complete this task in simulation. In real rescue systems, identification of victims is often performed by human operators watching camera feedback. To lower the burden on teams using USARSim for rescue systems research, a 'VictimSensor' has been developed for the simulator that mimics recognition of victims in real systems [1]. Modeled after template based human form detection, this sensor must be associated with a camera and performs line-of-sight calculations to the victim. It starts reporting victims at a distance of about 6 metres, and its accuracy improves with increased proximity. However, in this paper we report on work-in-progress towards providing a fast vision-based detector as an alternative, based on the work of Viola and Jones. The hope is that this will provide a more realistic simulation of real-world victim detection, and bring USARSim-related research one step closer to reality.

A secondary goal of search and rescue efforts is to produce high quality maps. One of the reasons to generate a map is to convey information, and this information is often represented as attributes on the map. In addition to victim information, useful maps contain information on the location of obstacles or landmarks, as well as the paths that the individual robots took.

With a view to improving map quality, we have developed a system for the automated labeling of recognisable items in USARSim. In robotics, there is often a need for a system that can locate objects in the environment – we refer to this as 'object detection'. Our detection system exploits the high quality image data from USARSim which allows for accurate classifiers to be trained with a relatively low false positive rate. Using images from USARSim as training data, we have trained various different classifiers to detect various objects, including victims.

The paper is structured as follows. Section II describes related work in object recognition and mapping. Section III provides an overview of our system, including the object detection method used and the training process. In Section IV we detail the process of integrating object detection and mapping into our controller software. In Section V we present preliminary results. Several possible extensions to our object detection systems exist. Some of these are detailed in Section VI followed by concluding remarks in Section VII.

II. RELATED WORK

Object recognition is a well-studied field of computer vision. Swain and Ballard [2] first proposed colour histograms as an early view-based approach to object recognition. This idea was further developed by Schiele and Crowley [3] who recognised objects using histograms of filtered responses. In [4], Linde and Lindeberg evaluated more complex descriptor combinations, forming histograms of up to 14 dimensions. Although these methods are robust to changes in rotation, position and deformation, they cannot cope with recognition in a cluttered scene.

The issue of where in an image to measure has an impact on the success of object recognition, and thus the need for 'object detection'. Global histograms do not work well for complex scenes. Schneiderman and Kanade [5] were among the first to address object categorisation in natural scenes, by computing histograms of wavelet coefficients over localised object parts. In a similar approach, the popular SIFT (scale-invariant feature transform) descriptor [6] uses position-dependent histograms computed in the neighbourhood of selected image points.

In recent years there has been increasing interest in using object detection in SLAM (simultaneous localisation and mapping) to provide information additional to that provided

by laser scans. Such an approach is denoted as visual SLAM (vSLAM). Cameras have an advantage over lasers in that they can offer higher amounts of information and are less expensive. Different methods have been used to extract visual landmarks from camera images. Lemaire and Lacroix [7] use segments as landmarks together with an Extended Kalman Filter-based SLAM approach. Frintrop *et al.* [8] extract regions of interest using the attentional system VOCUS. Others [9] have used SIFT descriptors as landmarks; Se *et al.* [10] and Gil *et al.* [11] track the SIFT features in successive frames to identify the more robust ones; Valls Miro *et al.* [12] use SIFT to map large environments. Davison and Murray [13], and Hygounenc *et al.* [14] use Harris Point detectors as landmarks in monocular SLAM. Finally, Murillo *et al.* [15] propose a localisation method using SURF keypoints.

Jensfelt *et al.* [16] integrate SLAM and object detection into a service robot framework. In their system, the SLAM process is augmented with a histogram based object recognition system that detects specific objects in the environment and puts them in the map generated by the SLAM system. Later the robot is able to assist a human when he/she wants to know where a particular object is. This situation is concerned with the problem of detecting a *specific* object as opposed to a general category of objects.

To the authors' knowledge, little work has been done on integrating object detection techniques into high fidelity simulation applications such as USARSim. For the 2007 Virtual Robot Competition Visser *et al.* [17] used a colour histogram approach for victim detection. A 3D colour histogram is constructed in which discrete probability distributions are learned. Given skin and non-skin histograms based on training sets it is possible to compute the probability that a given colour belongs to the skin and non-skin classes. A drawback of this approach is that in unstructured environments there is no a priori data on the colours present in the environment, which could result in a large number of false positives. In this paper we focus on a more advanced method of detecting general classes of objects in USARSim, and putting them into the environmental map as the exploration effort unfolds.

III. SYSTEM OVERVIEW

Viola/Jones Algorithm: The method we use is based on Viola and Jones' original algorithm for face detection [18], which is the first object detection framework to provide accurate object detection rates in real time. Used in real-time applications, the original detector ran at 15 frames per second on year 2000 hardware; it is therefore suitable for object recognition in robot simulation. We also use this method because it is suitable for detecting objects in complex scenes and under varying lighting conditions, which is typical of robot rescue scenarios.

The method uses a variant of the AdaBoost algorithm for machine learning which generates strong decision tree classifiers from many weak ones. The weak learners are based on features of three kinds, all of which can be individually computed quickly at frame rates. However for a 24×24 pixel sub-window there are more than 180,000 potential features. The task of the AdaBoost algorithm is to pick a few hundred of these features and assign weights to each using a set of training images. Object detection is then reduced to computing the weighted sum of the chosen rectangle features and applying a threshold. Thus, although training of the classifiers takes *a lot* of computer time, the resultant classifiers can be run very quickly.

Cascade of Boosted Classifiers: We adopt a fast approach used in [19] where we cascade many such detectors, with more complex detectors following simpler ones. Input (an image from a robot's camera) is passed to the first detector which decides true or false (victim or not victim, for example). A false determination halts further computation; otherwise the input is passed along to the next classifier in the cascade. If all classifiers vote true then the input is classified as a true example. A cascade architecture is very efficient because the classifiers with the fewest features are placed at the beginning of the cascade, minimising the total computation time required.

Training classifiers: For each classifier, our training set consisted of several thousand 400×300 images taken from many different USARSim worlds. In the positive examples (i.e. images containing the object of interest), the object was manually tagged with a bounding box and the location recorded in an annotation file. Ideally each bounding box should have the same scale. In addition to improving classification accuracy, this makes it easier to estimate the real world location of detected objects. For faces we used square bounding boxes.

A major issue with accurate object detection is the effect of different viewpoints on how an object looks. For this reason our training set contained images of objects from many different angles.

Training was carried out over several weeks using the popular open source computer vision library OpenCV[1]. To greatly speed up the process we employed the use of a 528 core SGI-ICE cluster at Oxford University's Supercomputing Centre (OSC), as described in the appendix. In the initial phase separate classifiers were trained for frontal and profile views of heads, both at close range and at further distances. The larger the training set the better; in particular it helped to have a very large number of negative examples. The finished detectors contain a few thousand nodes each, with each classifier having depth of a few tens of nodes. For comparison, classifiers were also then trained for common obstacles found in USARSim environments, such as chairs and potted plants.

We found that the false positive rate can be reduced by using images that were misclassified in one stage as negative examples for successive stages. Our initial face detectors were incorrectly identifying wheels of cars quite often. When we used images of car wheels as negative examples in the training set, the rate of false alarm went down. An example of successful detections, along with one false positive, are shown in Figure 1. (To date the rate of false positives seems acceptable to us for a Robocup Rescue scenario, given that they would presumably then be screened by the operator; however the system is yet to be tested under competition conditions.)

[1] http://sourceforge.net/projects/opencvlibrary/files/

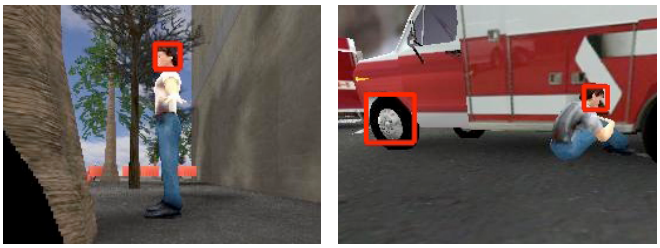


Fig. 1. Our face detector can recognise faces at greater distances than the VictimSensor. Some detections are false positives, but these can be used as negative examples in subsequent levels of training.

IV. INTEGRATION OF OBJECT DETECTION & MAPPING

A key competence for a mobile robot is the ability to build a map of the environment from sensor data. For SLAM we use a method developed by Pfingsthorn *et al.* [20], inspired by the manifold data structure [21], which combines grid-based and topological representations. The method produces highly detailed maps without sacrificing scalability. In this section we describe how we augment our map with the location of objects.

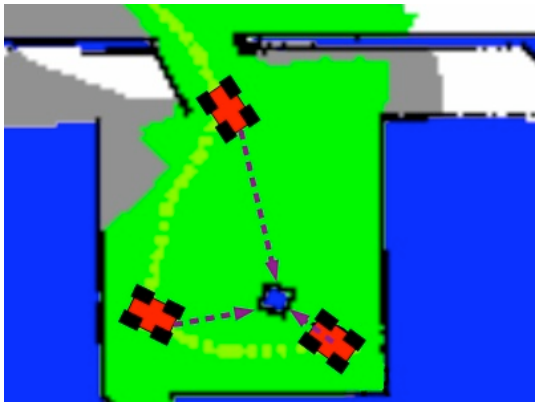


Fig. 2. An object has been detected by the robot from three different locations. The position estimate can be improved by triangulation.

Finding the position of objects: Each detector is defined within a single small XML file which is read by a robot's camera sensor; the detector is scanned across the image at multiple scales and locations, using code from the OpenCV library. When an object is detected, the image is saved together with the location of the object in the image and the robot's current pose. The location is taken to be centre of the bounding box. Using the size of an object's bounding box as a gauge of its distance from the robot and its angle relative to the robot, an accurate position estimate is calculated. If the same object is detected from several camera locations the position estimate can be improved by triangulation (Figure 2). This is then placed in the map. An annotated example of a final map is shown in Figure 3.

A position estimate can be quite inaccurate if the bounding box does not fit the object's boundaries closely, since our calculations are based on the real world dimensions of an object.

Multi-robot object detection: Using USARSim's Multi-View we can extend our object detection system to multiple robots exploring the environment simultaneously. In this way each robot has its own object detection capability. Ideally the resolution of each subview should be no lower than that of the training images.

Re-detection of objects: If a newly detected object is within suitably close range of an existing object already detected, this suggests that it is the same object. The position estimate is made more accurate using triangulation. Re-detection is key to the accuracy of our position estimates. Using triangulation, our position estimates are generally within 1 metre of ground truth. Moreover, re-detection helps us to deal with false positives. Detections occurring only within one frame are likely to be false positives, whereas repeated detections in multiple frames increase the confidence that the detection is correct.

V. RESULTS

Our object detection system is a work in progress. However, initial results are encouraging, and providing false alarm rates can be reduced, object detection shows promise for use in both high fidelity simulators like USARSim and real robot rescue systems.

Classification accuracy: We tested our detector in several standard USARSim worlds, including the CompWorld and Reactor environments, using multi-robot teams. Figure 4 shows ROC curves for each of our classifiers, and Figure 5 shows some other examples of faces that have been correctly identified by our system, even though they were not detected by the simulated VictimSensor.

Results for faces and plants have detection rates of more than 80% (for some more examples, see Figure 6). However, false alarm rates increase with detection rates. Given our experience now with training for faces on the supercomputer we intend to soon re-visit the issue of training for other objects.

Fig. 5. Four other successful detections of faces.

Our results for hands are less impressive than those for faces and plants. We surmise that there are two reasons for this: firstly, hands come in a wide range of varying poses,

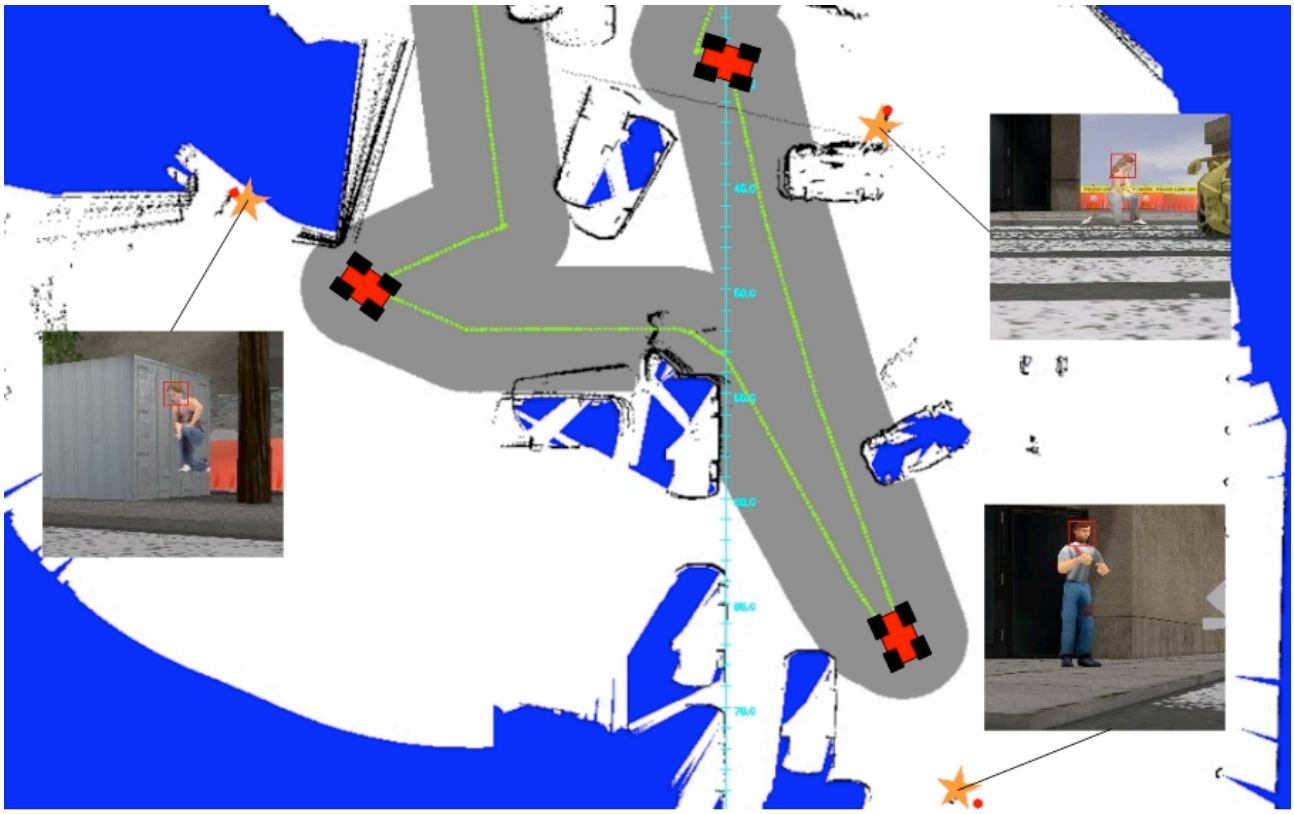


Fig. 3. A completed map together with automatically saved images of detected objects. Green line is the robot's path, orange stars are position estimates and red dots are actual positions.

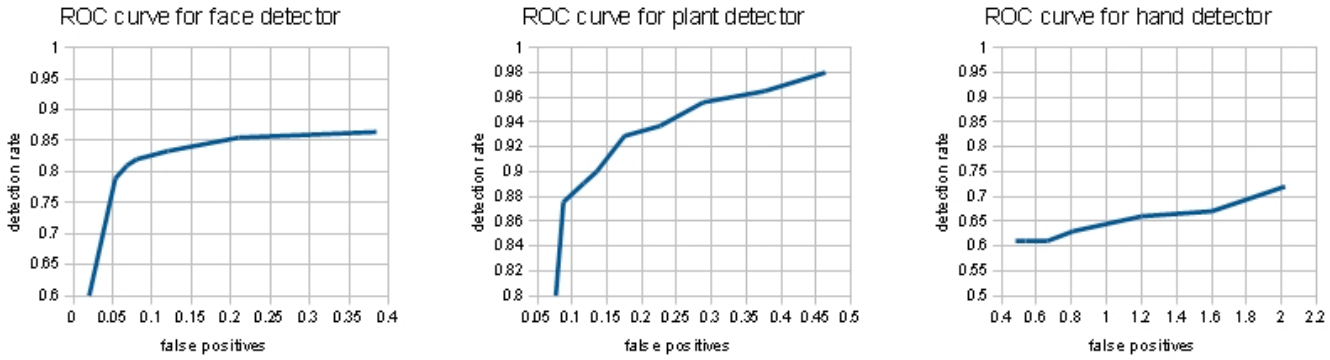


Fig. 4. ROC Curves showing the performance of our object detection system.

from outstretched to clenched fists to hands that are waving. It is therefore difficult to extract the salient features. Faces, conversely, exhibit common features which can easily be learned by a classifier. Plants, particularly the generic potted plants in USARSim, tend to be (vertically) rotation invariant and have the same general characteristics. Secondly, our hand classifier was one of the first classifiers to be trained, before we had access to the supercomputer, so we didn't use as large a training set as would have been optimal.

Detection time: To save computation time our detection module searches for objects every n (say, $2 \leq n \leq 4$) frames. The lower n is the more likely an object is to be detected quickly during fast robot motion. Computation time is also dependent on the number of classifiers being passed over a frame. Using an Image Server resolution of 400×300, objects are detected in a few tens of milliseconds on a 2.4 GHz Intel Core 2 processor.

Real images: To evaluate the usefulness of our vision-based classifier as a simulation tool, we ran some real images through it. While some faces are classified perfectly, false positive rates significantly increased (for some examples, see Figure 7). This was to be expected, given that real faces exhibit a much higher degree of detail and a much wider variety of features than those used to train our classifier. However, the same classifier has been trained with greater success on real faces by Viola and Jones [19], and the

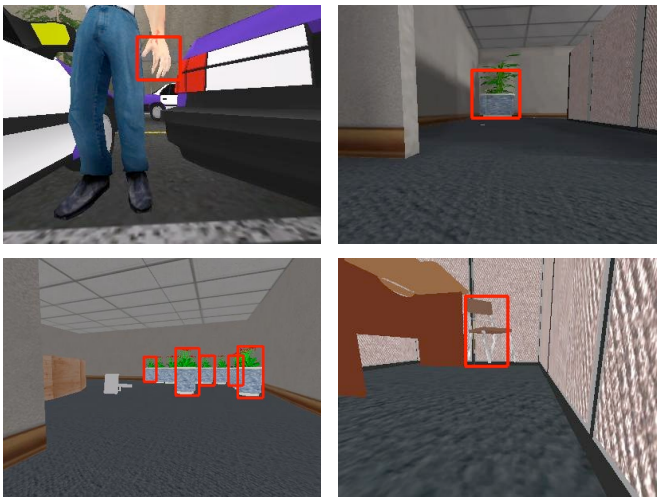


Fig. 6. Examples of other objects detected: hand, plants, chair.

training process for classifiers of real rescue robot systems would not differ from the training process we used within USARSim. In fact, our classifier correctly identifies faces in approximately 88% of simulation images, while Viola and Jones' classifier correctly identifies faces in approximately 90% of real images. Consequently we believe that it is a valid simulation tool: vision-based automated object recognition in real rescue systems would provide similar data and need to be integrated in a similar way to our simulation-based classifier.

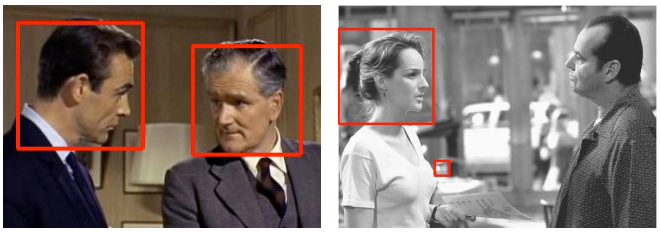


Fig. 7. Results of running real images through our trained classifier. Some faces are recognised well, but the rate of false positives and undetected faces increases significantly.

VI. FURTHER WORK

Several extensions to our object detection system could lead to further improvements in victim detection and map quality. Some of these are detailed here.

Improved detection range and detection of more object types: We hope soon to train classifiers that can detect faces at further distances than at present, using higher resolution images from USARSim. We further hope to train the classifier on a wider range of objects.

Eliminate need for the simulated VictimSensor: Currently our face classifiers work for upright faces only. Since the primary goal of robotic search and rescue is to find victims, we plan to extend our victim detection system to victims in differing poses, such as those that are lying down. We hope also to train classifiers for other body parts so as to eliminate reliance on the VictimSensor. If our vision-based victim sensor proves to be very reliable, we envision eventually integrating it either into the simulator itself or into the image server, so that the classifier may be available to the wider USARSim community.

Tracking by AirRobots: Since AirRobots are increasingly being used successfully in exploration efforts, most recently in RoboCup 2009 where our team made extensive use of AirRobots in various tests, we plan to use our object recognition system to enable AirRobots to track objects on the ground. This can in turn be used to improve mutual localisation amongst ground robots within the team.

Object recognition using non-standard views: A recent addition to USARSim has been the catadioptric omnidirectional camera which provides a full 360 degree view of a robot's surroundings. Additionally, the Kenaf robot platform uses a fish-eye lens to provide a top-down view of the robot. We are interested in investigating whether object recognition can be applied to such non-standard image data using an internal representation of a given object, or using a separate classifier.

Dust and Smoke: Real disaster scenes are likely to be subject to dust and smoke; it would be interesting to evaluate our system in the presence of such visual clutter.

VII. CONCLUSIONS

We have developed a recognition system for faces and common obstacles in disaster zones exploiting USARSim's highly realistic visual rendering. Although the algorithms that we have used are not new, our main contribution is the integration of existing algorithms within a full robot rescue system. One novel feature of our work is the use of a super-computer to train the detectors; without that, the results reported here would not have been achieved.

Our victim detection rivals USARSim's simulated VictimSensor, both in terms of the number of victims found and the distance at which victims may be identified. Detectors for obstacles such as furniture and potted plants allow us to produce richer maps which give a clearer view of an environment's topography. Since our object detectors consist of single XML files generated using open source libraries, they can easily be integrated into any application interface to USARSim.

For chairs and hands the results were less impressive than for faces and plants; chairs are difficult to recognise because their shapes are complex and they are characterised by thin stick-like components, and hands are even more difficult to recognise because there are so many different gestures. However with our recent experience in developing classifiers with the help of the supercomputer cluster we hope to re-visit such items to improve our current classifiers for them.

Our classifier does not perform as impressively on real-world images. This makes sense however, given that it has been trained on simulator images. Similar real-world classifiers for real robot rescue systems could be trained in the same way, as has already been performed by Viola and Jones [19]. Consequently any future research in USARSim that draws conclusions based on a simulated vision-based classifier such as ours is relevant to real-world systems.

In addition, many modern camera systems have the facility to run small pieces of code on the raw image near the camera, and to only then send the results to the main processor. Given

the small code size and simplicity of our classifier, it could be run on the camera itself. The object detection results would then be available to any system using the camera. Since this is a realistic possibility in reality, we envision extending USARSim to behave in a similar manner: an object detection system could be built into the image server or the simulator itself, and the detection results would be available to the wider USARSim community.

VIII. ACKNOWLEDGMENTS

The authors gratefully acknowledge the use of the Oxford SuperComputing Centre cluster, without which the classifiers used could not have been generated in a reasonable time.

REFERENCES

[1] S. Balakirsky, C. Scrapper, and S. Carpin. The evolution of performance metrics in the robocup rescue virtual robot competition. In *The Evolution of Performance Metrics in the RoboCup Rescue Virtual Robot Competition*, 2007.

[2] M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, Jan 1991.

[3] B. Schiele and J. L. Crowley. Recognition without correspondence using multidimensional receptive field histograms. *International Journal of Computer Vision*, 36:31–50, 2000.

[4] O. Linde and T. Lindeberg. Object recognition using composed receptive field histograms of higher dimensionality. *17th International Conference on Pattern Recognition, 2004*, 2:1 – 6 Vol.2, Jul 2004.

[5] H. Schneiderman and T. Kanade. A statistical method for 3d object detection applied to faces and cars. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1:1746, 2000.

[6] D. Lowe. Object recognition from local scale-invariant features. *International Conference on Computer Vision*, Jan 1999.

[7] T. Lemaire and S. Lacroix. Monocular-vision based slam using line segments. *2007 IEEE International Conference on Robotics and Intelligent Systems*, Jan 2007.

[8] S. Frintrop, P. Jensfelt, and H. Christensen. Attentional landmark selection for visual slam. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2582 – 2587, Sep 2006.

[9] S. Se, D. Lowe, and J. Little. Vision-based mobile robot localization and mapping using scale-invariant features. *2001 IEEE International Conference on Robotics and Automation*, 2:2051 – 2058 vol.2, Jan 2001.

[10] S. Se, D. Lowe, and J. Little. Global localization using distinctive visual features. *International Conference on Intelligent Robots and Systems*, Jan 2002.

[11] A. Gil, O. Reinoso, W. Burgard, C. Stachniss, and O. Martinez Mozos. Improving data association in rao-blackwellized visual slam. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2076–2081, Beijing, China, 2006.

[12] J. Valls Miro, W. Zhou, and G. Dissanayake. Towards vision based navigation in large indoor environments. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2096 – 2102, Sep 2006.

[13] A. Davison and D. Murray. Simultaneous localization and mapbuilding using active vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Jan 2002.

[14] E. Hygounenc, I. Jung, P. Soueres, and S. Lacroix. The autonomous blimp project of laas-cnrs: Achievements in flight control and terrain mapping. *International Journal of Robotics Research*, Jan 2004.

[15] A. Murillo, J. Guerrero, and C. Sagues. Surf features for efficient robot localization with omnidirectional images. *2007 IEEE International Conference on Robotics and Automation*, pages 3901 – 3907, Mar 2007.

[16] P. Jensfelt, S. Ekvall, D. Kragic, and D. Aarno. Augmenting slam with object detection in a service robot framework. *15th IEEE International Symposium on Robot and Human Interactive Communication, 2006*, pages 741 – 746, Aug 2006.

[17] A. Visser, B. Slamet, T. Schmits, L. A. Gonzlez Jaime, and A. Ethembabaoglu. Design decisions of the UvA Rescue 2007 team on the challenges of the virtual robot competition. In *Fourth International Workshop on Synthetic Simulation and Robotics to Mitigate Earthquake Disaster*, pages 20–26, Atlanta, July 2007.

[18] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Jan 2001.

[19] P. Viola and M. Jones. Robust real-time face detection. *International Journal of Computer Vision*, Jan 2004.

[20] M. Pfingsthorn, B. Slamet, and A. Visser. A scalable hybrid multi-robot slam method for highly detailed maps. *Lecture Notes in Computer Science*, Jan 2008.

[21] A. Howard. Multi-robot mapping using manifold representations. *2004 IEEE International Conference on Robotics and Automation*, 4:4198 – 4203 Vol.4, Jan 2004.

APPENDIX

Training a classifier to detect objects in USARSim

Collect images: A collection of both positive and negative examples is required. Positive examples contain the object of interest, whereas negative examples do not. In the positive examples, rectangles must mark out the object of interest, and these rectangles should be as close as possible to the object's boundaries. Ideally, the bounding boxes in every image should have the same scale.

Four sets of images of required: a positive set containing the object of interest; another positive set for testing purposes; a negative (or 'backgrounds') set for training; and a negative set for testing. The test sets should not contain any images that were used for training. Several thousand images are required, both for positive and negative samples. For frontal faces we used 1500 positive training images, 100 positive testing images, 5000 negative training images and 100 negative testing images.

Create samples: OpenCV's `CreateSamples` function can be used to create positive training samples. If there are not sufficient images for training (several thousand are required) additional images may be created by distorting existing images. However, the wider the range of reflections, illuminations and backgrounds, the better the classifier is likely to be trained.

The `CreateSamples` function generates a compressed file which contains all positive images. Assuming a sample size of 20x20 is suitable for most objects, samples are reduced to this size. We experimented with larger sizes, but there was not any noticeable improvement.

Training: OpenCV's `HaarTraining` function may be used to train the classifiers. Custom parameters include minimum hit rate, maximum false alarm, type of boosting algorithm and the number of stages. OpenCV developers recommend that at least 20 stages are required for a classifier to be usable. We obtained the best results using 30 stages and the default values for the other parameters.

For our training, we used an Oxford Supercomputing Centre cluster having 66 nodes, each having 8 processors (2 quad core Intel Xeon 2.8GHz) and memory 16GiB DDR2. An essential advantage of this cluster was its parallel processing capability, which allowed for the classifiers to be trained in a reasonable time. Training took approximately 72 hours for each classifier (for comparison, we estimate that the same task on a single PC would take over a month).

Performance evaluation: Performance of a classifier can be measured using OpenCV's performance utility. This evaluates the entire testing set and returns the number of correct detections, missed detections and false positives.

3D Mapping: testing algorithms and discovering new ideas with USARSim

Paloma de la Puente, Alberto Valero, Diego Rodriguez-Losada

Abstract— **It is usually not very convenient to use real robots for software development, testing and evaluation. Having access to a good simulator allows researchers to recreate special experimental conditions and gain increased flexibility and reliability, saving a lot of time. In the 3D mapping context specific issues arise. Here we describe and compare our experiences in this area with a real robot and with the Unified System for Automation and Robot Simulation (USARSim).**

I. INTRODUCTION

Most of the problems that direct experiments with real robots pose are well known and recognized. Set up of equipment and hardware configuration changes are time consuming and not always possible. Environmental conditions may not be adequate at every moment, neither is it easy to arrange real scenarios to be like we would want them to. In addition, running out of battery in the middle of an experiment may require to come back to a starting position perhaps far away located. These are just a few examples, not to talk about the risk it implies testing new software on a mobile, probably expensive, platform. Hence the important role simulators play in the development and validation of algorithms and techniques in the research field [1].

In contrast with some existing robotic simulators which only offer 2D virtual worlds, like MobileSim [2], by ActiveMedia Robotics [3], or Stage, belonging to the Player/Stage/Gazebo project [4], USARSim [5] provides a wide variety of 3D highly realistic models of different environments, more than 23 robotic comercial platforms and more than 15 sensors [6]. The available world models can be edited to generate new desired ones. USARSim also approximates kinematics and dynamics with a good precision. Furthermore, the user can easily see that perceptual fidelity for a better human-robot interaction (HRI) is one of the most outstanding features that USARSim presents.

Recently, we have been working with USARSim with two main purposes. At first, we wanted to collect data to obtain 3D point clouds associated to odometry data as the robot operated in a stop-scan-go manner. Even though we do have a 3D mapper real robot working, for the afore-mentioned reasons this allowed us to speed up our 3D data gathering processes and increment the number and variety of data sets we could use to test and tune our data processing and mapping algorithms [7]. Afterwards, we got involved in the RoboCup'09 Search and Rescue Virtual Robots Competition [8], [9]. For the integration of the 3D mapping components [10] and aiming to get a suitable functional solution according to the competition's goals and conditions, different modifications and adaptations were incorporated and a new robot configuration was designed.

This work was partially funded by the Spanish Ministry of Science and Innovation, DPI2007-66846- c02-01

P. de la Puente, A. Valero and D. Rodriguez-Losada are with the Intelligent Control Group, Universidad Politecnica de Madrid, 28006 Madrid, Spain

Fig. 1. Left:3D data acquisition system, mounted on our robot Nemo.

The paper is organized as follows. Section 2 is a description of our P3AT robot, Nemo, and its 3D laser sensor, along with a brief report on how we operate with it. Section 3 is about our simulated Nemo P2AT, emphasizing its similarities and differences with the real robot. Section 4 presents ATRVJr3D, the final robot configuration we chose for the competition. In Section 5 we outline some of the techniques we developed for the competition. Section 6 presents some experiments and analyzes the performance of our models. The paper finishes with our conclusions.

II. THE REAL ROBOT, NEMO

Our real robot is a Pioneer 3AT by MobileRobots.Inc [11], we call it Nemo (Fig. 1). At its front we have mounted a SICK LMS200 laser sensor on top of a servo pan-tilt unit. This device's model is PowerCube Wrist 070 (PW70), by Amtec Robotics [12]. The wrist requires 24V cc, which are obtained by means of a dc/dc converter from 12 to 24V. Both the laser and the wrist are connected to a mini laptop onboard the robot with USB to RS-232/422/485 adapters. A data server processes the data and sends synchronized updated information about odometry, PW70 and laser measurements at clients' cyclical requests. The port's baud rate for the laser scanner is set at 500 kb so as to gain velocity and allow for a good precision in the synchronization with the PW70. This is of utmost importance to avoid distortions when applying the relative transformations to calculate each point's cartesian 3D coordinates.

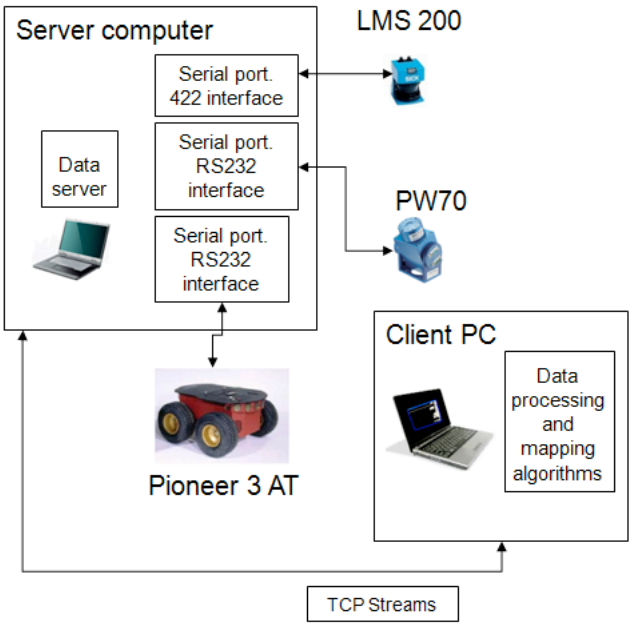


Fig. 2. Distribution schema for our work with the real robot

Fig. 2 depicts the general configuration schema that we use.

The pan/tilt unit presents a robust and compact design, yet being light enough to permit a good turning speed (a maximum of 248ž/sec for the tilt angle). The movement is smooth and quiet. The tilt angle is in the š 120ž range, with a repeatability of š 0.02ž. The PowerCube device can receive position, speed and torque commands for each joint, and can provide feedback about the pan and tilt angles at any given moment.

The data collection procedure is implemented as a finite state machine. As the scan order is given, a command to move the wrist up to its final superior position is sent. When the error between that value and the actual tilt value given by the reading from the wrist sensor is below a threshold,the 3D data acquisition process is ready to begin. The wrist is commanded to go down and the laser measurements associated to the tilt value at every cycle are stored in both memory and text files. When the difference between the tilt value and the due final tilt value downwards looking is below the threshold, a command to get the wrist up again is sent and the data are stored in reverse order, to keep the point cloud structured for further processing. As mentioned before, this process is controlled by a scan parameter that sets the frequency with which the point clouds are obtained.

At first, we teleoperated the robot and made it stop before manually demanding to take a 3D scan, collecting data only when the wrist was going down. Owing to the fact that for the competition we required continuous updating to achieve real 3D obstacle avoidance capabilities, we decided to make it capture data when coming up as well.

III. THE SIMULATED ROBOT NEMO P2AT

Building a robot able to take 3D scans in USARSim is not a complicated task. There are 2 straightforward possibilities:

- The first one is to use the 3D range scanner sensor. USARSim includes with its stable release a 3D range scanner that returns 3D point clouds within a range that can be configured by the sensor parameters.
- The second one is to use the already available robot Kurt3D. Kurt3D is a robot which has a SICK range scanner mounted on a device that can be tilted and it is included in the latest versions of USARSim. Its tilting device accepts position commands, so that you can bring the laser to a desired angle. The device does not accept speed, nor torque, commands, and it does not provide the tilt angle either. Kurt3D's scanner does not have pan motion.

However, none of these two solutions could reproduce properly our real robot's behavior.

-The 3D range scanner is significantly different from our real sensor. Using it, one cannot control the scan's limits. We could want, for example, to initiate our scan 45 degrees under the zenit and end it at a tilt value of 10 degrees above it, which is not possible with this kind of sensor. Furthermore, it does not let us control the tilt angle interval between two consecutive 2D scans.

-Kurt3D was a close solution albeit it also presents some drawbacks. The most relevant one is the fact that it cannot provide feedback about the value of the tilt angle. This can be solved in two ways. The first option is to give a small angle increment and wait long enough in order to make sure that the commanded angle has been reached. An alternative may be to mount an additional sensor, like an INS (to get the angle directly), or an encoder. Both solutions are relatively easy to implement in USARSim. Another problem this configuration presents is the fact that you cannot control the tilt device using speed or torque commands.

In our first experiments to gather 3D data from the simulator we used Kurt3D's tilt unit and implemented a step by step motor system as described in [7]. Our simulated Nemo is shown in Fig. 3. Its basis is a robotic P2AT standard platform. We added Kurt3D's *scannersides* sensor to the USARBot.NemoP2AT model we had created and set it as the parent for the SICK LMS laser. The main modifications introduced to the USARBot.ini file are the following.

```
JointParts=(PartName="ScannerSides",
PartClass=class'USARModels.Kurt3DScannerSides',
DrawScale3D=(X=1.0,Y=0.4,Z=1.0),
bSteeringLocked=true,bSuspensionLocked=true,
BrakeTorque=100.0,Parent="",
JointClass=class'KCarWheelJoint',
ParentPos=(X=0.16,Y=0.0095,Z=-0.16),
ParentAxis=(Z=1.0), ParentAxis2=(Y=1.0),
SelfPos=(Z=0.0), SelfAxis=(Z=1.0),
SelfAxis2=(Y=1.0))
Sensors=(ItemClass=class'USARModels.SICKLMS',
ItemName="Scanner1", Parent="ScannerSides",
Position=(X=0,Y=-0.0095,Z=-0.11),
Direction=(X=0,Y=0,Z=0))
```

Fig. 3. Our simulated Nemo P2AT

The corresponding NemoP2AT.uc file to create the class was based upon the existing one for the standard P2AT.

After adjusting the 3D laser height parameters, this simulated robot performed well and let us obtain 3D point clouds from several different environments. In Section 6 more details about parameters' values and results will be given.

IV. THE SIMULATED ROBOT ATRVJR3D

The configuration described in the previous section was fine to acquire 3D data in a stop-scan-go manner, but that system was too slow to work in more realistic conditions. We added an encoder to avoid needing to wait every time we sent a command and be able to move the *scannersides* all at once to its final position up or downwards looking. However, as the device's speed could not be regulated, the readings did not arrive on time and the error was much larger; it was difficult to configure the settings to obtain consistent data.

For these reasons, we decided to build a pan/tilt device emulating the real PowerCube070 wrist. In USARSim this is quite easy to do thanks to the mission packages. We built a laserpantilt mission package which consisted of two rotational joints. These joints can be controlled independently with angle, speed and torque, and they provide feedback about their position. A similar example of a mission package is the camerapantilt package, distributed with USARSim. Once the SICK range scanner was mounted on top of such a device we were able to acquire 3D data in pretty much the same way as with our real robot.

For the RoboCup 2009 competition, we mounted this 3D device on an ATRVJr platform; we called our robot ATRVJr3D. The reason why we did so is simple, a P3AT cannot afford to carry two SICK range scanners without compromising the dynamics of the platform. We wanted to have both a 2D and a 3D range scanner on the same robot, the 2D one for 2D SLAM and the 3D one to obtain the point clouds that would be processed afterwards. We opted to use a SICK laser for the 3D data acquisition rather than an Hokuyo laser device as proposed in our Team Description Paper [10] mainly to cover a wider distance range and to have less noise. We also took into consideration the fact that Hokuyo lasers scan a wider field of view and therefore make ray tracing in USARSim get notably heavier.

Fig. 4. Our simulated ATRVJr3D

With the P3AT, equipped with just one SICK as described in Section 2, we acquire the 3D scans with the robot still, for the robot's position cannot be accurately corrected in the meantime, while the laser has not yet finished the acquisition of a complete 3d point cloud. With the simulated ATRVJr3D, as the robot can be localized using the fixed SICK while the other laser may be still collecting data, we can continuously acquire 3D data, stopping the robot only when requiring enhanced precision. The simulated ATRVJr3D is shown in Fig. 4.

For this robot we created the USARBot.ATRVJr3D class and added the following lines to its definition in the USARBot.ini file so that the new sensor be properly positioned, on top of the robot without colliding or seeing the camera and the other laser scanner.

```
MisPkgs=(PkgName="LaserPanTilt",
Location=(X=0.16,Y=0.0095,Z=-0.25),
PkgClass=Class'USARMisPkg.LaserPanTilt')
Sensors=(ItemClass=class'USARModels.SICKLMSr',
ItemName="Scanner3D", Parent="LaserPanTilt_Link2",
Position=(X=0,Y=-0.0095,Z=-0.06),
Direction=(X=0,Y=0,Z=0 ))
```

The SICKLMSr is a normal SICK LMS laser except for the fact that it is drawn downwards looking in the simulator (Fig. 4 must not lead the reader to confusion, it was taken before the last changes only for esthetic purposes). We used this class during the competition because we could not add yet another laser model and we wanted to change the maximum range in order to reduce the simulator's burden.

V. NEW TECHNIQUES INTRODUCED FOR THE COMPETITION

The major novelty we introduced for the competition was the creation of a ground map to make sure that the robot would not fall into holes nor collide with low obstacles when in autonomous operation. To do so, we checked the 3D points obtained from the data provided by the tilting laser. The points whose z coordinate's absolute value was below a threshold were converted to 2D points in the map image's reference frame and considered as belonging to an area the robot could safely traverse. We established several criteria to overwrite the colors in the 2D grid map generated with the

data coming from the horizontal laser. Our list of colors is the following:

- Black: obstacles in the map generated with the 2D data
- White: free areas in the map generated with the 2D data
- Blue: unexplored areas in the map generated with the 2D data
- Gray: obstacles detected by the 3D laser
- Green: solid ground free of holes and 3D obstacles (traversable areas)

The three first colors were used as defined in the rules for the competition. We added the other two colors to provide more information for the operator and to exclusively use the green areas for safe navigation.

At first, the ground map was not updated until a whole 3D point cloud was acquired, and green was never allowed to overwrite gray. This method overloaded the system a bit, and whenever a 3D scan was completed an important delay prevented the robot's pose from being properly updated. So, we decided to update the ground map periodically, with a relatively small period, only using the data that had been obtained within the last cycle.

Not letting green substitute gray was important to avoid removing obstacles situated at intermediate heights when the wrist was driving the laser down. However, we did want to remove 3D obstacles which were no longer there: dynamic objects (other robots), nonexisting small obstacles generated by noisy measurements .... We also wanted to eliminate some ugly lines that cropped up from time to time as a result of suddenly stopping the robot, as severe decelerations made it incline.

Our choice was to allow green overwrite gray only when the 3D laser was capturing data towards its upper position. This way, real 3D obstacles detected below the robot's height were not permanently there but as soon as the 3D laser saw them again when coming up, they were restituted so that reactive control could avoid them.

When the robot was spawned to operate autonomously right from the beginning of a mission it would not move, since the wrist does not get the laser to capture data below it. The same thing could happen as well when the robot explored new narrow areas appearing from behind a corner, for example. To solve that problem a fixed size green square was created around the robot's pose on the assumption that there would not be a hole in the closest surroundings, so that the movement towards the chosen frontiers for the autonomous exploration could be initiated. Fig. 5 shows some examples of ground maps generated like this.

The areas explored by the 3D laser sensor were sometimes not continuous due to the discrete distribution of the laser's beams, whose effect is most noticeable when the, normally slightly tilted, laser scans further obstacles. To reduce this undesirable result which did not always let the robot advance we adopted two measures.

The first one was rather simple and came to not only coloring green the pixels corresponding to a single 3D point considered an obstacle but also those in a window around that pixel in the map image. This worked quite well, as we

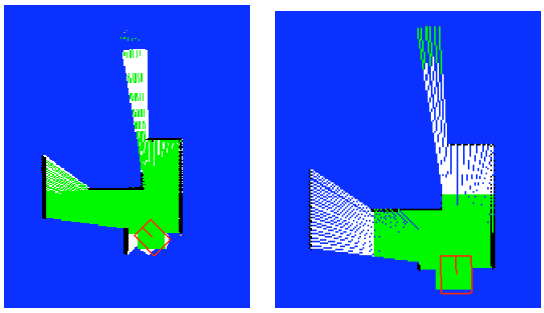


Fig. 5. Creation of a green square around the robot's pose so that it can start autonomous exploration. Left: small square. Right: larger square

kept on making sure that these pixels could not erase gray or black obstacles. Still, there remained some small white holes inside clearly green safe areas and this made the robot's motion slower.

To get green areas even, we decided to apply a *closing* (dilate + erode) morphological operation to the map image. What we did in the end was to iteratively apply a *floodfilling* algorithm to every white pixel in the map image, filling the area they belonged to with a different color. Afterwards, the new colored areas' size was checked to avoid removing real holes from the ground map; small holes were filled with green while bigger holes were turned white again. Fig. 6 is one such map image before classifying and recoloring the detected holes.

For the implementation of the algorithm we used the OpenCV libraries [13]. All our software is composed of a collection of self-developed and open-source libraries and modules, within the OpenRDK framework [14].

When there were a lot of white points inside green areas (due to excessive noise, in certain world models...) the execution of the *closing* module took too long and the whole system was affected. Therefore, the application of the *floodfilling* algorithm was restricted to a smaller region of the image containing the robot's pose and this technique was performed only when significant changes in the robot's pose were appreciated. Moreover, we introduced a limit in the number of holes so that when there were too many small white points close to the robot some of them were left white for subsequent cycles. All this yielded an overall good performance of the system.

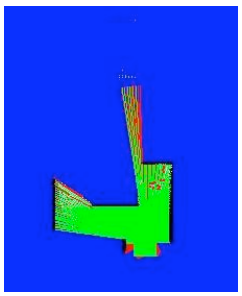


Fig. 6. Map image after applying a floodfilling algorithm with white pixels as seeds

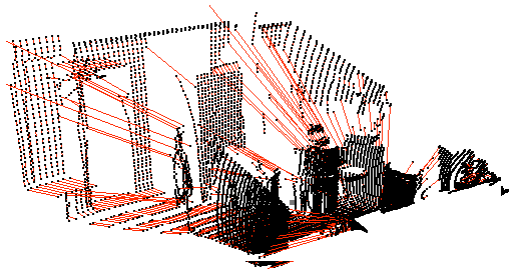


Fig. 7. 3D point cloud obtained with our real scanner

Fig. 8. Segmentation image obtained from the 3D scan in Fig. 7

VI. EXPERIMENTS AND RESULTS

USARSim has served us as a great tool for development and experimentation. As commented in the introduction, at fist we used it as a means to obtain new data to validate our already developed 3D mapping algorithms [7]. Our main concerns in the initial stage dealt with the resolution of the 3D point cloud. To capture the 3D data with the real robot, the laser is tilted from -25 degrees (upwards looking) to 30 degrees (downwards looking) at a constant speed of 0.05 rad/s, which typically results in a 70x181 matrix of measurements. Fig. 7 and Fig. 8 show a point cloud gathered with our real system and the segmentation image obtained from it by applying the algorithms described in [15], [7], of which a brief account follows.

The residuals from fitting each points neighborhood to a plane are used as measurements to generate a range image from the structured 3D data acquired by the laser scanner. A second order *closing* (dilate + erode) morphological operation is applied to the image so as to help edges be better defined, without breaking improperly.To eliminate false borders caused by the presence of noise, the image is binarized so that only important edges are left. Once this has been accomplished, a floodfilling algorithm is used to assign the same color to the pixels inside each large enough region enclosed by the remaining borders. This way, taking advantage of the ordered nature of the data, real-time capabilities are achieved. A final check comparing local normal vectors is performed to make sure different surfaces are correctly separated even if some edges in the image are not satisfactorily closed. This allows for further robustness without being significantly more time consuming.

When working with the simulated Nemo P2AT, at first we set the tilt limits at 23 degrees and -30 degrees (notice that the tilt's sign is inverted in relation with the real PowerCube wrist) and got a laser 2D scan at every 0.25 degrees in order to obtain continuous data from the floor and the ceiling.

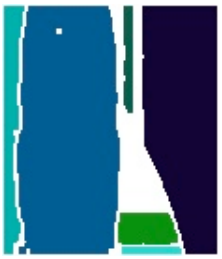


Fig. 9. Segmentation image with too many rows and not covering lower parts

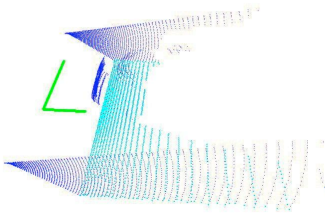


Fig. 10. A point cloud captured with the simulated Nemo P2AT in the DM_Mapping 2006 USARSim world. View from above, the floor is depicted in cyan. Only the rear part of the car can be seen from the robot's current pose, marked in green, with the robot facing the shortest axis

This, however, led to having too many rows in the 3D data structure and the images created did not allow for an easy recognition of the objects in the scene. Furthermore, the tilt variation proved not be enough. An example of this is shown in Fig. 9, obtained from a 3D scan got from the *PlayerStart* pose in DM_Mapping.

With the final configuration, using the mission package and commanding the wrist with a final angle, the results improved significantly. Fig. 10 shows a 3D point cloud obtained from the same pose and Fig. 11 is the segmentation image corresponding to that point cloud. The threshold referred to in Section 2 is slightly smaller than the one we used when operating with the step by step motor system, for a more precise final position of the laser is desired. Apart from that, the other parameters are kept the same. A useful comparison to perform may be to acquire some data sets with both the real robot and its simulated model and analyze the discrepancies in the number of 2D scans (rows in the corresponding image) integrating the 3D scans of both categories.

The uncountable series of experiments we conducted for

Fig. 11. Segmentation image obtained with the final configuration obtained from the same pose as the one shown in Fig. 9. The car can be better identified now

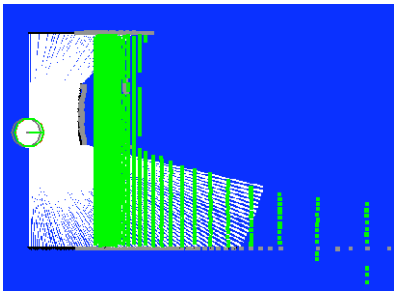


Fig. 12. Ground map corresponding to the point cloud in Fig. 10

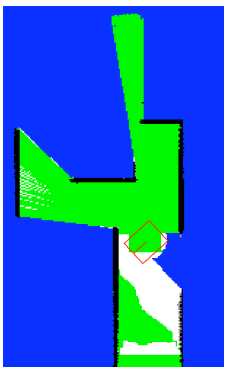


Fig. 13. Ground map corresponding to a 3D scan acquired at *robot1* pose in DM_Mapping

the competition were not reproduced with the real robot, for two main reasons: lack of time and the fact that our robot is not equipped with two laser range finders.

The segmentation of the simulated data worked quite well with no changes in the parameter's values with respect to those used with the real data. However, if the threshold used for the image binarization is smaller, somewhat better results are obtained if the laser's noise presents its standard value for the SICK LMS200.

Tests with USARSim let us decide on the values of the parameters for the construction of the ground map so that in the end it was quite effective. Fig. 12 and Fig. 13 show some images of such maps.

VII. CONCLUSIONS

In this paper we have presented the robot models we have used to acquire 3D data from USARSim environments along with some comparisons regarding our work with the real robot and some of the innovations we have introduced for the RoboCup'09 Search and Rescue Virtual Robots Competition. USARSim has undoubtedly been a very valuable tool for acquiring a wide variety of data to validate our algorithms and it has saved us a lot of time, making things convenient and realistic.

VIII. ACKNOWLEDGMENTS

The authors gratefully acknowledge the Spanish Ministry of Science and Innovation (DPI2007-66846- c02-01), the Comunidad de Madrid (Consejeria de Educacion) and the European Social Fund (ESF) for supporting this work.

REFERENCES

[1] (2008) Iros'08: Workshop on robot simulators: available software, scientific applications and future trends. [Online]. Available: http://www.robot.uji.es/research/events/iros08
[2] Mobilesim. [Online]. Available: http://robots.mobilerobots.com/wiki/MobileSim
[3] Activmedia robotics. [Online]. Available: http://www.activrobots.com/
[4] Player/stage/gazebo project. [Online]. Available: http://playerstage.sourceforge.net/index.php?src=index
[5] (2009) The USARSim website. [Online]. Available: http://usarsim.sourceforge.net/
[6] B. Balaguer, S. Balakirsky, S. Carpin, M. Lewis, and C. Scrapper, "USARSim: a validated simulator for research in robotics and automation," in *IROS'08.Workshop on robot simulators: available software, scientific applications and future trends*, Nice, France, 2008.
[7] P. de la Puente, D. Rodríguez-Losada, A. Valero, and F. Matía, "3D Feature Based Mapping Towards Mobile Robots Enhanced Performance in Rescue Missions," in *Proc. IEEE International Conference on Intelligent Robots and Systems (IROS'09)*, 2009.
[8] (2009) The RoboCup rescue website. [Online]. Available: http://www.robocuprescue.org/
[9] (2009) The RoboCup rescue website. virtual robots competition. [Online]. Available: http://www.robocuprescue.org/wiki/index.php?title=VRCompetitions
[10] A. Valero, G. Randelli, P. de la Puente, D. Calisi, D. Rodriguez-Losada, F. Matia, and D. Nardi, "UPM-SPQR rescue virtual robots team description paper," Team Description Paper for RoboCup 2009.
[11] MobileRobots Inc. [Online]. Available: http://www.mobilerobots.com
[12] Amtecrobotics. [Online]. Available: http://www.amtec-robotics.com/index.html
[13] Opencv libraries. [Online]. Available: http://sourceforge.net/projects/opencvlibrary/
[14] (2009) The OpenRDK website. [Online]. Available: http://openrdk.sourceforge.net/
[15] P. de la Puente, D. Rodríguez-Losada, R. López, and F. Matía, "Extraction of geometrical features in 3D environments for service robotic applications," in *Proc. 3rd. International Workshop on Hybrid Artificial Intelligent Systems (HAIS'08)*, ser. LNCS, A. A. P. W. Corchado, E., Ed., vol. 5271. Springer-Verlag, 2008, pp. 441–450.

A Color Based Rangefinder for an Omnidirectional Camera

Quang Nguyen and Arnoud Visser

Abstract— **This paper proposes a method to use the omnidirectional camera as a rangefinder by using color detection. The omnicam rangefinder has been tested in USARSim for its accuracy and for its practical use to build maps of the environment. The results of the test shows that an omnidirectional camera can be used to accurately estimate distances to obstacles and to create maps of unknown environment.**

I. INTRODUCTION

An important aspect of robotics is the task of collecting detailed information about unexplored or disaster struck areas. A part of this task is to make a robot measure distances to obstacles in order to localize itself and to create a map of the environment. This paper proposes the use of an omnidirectional camera combined with a color based free-space classification system to create a rangefinder which can estimate these distances. 'Can an omnidirectional camera be used effectively as a rangefinder?' is the research question of this paper.

Until now, active sensors like laser scanners or sonar are being used as rangefinders. Active sensors can generate highly accurate measurements, but have their limitations. They typically scan a surface, missing obstacles just above or below this surface. Some highly reflective or absorbent surfaces are invisible for these sensors. Further they have limits on their range and field of view. Because it are active sensors, they are relatively heavy and can consume a substantial amount of the robots battery capacity. An active sensor on a small or flying robot is therefore not a viable choice.

This is why it is important to focus attention on alternative sensors like a passive sensor as proposed in this paper. A passive sensor based on omnidirectional camera can have a 360^o field of view by using an omnidirectional camera. However a visual sensor tends to be inaccurate compared to a laser sensor and it is hard to estimate the depth on an image using an omnidirectional camera.

Again, this paper proposes a method to implement a rangefinder for an omnidirectional camera. This rangefinder uses a color histogram, which is a color based statistical model, to identify free space in the immediate surroundings. Furthermore a comparison between an omnicam and a laser determines which sensor, in terms of accuracy and practical use, performs better in what kind of environment or circumstance. Practical use is tested by letting a rangefinder sensor build a map of an environment. Scanmatching algorithms are used in combination with the rangefinder in order to estimate its position on the map, which is an important part of mapping unknown environments. The implementation and testing of this method is done in USARSim [1], which is a simulation environment that can be used as a research tool [2].

Intelligent Systems Laboratory Amsterdam, Universiteit van Amsterdam, Science Park 105, 1098 XH Amsterdam, The Netherlands

Section 3 describes the theory and method that have been used for the development and validation of the omnicam rangefinder. Section 4 describes how the experiments are set up to test the omnicam rangefinder. Section 5 handles the outcome of the test results while section 6 describes the discussion and further work. The final section makes a conclusion of this research.

II. RELATED WORK

Several methods of using a visual sensor to detect free space have already been created. The winner of the DARPA challenge 2005 used a visual classifier based on color information to estimate the road ahead [3]. Rauskolb et al. [4] improved this algorithm so it can also be used in urban environments.

Since the introduction of the omnidirectional camera in USARSim [5] many new applications have been designed for the omnicam. Roebert used the omnidirectional camera to create a bird-eye view map of an environment [6]. However these bird-eye view maps of the environment are not usable by autonomous robots for navigation. The robots need to have an obstacle detector, know what the free space is and have the ability to create a map where the free space and obstacles are shown in order to navigate through an environment.

Maillette de Buy Wenniger [7] created a free-space detector that uses probabilistic methods to learn the appearance of free space in a bird-eye view image based on the color signature. This free space detector is extensively tested, but not used for navigation or mapping yet. Scaramuzza [8] has designed a rangefinder for a low-cost omnicam sensor that can detect and measure the distance to obstacle points in a simple black and white world. In this paper both approaches are combined with a scanmatching algorithm [9], which allows localization and mapping of an environment purely on visual information.

III. METHOD

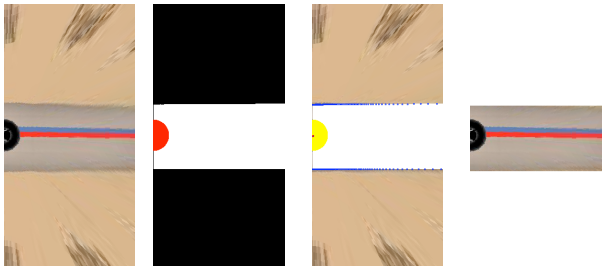
This section describes the method and theory behind the omnicam rangefinder. The theory behind the rangefinder is to use a trained color histogram to classify free space pixels in an image. Maillette de Buy Wenniger's work [7] has shown that a color histogram can be used for reliable identifion of free-space for a simulated and a real robot.

This paper suggests an omnicam rangefinder which uses the free space detection system of Maillette de Buy Wenniger to classify pixels as either free space or non-free space. From there on the rangefinder uses this knowledge to detect boundary points of an obstacle by using polar scanning combined with false-negative and false-positive filters. The robot can then estimate the metric distance between the robot and the boundary point. After the distances have been estimated an outlier rejection filter is used to reject the estimated distances that have a high probability of being inaccurate.

A. Free space pixel identification

The color histogram is a statistical model used to identify free space pixels based on their color values. It needs to be trained by using a set of training data which consists of a collection of pixels where the class, which can be free space or non-free space, is already known. A trained color histogram can then calculate the probability if a certain color value belongs to a certain class.

1) Collecting training data: When the robot is deployed into an unexplored environment it is going to need a laser scanner to create the training set for the color histogram. After creating the training set the robot can venture further into unexplored areas without using the laser scanner to measure the distances to obstacles. In order to build the training set the robot needs to provide a bird-eye view image of the environment [6] and the laser measured distances image of the environment. These two components are then fused to create a bird-eye view image where the free space is shown. The pixels that are classified as free space on that image are used to train the histogram. This entire procedure is visualized in figure 1. The laser rangefinder has a field of view of 180^o compared to the omnicam's 360^o, therefore the created bird-eye view images of the omnicam has been cut in half to make it synchronize with the laser image.



(a) Bird-eye view in the factory environment. (b) Image create by the laser range measurements. (c) The combined image of the bird-eye view image and the laser image. (d) The resulting image that is used for training.

Fig. 1: Procedure to create training data. The half circle on the left part of the images is the base of the omnicam, this will not be used for the training data.

A bird-eye view image can be created by following Nayar's described relation between a pixel on an omnicam image and the corresponding pixel on the bird- eye view image if nothing obstructs the view [10]. A pixel on an omnicam image is described by $p_{omn} = (x_{omn}, y_{omn})$ and a pixel on the bird-eye view image is described by $p_{be} = (x_{be}, y_{be})$. These are the equations:

$$= \arccos \frac{z}{\sqrt{x_{be}^2 + y_{be}^2 + z^2}}, \tag{1}$$

$$= \arctan \frac{y_{be}}{x_{be}}, \quad r = \frac{R}{1 + \cos} \tag{2}$$

$$x_{omn} = r \sin \cos , \quad y_{omn} = r \sin \sin \tag{3}$$

The constant R is the radius of the circle describing the 90^o incidence angle on the omnicam effective viewpoint. The variable z is used to describe the distance between the effective viewpoint and the projection plane in pixels. Roebert's work [6] has shown that accurate bird-eye view maps can be achieved by using these formulas.

An image of the laser measured distances can be constructed by using a laser rangefinder to measure the distances on every scanline between a 'hit point' and the origin of the laser, which is the sensor position on the robot. If the distance of the 'hit point' is bigger than the detection range, the laser will return the maximum detection range for that scanline. Multiple laser measurements can be done to increase the probability that free space is detected on a certain scanline. These measured distances can then be converted to an image like in figure 1b.

Finally a combined image between these two components can be created and the color pixels that are classified by the laser rangefinder as free space on that image can be used as the training data for the color histogram.

2) Color Histogram: The color histogram is trained by counting how many times a certain RGB value exists in the training data. Before the training can start a decision about the number of bins (n) of the histogram has to be made. In this study histograms with 13 bins are used. A trained color histogram is usable to classify free space pixels in new omnicam images. The trained color histogram uses this discrete probability distribution to classify a pixel as either free space or non-free space:

$$P_{HIST}(rgb) = \frac{c[rgb]}{T_c} \tag{4}$$

Here, $c[rgb]$ returns the count of the histogram bin that is associated with the rgb color. T_c returns the total count of all the bins of the histogram. The outcome of a particular color will thus have a value between 0 and 1. In order to filter out the false positives the histogram uses a probability threshold is set as $0 \leq \leq 1$. Therefore a pixel is considered as free space if

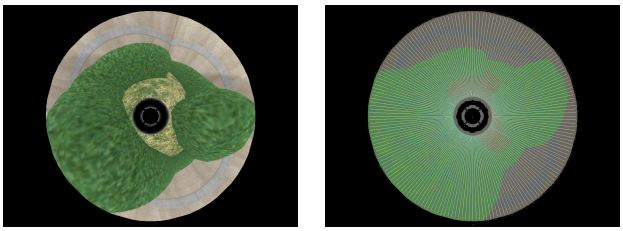
$$P_{HIST}(rgb) \geq \tag{5}$$

In our experiments, a different threshold was used to accommodate for the circumstances in the different environments (as indicated in table I).

B. Omnicam Rangefinder

The basis of this method has been derived from Scaramuzza's black and white omnicam rangefinder [8]. However the proposed omnicam rangefinder in this article uses the color-based free-space detection described in section 2.1 instead of only detecting black and white colors. This free-space detector is combined with the rangefinders very own detection method which uses polar scanning combined with false-positive and false-negative filters to detect pixels of an obstacles boundary point. These detected pixels are considered as the hit points of the scanlines. Having detected a hit point the rangefinder estimates the metric distance between the robot and that hit point. At the end of this method an outlier rejection filter is used to reject the estimated measurements that have a high probability of being inaccurate.

1) Polar scanning: The omnicam rangefinder uses polar scanning to create scanlines that are coming from the center of an omnicam image, a visualization can be found in figure 2b. Every pixel in each scanline gets classified as either free space or non-free space according to the trained color histogram. The omnicam rangefinder then uses its false-positive and false-negative filters to find the correct hit point for each scanline. These filters use a small number of parameters which can be optimized on the environment or situation, as indicated in table I.



(a) The original image of the omnicam. (b) The same image with scanlines visualized on it.

Fig. 2: A visualization of polar scanlines on an omnicam image.

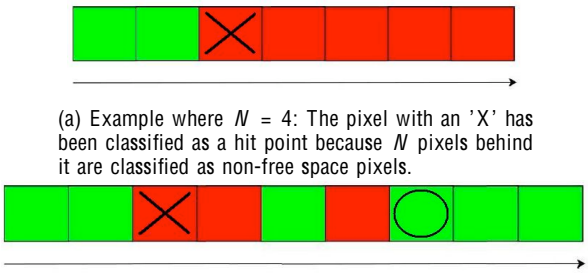
False-positive filter:

This filter determines if a non-free space pixel is a hit point by checking if N pixels behind it are also classified as non-free space pixels. An example of this filter can be found in figure 3a.

False-negative filter:

This filter makes sure that a candidate hit point does not get rejected because a free-space pixel, which was actually misclassified, interrupts the sequence of non-free space pixels described in the false-positive filter. The parameter K determines how long the sequence of free-space pixels should be before rejecting the candidate hit point. An example is given in figure 3b.

The metric distance from the robot to a pixel is calculated whenever that pixel is classified as a hit point according to the rangefinder. When there is no hit point on a scanline the rangefinder returns the maximum range, which is another variable parameter. When the hit point is too close to the position where the robot is, this would make the measured distance unreliable, the rangefinder returns the minimum range, which is also a variable parameter.



(a) Example where N = 4: The pixel with an 'X' has been classified as a hit point because N pixels behind it are classified as non-free space pixels.

(b) Example where K = 2: Hit point has not been found because there is more free space starting from the 'O' pixel. This is because K pixels behind it are also classified as free-space pixels. Note that the free-space pixel between the two non-free space pixels is negated because the next pixel behind it is classified as a non-free space pixel.

Fig. 3: Illustration of the false-positive and false-negative filters. The figures represent a set of pixels from a scanline, green pixels are classified as free space and red pixels are classified as non-free space pixels.

2) Measuring distance: The rangefinder calculates the metric distances to each hit point that it can find. The metric distance d is calculated by using the formula from Scaramuzza [8]:

$$d = h\tan(\ ) \tag{6}$$

where is the incidence angle on the mirror and h is the height in meters from the ground to the effective viewpoint of the hyperbolic mirror. The metric distance d is calculated in meters. The incidence angle can be estimated by a first order Taylor expansion $\approx -$ when the shape of the mirror is not well known. represents the pixel distance from the hit point to the center of the image, is a constant value that depends on the mirror shape and the camera-mirror distance. The constant can be estimated by calibrating the omnicam sensor measurements against a laser range sensor measurements. The variables , , and h are illustrated in figure 4. In our case, the shape of the mirror is well known; $= 2\arctan(\frac{r}{R})$.

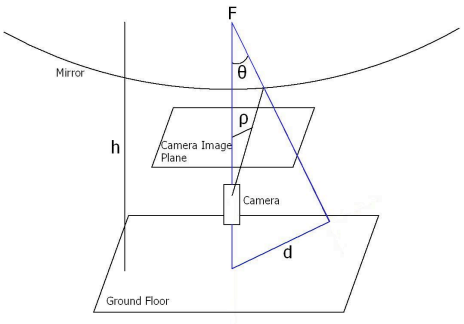


Fig. 4: Illustration of the location of the mirror, image plane and ground floor, as used in equation (6).

Pixels which have a difference larger than 1.25m between the distance estimate of the laser range scanner and the omnicam range scanner were not used for training or evaluation.

IV. EXPERIMENTAL SETUP

This section describes the testing environment and the experiments that are used to test the omnicam rangefinder. The implementation of the omnicam rangefinder and the testing of it were done in USARSim which is a simulation environment. The omnicam rangefinder was tested for its accuracy and for its practical use to create a map of an environment.

A. USARSim

USARSim is a 3D simulation environment that can simulate real world environments and situations. This program is intended as a research tool to study the use of robots in the real world. Robots in USARSim can therefore use simulated realistic tools to complete their tasks. Experiments have shown that perception algorithms that are developed in USARSim can easily be converted and used for the real world [2].

B. Accuracy Test

This first experiment compares the measured ranges from an omnicam against the measured ranges from a laser. The laser sensor is highly accurate[1] and can therefore be used as a reference measurement. The ranges are collected by letting the robot drive around in an environment while measuring the distances using both the omnicam and the laser. Each scanline measurement from the laser gets compared to the appropriate scanline measurement of the omnicam. By subtracting the omnicam measurements from the laser measurements one can get the differences between them. These differences can then be plotted in a histogram to show the omnicam's accuracy compared to the laser's accuracy.

C. Map Building Test

The second experiment tests if the omnicam sensor can actually be used for localization and building maps. The sensor has to create accurate maps of environments in order to achieve this. Again, the laser sensor is used to create a reference map of the environment. The omnicam then drives exactly the same route as the laser to create an omnicam map. The omnicam map is then compared with the reference map to determine how different the map is.

The map building is done using two different ways:

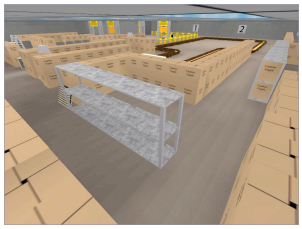
- Using Deadreckoning with the GroundTruth as the sensor. This reference setting makes sure that the robot always knows where it is on the map. This means that the robot can always localize itself without scanmatching, therefore the quality of the map fully depends on the accuracy of the sensor.
- Using Quad Weighted ScanMatching (QWSM) [11] with an Inertial Navigation System (INS) as the initial pose estimate. INS uses the robots acceleration sensors to estimate the current pose. That pose estimate is checked on consistency with the observations of the range scanner, which results in a new estimate of the current pose. This setting is a more realistic setting and must be used when the robot is venturing into unknown terrain.

Using QWSM and INS results in a less accurate map, but the omnicam needs to be evaluated with this setting in order to proof that it can be used in a realistic configuration.

D. Environments and robot model

Two different environments are used to test the omnicam rangefinder. The first one is a maze as shown in figure 5a. This environment consists of very small corridors, lots of turns and the hedge has nearly the same color as the floor, which is covered with grass. The challenge is to correctly map the small corridors and to make a distinction between the hedge and the grass. The second environment is a factory as seen in figure 5b, this environment has relatively wide corridors, less turns and it contains a number of unique objects. The challenge in this map is also to detect those unique objects as obstacles, figure 6 shows a couple of examples of these obstacles.

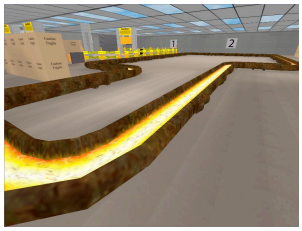
(a) The maze environment.



(b) The factory environment.

Fig. 5: Environments in USARSim to train and test the omnicam rangefinder.

(a) A cabinet.



(b) A pipe with liquid metal.

Fig. 6: Some unique objects in the factory environment.

The type of robot that is used to do these experiments is the OmniP2DX (Figure 7), which is a robot equipped with a laser scanner and an omnicam sensor. The height of the omnicam's effective viewpoint h to the ground is 0.931m.

Before starting the experiments it is important to choose a maximum range of the algorithm described in section **??**. The maximum range is dependent on the curvature of the mirror of the OmniP2DX. The maximum range can be estimated by rewriting the equation (6) and performing several sample distance measurements by letting a range sensor measure a

[1] A SICK LMS 200 has indoors a statistical error $< 5mm$

Fig. 7: OmniP2DX, with the omnicam sensor high above the Sick laser range scanner.

Map	Nonfreepixels	Groupedpixels	Probabilitythreshold
Maze	20	2	0.05
Factory	20	2	0.075

TABLE I: Parameters used in the algorithm

distance in meters and letting the omnicam sensor measure that same distance in pixels. Having estimated the the distance formula can be used to plot the relation between the pixel distance and the metric distance. Figure 8 shows this relation and it also shows that the discretization error between pixels increases hyperbolically. This means that having an off-by-one-pixel-error misclassification on a distance far away from the robot can result into a high metric distance error. Setting the maximum range at 3.8 meters is therefore a good tradeoff between the maximum range and the accuracy of the omnicam. The number of scanlines used was 360.

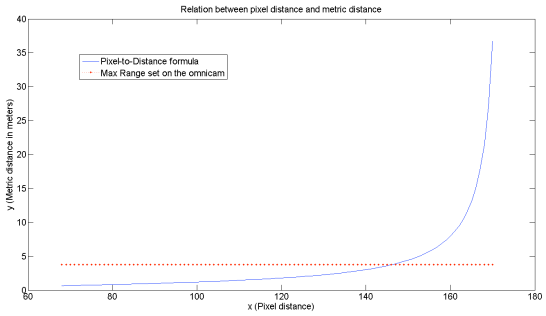
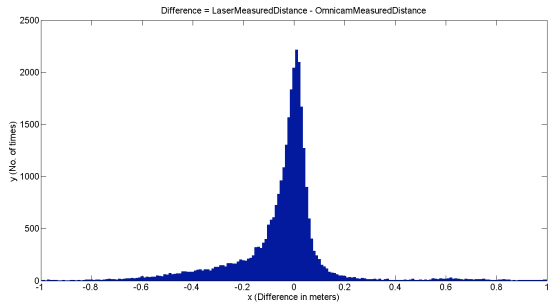


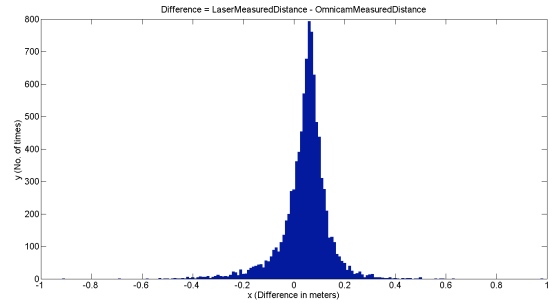
Fig. 8: The relation between the pixel distance and the metric distance.

V. RESULTS

This section deals with the results of the experiments with the omnicam rangefinder. The results have been obtained by using the parameters stated in table I. These are the optimal parameters that have been derived by hand. At the time of writing there exists no learning algorithm for finding the optimal parameters for every situation is applied. The laser sensor used for these experiments is the SICK laser which has a maximum range of 19.8 meters compared to the omnicam's maximum range of 3.8 meters. The image resolution for the omnicam images used for these experiments is 1024x768 (R=192 pixels).



(a) Maze: Histogram of the omnicam accuracy



(b) Factory: Histogram of the omnicam accuracy

Fig. 9: The omnicam accuracy compared to the laser accuracy measured in both environments

A. Accuracy Results

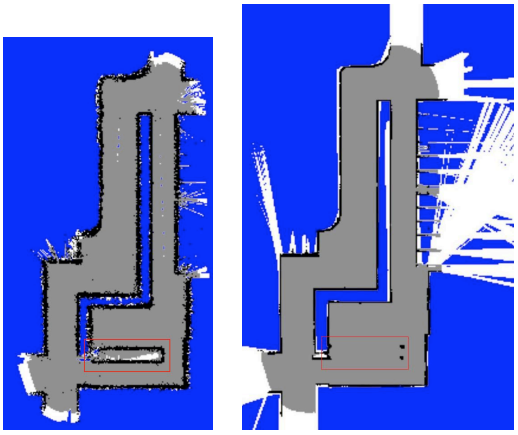
Figures 9a and 9b shows the results of the accuracy test of the omnicam rangefinder for respectively the maze and the factory. The figures show a bigger measurement error is made for the maze, but also that this error mainly due to the tail of the distribution. The factory measurements are quite symmetric whereas the maze measurement are skewed to the left, as can be seen in the histograms of figure 9. For the maze measurements the systematic error is nearly zero, while for the factory measurements the mean of the distribution is a few centimeters to the right.

The measurements of the factory environment were done in wide corridors, which means that the factory's histogram is showing the accuracy of the omnicam in a more optimal situation. The environment did not have a lot of corners compared to the maze environment which was chosen to stresstest the omnicam rangefinder.

Table II shows that the omnicam rangefinder has an average absolute accuracy difference of 8.09cm in the factory compared to 13.75cm in the maze. Notice that the route driven with the robot in the factory was shorter then the route through the maze, but nicely closes a loop.

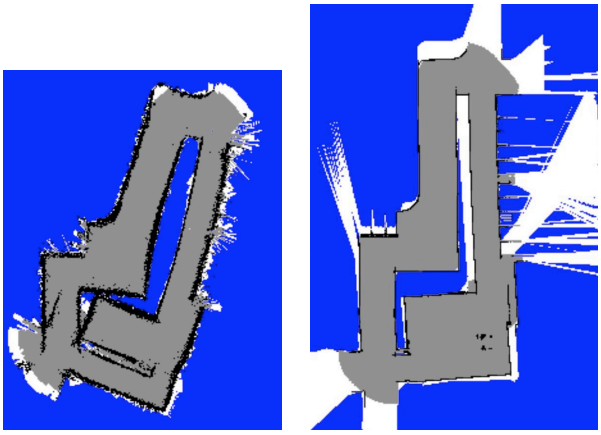
Map	Avg Absolute Difference	Avg Percentage Difference
Maze	0.1375m	7.639%
Factory	0.0809m	4.493%

TABLE II: Table with differences between laser and omnicam range measurements



(a) Omnicam sensor factory map (b) Laser sensor factory map

Fig. 10: Factory map created with localization on ground truth



(a) Omnicam sensor factory map (b) Laser sensor factory map

Fig. 11: Factory map created with QWSM and INS

B. Map Building Results

Factory environment: Figure 10a and 10b shows the results of building a map of the factory environment using an omnicam sensor and a laser sensor combined with the ground truth (available in simulation) as localization. Because of its accuracy the laser created map serves as an indication for what the ground truth map should look like. Comparing both maps shows that the omnicam map does not differ that much from the laser created map. The black dots and lines on the map represents detected obstacles, the gray color represents the safe space while the white color represents the free space detect by the rangefinder. Both gray and white indicates areas free of obstacles, but grey indicates areas that are well explored, while white indicates areas that could be further explored. The main difference by the maps generated with the omnicam and the laser, is the thickness of the walls. The omnicam map is not as razorsharp as the map generated with the laser scanner.

A less obvious difference between both maps is visible at the bottom of the map, indicated with a red rectangle. The omnicam map has found a obstacle at that location while on the laser map only four small dots are visible. The omnicam map is correct at this situation, there is indeed a big obstacle (figure 6a) present on this location. The laser scammer looked right through the cabinet, because no shelf was present at measurement height of the sensor. Another obstacle which had the potential to be difficult to detect, the pipe with liquid metal from figure 6b, was detected without problems by both sensors. This pipe is visible on both maps as the curved upper left wall.

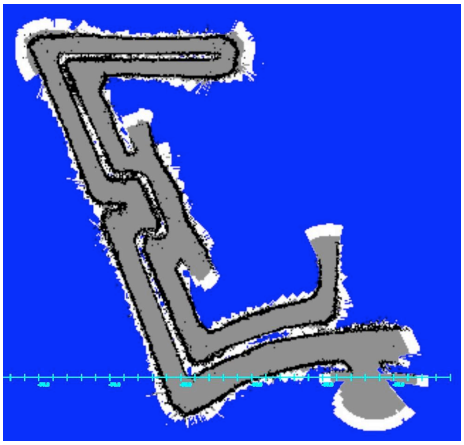
Figure 11 shows the results of the same route using QWSM scanmatching with INS. Comparing these maps with the maps from figure 10 shows that the map from figure 11b is more accurate than the map from figure 11a. These maps show that localization with the omnicam sensor performs worse than localization with a laser sensor for this situation.

The reason for this is that if the corridors are relatively wide, the omnicam with its limited range of 3.8 meters sees not that many features. The walls on other side of the crossing are out of range. When the robot is turning on a a location with sparse features, an error in the precise value of this rotation is easily made. In figure 11a small rotation errors are made on multiple locations.

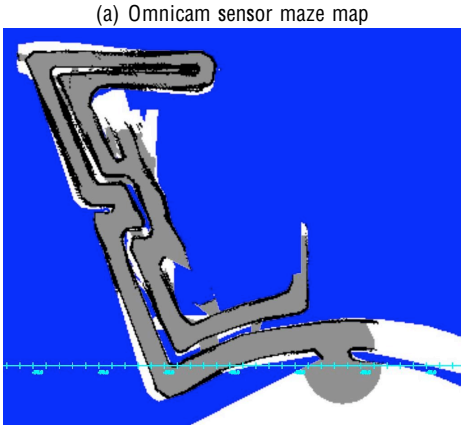
This rotational error could be corrected by post-processing, when loop closure is detected. Note that this are initial results, the used scanmatching algorithms were developed and optimized for the laser range sensor measurements. Although the applied scanmatching algorithms do not have that many parameters, no sensitivity study is performed to study the optimal parameter values for the omnicam range measurements.

Maze environment: Figure 12 shows the results using localization on the ground truth to build a map of the maze. Because of the turns in the maze the SICK laser loses its advantage to measure distances up to 20 meters. The robot started in the middle of the maze and made its way to the exit of it. Apart from the noise on the path both maps look quite similar to each other. The laser could sometimes look through the hedge because of the very small holes in the hedge. The laser would then have a preview on the path on the other side of the hedge. This could be a benefit, but it can also create some distortions like the area in the upper part of its map. The omnicam map does not have those problems.

Figure 13 shows the created maps of the maze with QWSM and INS. The localization for the laser sensor went wrong on this map. This can have several reasons. Firstly, the measurements right through hedges which creates outliers that can breakdown the scanmatcher. Further, the corridors on the outer edge are quite long. The laser range measurements could not detect the end of the corridor, which makes it difficult to notice progress in movement through such corridor. The omnicam has an even shorter range and can also not see the end of the corridor, but detects more structure in the walls which can be used to detect progress in movement. The omnicam map looks good, the map only has a small

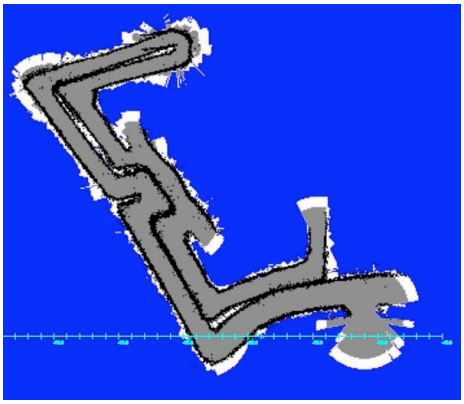


(a) Omnicam sensor maze map

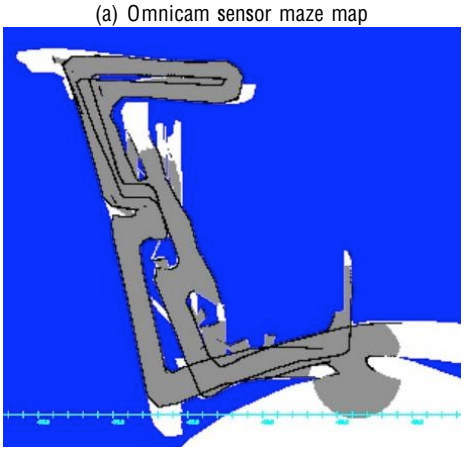


(b) Laser sensor maze map

Fig. 12: Maze map created localization on ground truth



(a) Omnicam sensor maze map



(b) Laser sensor maze map

Fig. 13: Maze map created with QWSM and INS

rotational error when compared with the maps from figure 12.

The maze proofs that the omnicam combined with localization can work in an environment with narrow corridors, lots of turns and a wall with nearly the same color as the floor.

VI. DISCUSSION AND FUTURE WORK

Accuracy

The difference in accuracy between an omnicam and a laser lies in the detection of object boundaries. Using color distinction to detect these boundaries does not provide perfect results. Using color detection can sometimes get a pixel misclassified as a hit point. These misclassifications are fatal for long distance measurements because of the hyperbolic shape of the mirror of the omnicam, as indicated in figure 8.

The accuracy tests also shows that the omnicam measures longer distances slightly further away than they in fact are. This is a good way to estimate the location of obstacles, as shown in the maps created by omnicam rangefinder. The first measurements on a large distance are initially drawn a bit away from the free space that could become a path for the robot. When the robot gets near that location, more accurate measurements indicate the precise boundaries of the path. Figure 8 shows that the metric distance error between those pixels on such a small distance is very small.

Future work for the accuracy improvement using color detection is trying other color spaces. This is also useful to let the omnicam rangefinder work in other areas where the RGB color space might fail. A fast way to increase accuracy is to use a higher image resolution, however higher resolution images need more time to process.

Thick lines

This next problem of the omnicam is a result of its inaccuracy. The omnicam draws thick lines because the distance measurements taken from different angles are not relative to each other. This leads into multiple obstacle points on the map that are close to each other instead of points that overlap with each other like with the accurate laser sensor. These inaccuracies therefore results into a thick line on the map. This problem needs to be solved because navigating through narrow corridors is undesirable if the thick lines are drawn half on the pathway.

The solution for this problem is creating a function that can smooth thick lines. The function would first need to identify the thick lines on the map so it can turn those thick lines into thin lines by taking the average of its points.

Automatic parameter learning

The experiments were set up by using parameters that were derived by hand. Future work would therefore include an automatic parameter learning algorithm. An approach would be to use the training set of the histogram to train the rangefinder parameters.

Further Improvements

The omnicam rangefinder does not work on a slope or other height differences. This is because the pixel to meters formula does not incorporate height differences, this can be seen in figure 4. Further improvements lies in the theory of detecting free space based on color detection. The free-space detection does not work well when the walls or objects all have the same color. This situation might happen literally but having a dark room can also create this situation. The free-space detector might also break down when it has to deal with different lighting conditions, a solution for this problem might be to adapt the color histogram to a HSV color space because this color space can remove color intensity. Future work should also include a method that can detect when the floor color changes.

VII. CONCLUSION

Based on the results found in section 5 it can be concluded that an omnidirectional camera can be used efficiently as a rangefinder. The omnicam rangefinder can have an accuracy of 8.09cm compared to the laser sensor. The results also shows that the omnicam rangefinder can use scanmatching algorithms that were optimized for a laser sensor to create accurate maps of an environment even though it does not possess the precise accuracy of a laser sensor. The omnidirectional camera can also detect obstacles, like an empty cabinet (figure 6a), that a laser sensor can not detect. The omnicam rangefinder is based on color detection and is therefore unusable when the obstacles have exactly the same color as the floor.

This research can lead to the development of small and flying robots that can use the lightweight, energy efficient and inexpensive omnidirectional camera to quickly create a map of an environment. These robots need to work together with a large robot equipped with a laser sensor to provide a color histogram. This color histogram can be learned by the large robot and distributed wirelessly to the small and flying robots.

Acknowledgements

We want to thank Gideon Maillette de Buy Wenniger for providing inside information about the free-space detection algorithm. We would also want to thank Tijn Schmits for his explanation about the specification and precise construction of the OmniP2DX in USARSim.

REFERENCES

[1] S. Carpin, C. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "Usarsim: a robot simulator for research and education," pp. 1400–1405, April 2007, proceedings of the 2007 IEEE Conference on Robotics and Automation.

[2] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "Bridging the gap between simulation and reality in urban search and rescue." in *RoboCup 2006: Robot Soccer World Cup X*, ser. Lecture Notes on Artificial Intelligence, G. Lakemeyer, E. Sklar, D. Sorrenti, and T. Takahashi, Eds., vol. 4434. Springer, October 2007, pp. 1–12.

[3] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney, "The robot that won the darpa grand challenge." *Journal of Field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.

[4] F. W. Rauskolb, K. Berger, C. Lipski, M. Magnor, K. Cornelsen, J. Effertz, T. Form, F. Graefe, S. Ohl, W. Schumacher, Peter, T. Nothdurft, M. Doering, K. Homeier, J. Morgenroth, L. Wolf, C. Basarke, T. Berger, F. Klose, and B. Rumpe, "An autonomously driving vehicle for urban environments." *Journal of Field Robotics*, vol. 25, no. 9, pp. 674–724, 2008.

[5] T. Schmits and A. Visser, "An omnidirectional camera simulation for the usarsim world," in *RoboCup 2008: Robot Soccer World Cup XII*, ser. Lecture Notes on Artificial Intelligence series, vol. 5339. Berlin Heidelberg New York: Springer, June 2009, pp. 296–307.

[6] S. Roebert, T. Schmits, and A. Visser, "Creating a bird-eye view map using an omnidirectional camera," in *Proceedings of the 20th Belgian-Netherlands Conference on Artificial Intell igence (BNAIC 2008)*, A. Nijholt, M. Pantic, M. Poel, and H. Hondorp, Eds., October 2008, pp. 233–240.

[7] G. Maillette de Buy Wenniger and A. Visser, "Identifying free space in a robot bird-eye view," in *Proceedings of the 4th European Conference on Mobile Robots (ECMR 2009)*, September 2009.

[8] D. Scaramuzza and S. Gächter, "Exercise 3: How to build a range finder using an omnidirectional camera," Autonomous Systems Lab Swiss Federal Institute of Technology, Zurich, April 2009, version 1.4.

[9] M. Pfingsthorn, B. Slamet, and A. Visser, "A scalable hybrid multi-robot slam method for highly detailed maps," in *RoboCup 2007: Robot Soccer World Cup XI*, ser. Lecture Notes on Artificial Intelligence, vol. 5001. Springer, July 2009, pp. 457–464.

[10] S. K. Nayar, "Catadioptric omnidirectional camera," in *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 482.

[11] A. Visser, B. A. Slamet, and M. Pfingsthorn, "Robust weighted scan matching with quadtrees," in *Proceedings of the 5th International Workshop on Synthetic Simulation and Robotics to Mitigate Earthquake Disaster (SRMED 2009)*, July 2009.

A USARSim-based Framework for the Development of Robotic Games: An Intruder-pursuit Example

Paul Ng, Damjan Miklic, and Rafael Fierro

Abstract— **This paper presents an example of using USARSim as a testbed for developing and experimenting with Robotic Games. We use the term Robotic Games to refer to a variety of scenarios in which autonomous and remotely controlled robots interact with each other and their environment, according to a predefined set of rules. Such scenarios can be used in robotics education as well as in scientific research. In this work we introduce a framework for facilitating the development and testing of different game algorithms. Our framework is based on the USARSim simulator, which provides a rich, realistic and extensible environment for robot interaction. To demonstrate our approach, we describe an implementation of a pursuit-evasion game, where a group of pursuers needs to locate and capture intruders entering the designated game area.**

I. Introduction

As technology progresses, the capabilities of robots and their usefulness in everyday life is expanding. Industrial robots are already being used extensively, particularly in the automotive industry. An emerging class of robotics is service robots [1]. Service robots can range from small personal devices for vacuum cleaning to large robots for construction or demolition. As robots are finding their way into more and more areas of everyday life, human-robot interaction is becoming an increasingly important topic. In order to be efficient and productive, humans will have to learn to interact and work with these machines. University engineering curricula have already begun to incorporate robotics as a practical application of theory. Additional efforts have been made to introduce younger students to the problems and challenges of controls and robotics through practical experiments, educational programs and robotic games. We use the term robotic games to refer to a variety of scenarios in which autonomous and remotely controlled robots interact with each other and their environment, according to a predefined set of rules. Examples of robotic games include RoboCup soccer [2], RoboFlag [3] and Marco Polo [4]. In addition to their value as an educational tool, robotic games are useful as tools for studying topics such as human-robot interaction, motion planning and cooperative control.

Because the complexity and cost of operating a real multi-robot system can be a limiting factor when designing new scenarios and games, a realistic simulation environment can be an invaluable development tool. A simulation environment should provide realistic 3D rendering and physical simulation and good correspondence with real-world experiments. Furthermore, it should be extensible, portable to various operating systems and enable a transparent transition from simulation to real robot hardware. USARSim is an open-source multi-robot simulator that meets all of the above requirements [5]. It has successfully been used for testing and comparison of control algorithms [6], validation of new robot designs [7] and studying human-robot interaction [8]. Authors in [9] show that simulated robots with realistically modeled dimensions and mass in USARSim exhibit a performance similar to that of real robots. Using the simulator in combination with the Player robot device server enables transparent porting of control programs to robot hardware [10].

In this paper, we extend the work described in [4] to complement the Robotic Games Framework with a realistic and extensible 3D simulator. As our motivating example, we present a simplified version of the pursuit-evasion problem described in [11]. In an environment realistically modeled after the Centennial Engineering Center at the University of New Mexico (UNM), a group of robots must detect and intercept intruders entering the designated surveillance area.

The rest of the paper is organized as follows. In Section II we describe the proposed Robotic Games Framework. Details of the intruder-pursuit scenario that we are considering are given in Section III. Simulation results from the USARSim environment are presented in Section IV. In Section V we give concluding remarks and outline our directions for future work.

II. The Robotic Games Framework

The Robotic Games Framework that we are proposing is a software architecture aimed at facilitating the design, implementation and testing of robotic games and research scenarios. Our goal is to design the system components necessary for game scenario creation and define the interfaces between them. The envisioned framework is based on the work presented in [4].

The main components of the framework and general system architecture are shown in Figure 1. The three main modules are the *robot device module*, the *user interface module* and the *game manager module*.

The *robot device module* encapsulates interfaces to the robotic platforms or "players". In our previous work, these platforms were Evolution Robotics' Scorpion robots and the control algorithms were implemented using the ERSP SDK. To achieve platform and vendor independence, and enable

P. Ng and R. Fierro are with the MARHES Lab, Electrical & Computer Engineering Department, University of New Mexico, Albuquerque, NM 87131-0001, USA. (e-mail: *f*png,rfierro*g*@ece.unm.edu)

When this work was done, D. Miklic was a visiting researcher at the Electrical and Computer Engineering Department, University of New Mexico, Albuquerque, NM 87131-0001, USA. (e-mail: damjan.miklic@fer.hr)

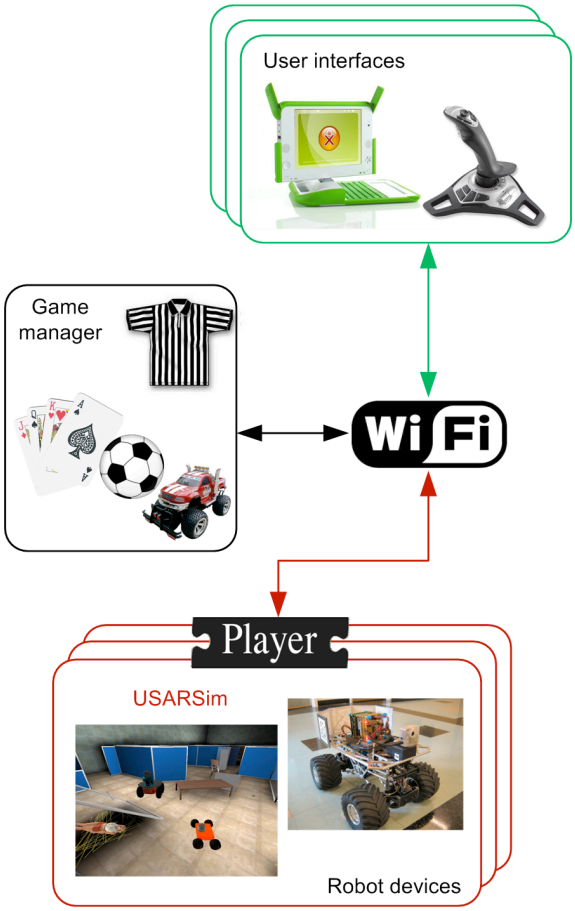


Fig. 1. Main components of the Robotic Games framework.

seamless integration with the USARSim simulation engine, we are basing our new development on the Player robot server [12]. Player is an open-source robot programming framework that has become a de facto standard in the academic community. It features a flexible and extensible architecture, supports a wide variety of robotic hardware, and has interfaces to several robot simulators. Player is also supported by an actively contributing international developer community. By using Player to implement the *robot device module*, we hide all the robot hardware details from the rest of our system. In fact, we can easily switch between simulation and real hardware, without making any changes to the other modules.

The *user interface module* enables human users to interact with the robots. It provides player input through joysticks, gamepads or other devices envisioned by the game designer. An interesting user input method is through hand gesture recognition, which can be implemented using the HandVu library. The *user interface module* also provides gameplay visualization for human players. Visualization includes a simple 2D representation of the environment and relevant game statistics. A possible extension, based on the USARSim simulator, would include a full 3D view of the game environment. Except for players, the *user interface module* also provides referees with visualization and control over the game flow. The module is based on the *Qt Application Framework*, an open-source, cross-platform C++ application and user interface framework.

The *game manager module* loads and configures game scenarios and establishes connections between user interfaces and robot players. It communicates with other system components through a custom message-based protocol built on top of the standard UDP/IP and TCP/IP networking protocols. In order to interact with game implementations, the *game manager module* requires that every game support at least the following messages:

- Load Game
- Start Game
- Stop Game
- Pause Game
- Query Status

At the beginning of the game, the *game manager module* loads the appropriate game. The *robot device module* establishes connections to the robots that are going to take part in the game and performs any necessary initialization procedures. After the game has been loaded, *user interface* clients can send requests to the *game manager module* in order to get connections to the robots. If the request is approved by the *game manager module*, the client can establish a direct connection to the robot through the *robot device module*. Once the necessary conditions have been met, the *game manager module* can start the game. To minimize communication overhead, human players maintain a direct connection to their robotic counterparts during the course of the game.

III. The Intruder-Pursuit Game

The framework described in the previous section is being implemented, based on the existing system described in [4]. Currently, we have developed prototypes of the *game manager module* and *robot device modules* and tested them in a simple, fully automated (without human interaction) pursuit-evasion scenario described in the following section. At this point these modules are not yet integrated with the existing *user interface module*.

A. Game Description

The game we have implemented is a simple intruder detection and capture scenario. Three mobile robots equipped with range sensors are designated as pursuers. They are initially placed within a bounded space called the surveillance area. The two robots playing the role of intruders start the game outside of the surveillance area. At random time intervals, the intruders enter the surveillance area moving along straight line paths. The pursuers are required to make two independent intruder detections before being dispatched to capture. An intruder is considered to be captured when a pursuer has intercepted its path at a distance less than one meter. The intercepting pursuer can then continue monitoring the surveillance area.

To make game simulations more realistic, we have built an accurate model of the UNM Centennial Engineering Center

Fig. 2. Photo of the Centennial Engineering Center courtyard at UNM.

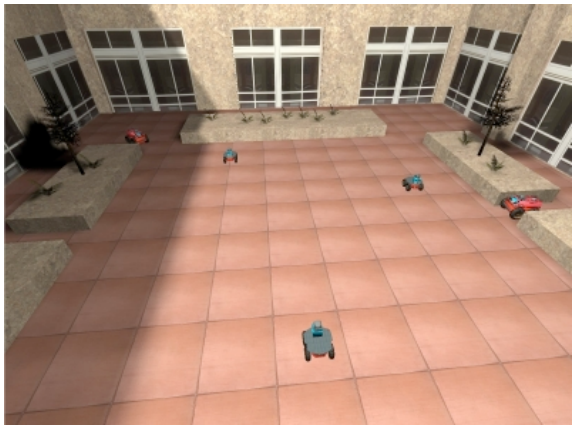


Fig. 3. USARSim model of the Centennial Engineering Center courtyard.

using the Unreal Editor 2.5. On Figures 2 and 3, a photo taken at the actual building is shown above a snapshot of the model. In our scenario, the surveillance area is restricted to the space between the landscape areas in the courtyard. The floor plan of the surveillance area is shown in Figure 4.

B. Detection and Capture Algorithms

Vehicle models used in the simulations are MobileRobots' Pioneer2-AT as pursuers and iRobot's ATRV-Jr as intruders. Detailed mechanical and physical models of both platforms are available in the USARSim distribution. For the purposes of our control algorithms, pursuer vehicle kinematics can be described by the unicycle model:

$$\begin{aligned} \dot{x}_p &= v_p \cos \theta_p; \\ \dot{y}_p &= v_p \sin \theta_p; \\ \dot{\theta}_p &= \omega_p; \end{aligned} \tag{1}$$

where pursuer input is linear and rotational velocity, $u_p = [v_p \quad \omega_p]^T$. The kinematic model of the intruders is given by

$$\begin{aligned} \dot{x}_i &= v_i \cos \theta_i(0); \\ \dot{y}_i &= v_i \sin \theta_i(0); \end{aligned} \tag{2}$$

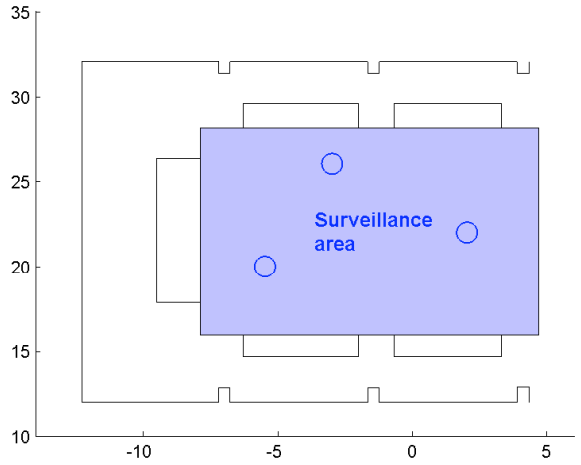


Fig. 4. Surveillance area. Pursuer initial positions are denoted with blue circles.

reflecting the fact that intruder motion is restricted to straight lines.

Pursuers are relying on laser range finders for intruder detection. Generally speaking, video cameras would be a better option; however, the Player robot server currently lacks a driver for accessing video from USARSim. To be able to detect intruders using only laser range finders, we must assume that the pursuers have full information about their static environment and about each other. Under this assumption, each robot provides a pair of coordinates $(x_i; y_i)$ for every intruder detection made by its laser scanner. By taking into account all the detections made by all robots we can write

$$\mathbf{Y} = \mathbf{X} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_i & 1 \\ \vdots & \vdots \end{bmatrix}; \tag{3}$$

where $\mathbf{Y} = [y_1 \cdots y_i \cdots]^T$. Then, by linear least squares fitting, we can estimate the slope and intercept of the intruder path as

$$\begin{bmatrix} a \\ b \end{bmatrix} = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{Y}. \tag{4}$$

In our software implementation, we used the CGAL [13] C++ libraries for least squares fitting.

Once the intruder path has been estimated from two independent sets of readings by two different robots, a pursuer is dispatched for interception. To find the capture point, we search for a point on the intruder's estimated path for which the equation

$$\frac{d_p}{v_p} = k \frac{d_i}{v_i}; \quad 0 < k \leq 1 \tag{5}$$

holds. Maximum velocities of the intruder and pursuer are denoted by v_p and v_i. In other words, we are looking for a point on the intruder's path which can be reached $1/k$ times faster by the pursuer than the intruder. Equation 5 can be solved from geometric conditions depicted in Figure 5. We

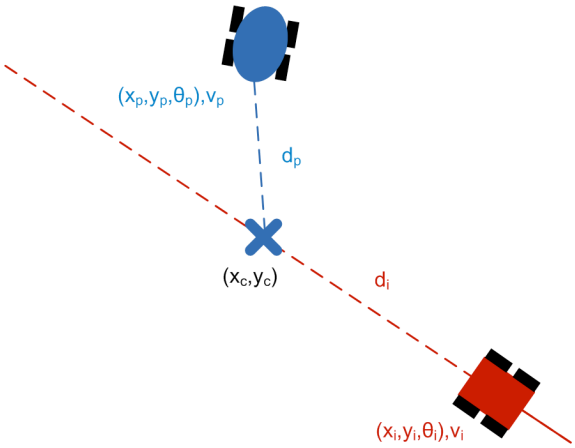


Fig. 5. Intruder interception.

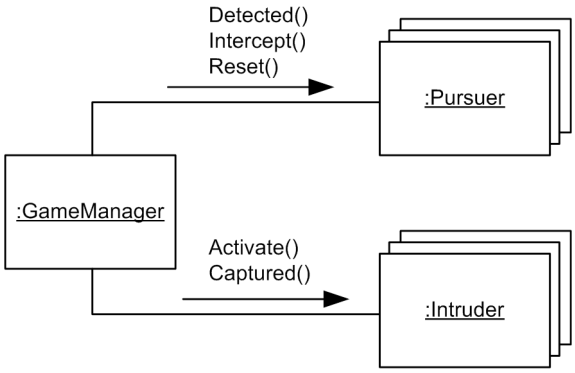


Fig. 6. Communication diagram between program components.

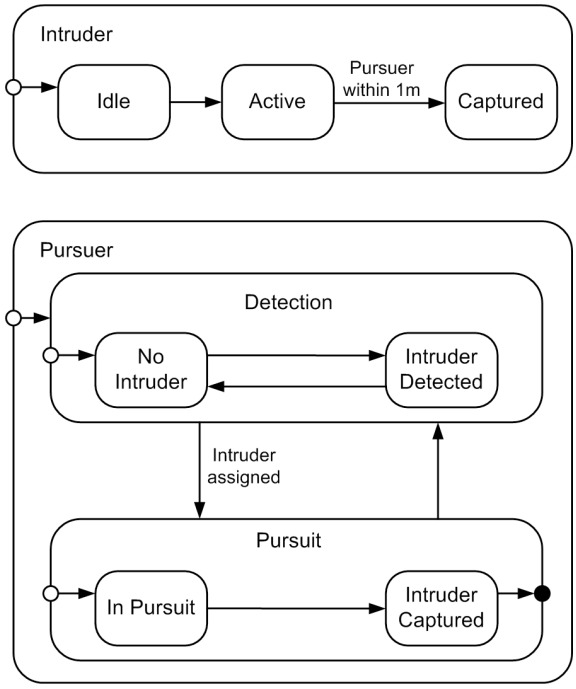


Fig. 7. Finite state machine model of intruder and pursuer behavior.

Velocites and Coefficents	
Pursuer Velocity	0.25 m/s
Intruder Velocity	0.1 m/s
k	0.3
Laser Scanning Range	10 Degrees Centered at 0

TABLE I
PARAMETERS OF THE CAPTURE ALGORITHM

are using a feedback linearization [14] to drive the pursuer to the capture point. From the kinematic model (2), robot inputs are computed as

$$\begin{bmatrix} v_p \\ \omega_p \end{bmatrix} = \begin{bmatrix} \cos(\theta_p) & \sin(\theta_p) \\ \frac{\sin(\theta_p)}{d} & \frac{\cos(\theta_p)}{d} \end{bmatrix} \begin{bmatrix} x_c \\ y_c \end{bmatrix}; \quad (6)$$

where x_c and y_c denote the desired capture point. The quantity d is the distance from the robot's center of mass to the controlled off-axis point, introduced to avoid singularities in the solution.

C. Software Implementation

The software implementation of the game can be broken down into three classes, corresponding to the *game manager module* and *robot device module* from Figure 1. An object of the *GameManager* class runs the game and interacts with *Intruder* and *Pursuer* objects as depicted in the UML communication diagram [15] in Figure 6.

Intruder and *Pursuer* behavior can be modeled by two parallel finite state machines, as depicted in figure 7. At the start of the game, *GameManager* object connects to all the robots and initializes pursuers to the "No Intruder" state and intruders to the "Idle" state. It then continuously polls *Pursuer* objects for intruder detections. At random time intervals it activates one of the intruders. After the pursuers have collected two independent intruder sightings, it dispatches the closest pursuer to capture. When the pursuer approaches the intruder to one meter intercepting its path, the *GameManager* switches intruder state to "Captured" and resets the pursuer to "Detection" mode. After all intruders have been captured, the *GameManager* ends the game.

IV. Simulations in USARSim

The following simulation is implemented on an Apple Macbook Pro 3.1 running Ubuntu 9.04. Unreal Tournament 2004 is installed with the 3369.1 Linux Patch applied and USARSim version 3.37 is used. The controller is implemented through Player robot server Subversion revision 7945 and the USARSim Player drivers are downloaded from Stefan Stiene [16].

The experiment is set up as an intruder-pursuit game. There are three Pioneer2-AT robots (P2AT) as pursuers, each equipped with a Sick Laser Scanner LMS200. The intruders are represented by the two iRobot ATRV-Jr robots (ATRVJr). Initial positions are denoted by circles in Figure 8 and are presented in tabular form in Table II. The experiment starts when the *GameManager* initializes each robot. The velocities and coefficients for the capture algorithms are shown in Table I.

The pursuers begin by recording the initial ranges detected by their laser scanner. The environment ranges could be a

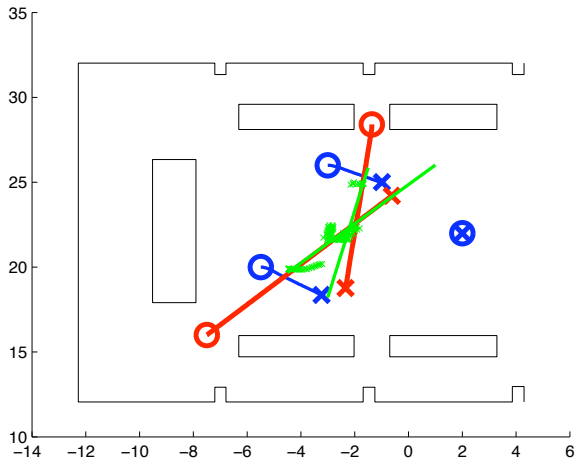


Fig. 8. Experiment data.

Name	X	Y	Theta
Pursuer 1	-7.00	-20.00	0.00 ž
Pursuer 2	3.00	-22.00	180.00 ž
Pursuer 3	-3.00	-26.00	0.00 ž
Intruder 1	-7.50	-16.00	60.00 ž
Intruder 2	-1.30	-28.40	105.00 ž

TABLE II

PURSUER AND INTRUDER STARTING POSITIONS AND YAW

detection of a wall or another pursuer, but the environment is assumed to be static. After the environment snapshot has been recorded, pursuers begin scanning for intruders.

After the pursuers have finished recording their environments, the *GameManager* tells one intruder to begin its path. Since we restrict intruder motion to following a straight line, the robot is initially placed on its path and rotated to the correct bearing. In Figure 9, an intruder moves into the laser scanning range of a pursuer, the laser readings are recorded and positive detection status is sent back to the *GameManager*. At this point the *GameManager* interpolates a capture point. This capture point is sent to the third pursuer who is then dispatched to capture the intruder. A successful capture is shown in Figure 10.

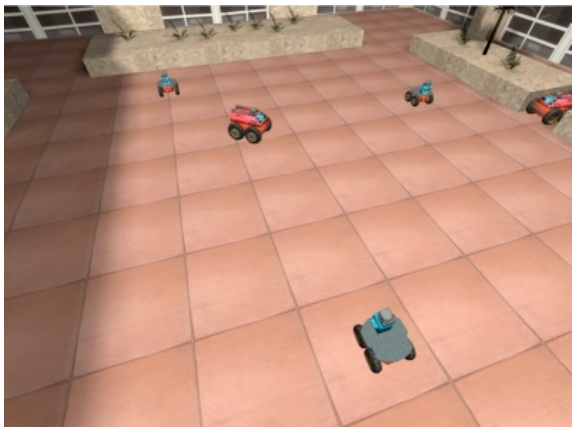


Fig. 9. The first intruder is detected.

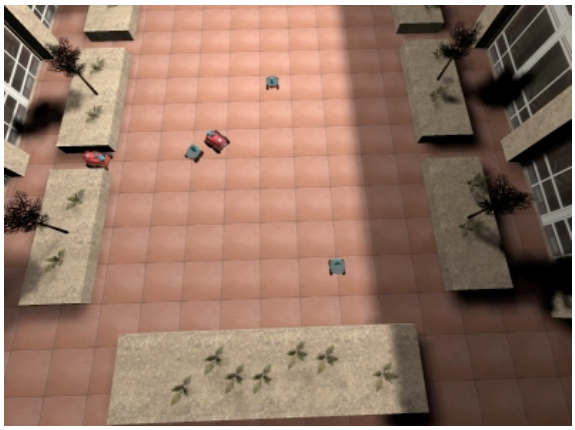


Fig. 10. The first intruder is intercepted and captured.

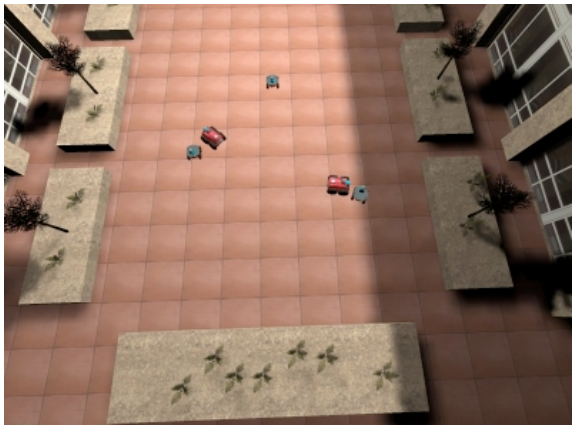


Fig. 11. Both intruders are captured.

Before the next intruder is activated, the *GameManager* repositions all pursuers to adequately scan the environment and resets their laser scans of the static environment. The *GameManager* repeatedly repositions and resets the pursuers until all the intruders are captured. In our specific game, the capture of the second intruder denotes the end of the game as shown in Figure 11.

Data recorded during the simulation run is plotted in Figure 8. The pursuers and intruders are prompted by the *GameManager* to log their location and intruder detection data to file. The files are compiled together later in MATLAB for the plot. Intruder paths are shown in red and pursuer paths in blue. Initial positions are denoted by circles and final position by crosses. Laser detection data and interpolated pursuer paths are plotted green.

We are currently looking at ways to obtain video camera images from the simulated robots. At the time of this writing, the USARSim team has developed an image server capable of recording images during the simulation. It seems like a camera driver for Player could be implemented based on the USARSim image server and this could be furthered as a video input which allows us to distinguish simultaneous intruders. Also, since the simulation is centralized, the current experiment is limited by the machine's computational

performance. We have noticed that the experiments could, at most, have twelve robots on the field. The Karma Engine fails to correctly render the thirteenth robot and the experiment will not begin.

V. CONCLUSIONS AND FUTURE WORK

We have presented a framework for developing and testing different robotic scenarios that we call Robotic Games Framework. These scenarios have applicability to education as well as research. The described framework is based on the Player robot server and USARSim simulator to enable fast prototyping and testing with a smooth transition from simulation to real-world experiments. Software flexibility and re-use are promoted by decoupling the framework into three distinct modules.

We have described a prototype implementation of a simple, fully automated intruder-pursuit game, based on the proposed framework. Prototypes of the *game manager module* and *robot device module* have been implemented as C++ objects. We have tested the game scenario with five robots in USARSim simulations within a realistically modeled environment. Simulations have enabled us to test, improve and validate our code much faster and more conveniently than would have been possible if experimenting on real robotic platforms.

The work presented in this paper is just a starting point for further research. First of all, implementing the USARSim image server to work with the camera drivers in Player would greatly help us introduce computer vision. Using cameras instead of (or in combination with) laser range finders would enable more robust intruder detection as well as simultaneous detection of multiple intruders. Also, to improve scalability, the framework could be distributed among several computers; dedicating the USARSim server, Player robot server, and each robot device to their own respective machine.

Results from [17] could be applied to the scenario, providing certain optimality guarantees to sensor area coverage. The intruder's linear path assumption might be appropriate in several applications. Dropping the restrictive assumptions on intruder paths would open up challenging problems of trajectory prediction and the possibility of including human players for intruder control. The *game manager module* and the *robot device module* would integrate with the *user interface module* as described in [4].

Finally, at the time of this writing, the framework is still under development. Adding recursive execution of different game scenarios and robot configurations can enable batch mode to test the framework's robustness. These game scenarios could include the Marco Polo game [18], Lion and Man game [19] and others. The framework currently has a simple error handling algorithm, which only checks if the distance between the pursuer and intruder increases during a pursuit. A more elaborate error handling can identify a more comprehensive list of errors and the *game manager module* would act accordingly. The experiments should also be conducted using real-world robots to verify and validate the simulation results.

ACKNOWLEDGEMENTS

This work is supported by NSF grants ECCS CAREER #0811347, IIS #0812338, CNS #0709329, and DOE URPR (University Research Program in Robotics) grant DE-FG52-04NA25590. The work of D. Miklic at the University of New Mexico was supported by the U.S. Bureau of Educational and Cultural Affairs through the Fulbright Program.

REFERENCES

[1] K. W. Lee, H.-R. Kim, W. C. Yoon, Y.-S. Yoon, and D.-S. Kwon, "Designing a human-robot interaction framework for home service robot," in *Robot and Human Interactive Communication, 2005. ROMAN 2005. IEEE International Workshop on*, Aug. 2005, pp. 286–293.

[2] G. Lakemeyer, E. Sklar, D. Sorrenti, and T. Takahashi, *RoboCup 2006: Robot Soccer World Cup X*. Springer-Verlag New York Inc, 2007.

[3] R. D'Andrea and R. Murray, "The RoboFlag Competition," in *Proc. of the American Controls Conference*, June 2003, pp. 650–655.

[4] B. Perteet, J. McClintock, and R. Fierro, "A multi-vehicle framework for the development of robotic games: The marco polo case," in *Robotics and Automation, 2007 IEEE International Conference on*, April 2007, pp. 3717–3722.

[5] S. Balakirsky, C. Scrapper, S. Carpin, and M. Lewis, "UsarSim: Providing a Framework for Multirobot Performance Evaluation," in *Performance Metrics for Intelligent Systems Workshop, PerMIS*, Gaithersburg, MD, USA, 2006.

[6] B. Balaguer, S. Carpin, and S. Balakirsky, "Towards Quantitative Comparisons of Robot Algorithms: Experiences with SLAM in Simulation and Real World Systems," in *IROS 2007 Workshop*, 2007.

[7] B. Taylor, S. Balakirsky, E. Messina, and R. Quinn, "Analysis and benchmarking of a whegs robot in usarsim," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, Sept. 2008, pp. 3896–3901.

[8] M. Lewis, J. Wang, and S. Hughes, "USARSim: Simulation for the Study of Human-Robot Interaction," *Journal of Cognitive Engineering and Decision Making*, vol. 1, no. 1, pp. 98–120, 2007.

[9] S. Okamoto, K. Kurose, S. Saga, K. Ohno, and S. Tadokoro, "Validation of simulated robots with realistically modeled dimensions and mass in usarsim," in *Safety, Security and Rescue Robotics, 2008. SSRR 2008. IEEE International Workshop on*, Oct. 2008, pp. 77–82.

[10] B. Gerkey, R. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, 2003, pp. 317–323.

[11] T. G. McGee and J. K. Hedrick, "Guaranteed strategies to search for mobile evaders in the plane," in *Proc. American Control Conf.*, Minneapolis, MN, June 14-16 2006, pp. 2819–2824.

[12] T. Collett, B. MacDonald, and B. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*, 2005.

[13] "CGAL, Computational Geometry Algorithms Library," Online, http://www.cgal.org.

[14] H. Khalil, *Nonlinear Systems*, 3rd ed. Prentice Hall, 2001.

[15] S. Ambler, *The object primer 3rd edition: Agile model driven development with UML 2*. Cambridge University Press, 2004.

[16] S. Stiene, "Usarsim player driver," Online, Jan. 2009, http://www.informatik.uni-osnabrueck.de/sstiene/unreal/usarsim.tar.gz.

[17] S. Ferrari, R. Fierro, B. Perteet, C. Cai, and K. Baumgartner, "A geometric optimization approach to detecting and intercepting dynamic targets using a mobile sensor network," *SIAM Journal on Control and Optimization*, vol. 48, no. 1, pp. 292–320, 2009. [Online]. Available: http://link.aip.org/link/?SJC/48/292/1

[18] Wikipedia, "Marco polo (game)," Online, 2006, http://en.wikipedia.org/wiki/Marco_Polo_(game).

[19] N. Karnad and V. Isler, "Lion and man game in the presence of a circular obstacle," in *A technical report accepted to Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, 2009.

Creating High Quality Interactive Simulations Using MATLAB® and USARSim

Allison Mathis, Kingsley Fregene and Brian Satterfield

Abstract—**MATLAB® and Simulink®, useful tools for modeling and simulation of a wide variety of dynamic systems, lack a high quality integrated visualization environment with realistic rendering of real-world effects. We describe a methodology to interface these tools with USARSim and Unreal Tournament® to create an easy to use simulator with the ability to represent high-fidelity (vehicle) dynamic models, provide feature-rich interactive graphics capabilities as well as support data passing for a variety of sensor types. Our approach takes a preexisting physical simulator with all the physics and dynamics modeled (something MATLAB®/ Simulink® excels at) and adding a world for entities in the simulator to explore (something USARSim and Unreal Tournament® work well for). This results in an integrated simulator that is straightforward to implement.**

I. Introduction

IN this paper we combine MATLAB®, Simulink®, USARSim, and Unreal Tournament® into a simulation and visualization tool for the Samarai project at Lockheed Martin Advanced Technology Laboratories. Samarai [9] is a nano-class (an unmanned aerial vehicles (UAV) approximately no larger than 10 cm) monowing UAV designed to be operated manually or autonomously and controlled by the modulation of a trailing edge wing flap and the speed of rotation. These vehicles exhibit complicated dynamic behavior due to the interaction of rotary wing aerodynamics with mechanisms for generation of forces and moments at very small scales. Creating a high-fidelity simulator for this system involves modeling complex aerodynamic effects and capturing multitudes of parameters that specify various physical properties of the vehicle. The simulator and a significant portion of the operational code were written in MATLAB®/Simulink®, and are being updated as development continues.

The other half of the simulator was written using USARSim [7] and Unreal Tournament® (UT) [8] to test an optical flow algorithm that estimates ground speed, vehicle attitude rates, and provides collision alerts. A user can interface with the simulator via a joystick and explore the environment while collecting data for the algorithm in real time. USARSim was chosen for the image quality produced by UT, and because it is well documented, open source software with an active development community.

MATLAB® and Simulink®, both produced by The Mathworks™ [6], are a commonly available development platform for many different types of systems. UT, produced by Epic Games™, is a computer game with a high quality graphics engine and a world builder, allowing the user to design any sort of environment within the Unreal Editor®. However, UT is proprietary and was not designed with the research community in mind. In order to access its features, one must use USARSim, currently maintained by the National Institute of Standards and Technology (NIST), a system that integrates with UT to allow two way communications between the agents or bots, acting within UT and the research software.

The two portions of the Samarai simulation are individually useful and become even more so when combined. The collision detection becomes more accurate when the collected data is based on the current physical abilities of the prototype, while the physical simulator gains through ease of error checking. Although data on how well Samarai is following its commands may be easily produced in graph form, it is more intuitive to watch it complete a maneuver. The combination of MATLAB®/Simulink® with UT and USARSim allows data to be viewed in any manner at any time, providing flexibility and control to the researcher.

II. Related Work

There have been a number of prior interfaces between USARSim and other simulation programs, such as the Mobility Open Architecture Simulation and Tools (MOAST), Pyro©, and Player. MOAST, created at NIST to complement USARSim, is a four dimensional environment for running and analyzing multi-agent simulations. It allows the user to test under specified conditions and test algorithms prior to real world deployment [1]. Like MOAST, Pyro© is also designed to abstract away the hardware to allow the user to interact with various types of robots without having to worry about the peculiarities of the various platforms. It supports a variety of commercially available robots and can take algorithms written for simulation and translate them into the appropriate format [5]. Player, a robot control program,

Manuscript received July 17, 2009. Support provided for this work under a Lockheed Martin Internal Research and Development project is gratefully acknowledged.

All authors are with Lockheed Martin Advanced Technology Laboratories, Cherry Hill, NJ, 08002, USA
Allison Mathis (email: amathis@atl.lmco.com)
Kingsley Fregene (email: kfregene@atl.lmco.com)
Brian Satterfield (email: bsatterf@atl.lmco.com)

with its simulator, Stage, is another program with a number of robots and components pre-built and ready for use in testing. Programs may be written in any language, and there is no set structure. This allows the user a great deal of freedom in designing simulations [2].

Although The Mathworks™ offers the Virtual Reality (VR) toolbox, it is not suitable for robot simulation applications due to its poor image quality and the lack of available feedback. As may be seen in Figure 1, UT presents a more realistic environment, which aids work with algorithms such as optical flow, as well as providing a more interesting experience for the user. Unlike USARSim, which keeps track of the objects in the environment and can provide simulated sensor readings, VR toolbox is only as a visualizer, and is a passive tool.

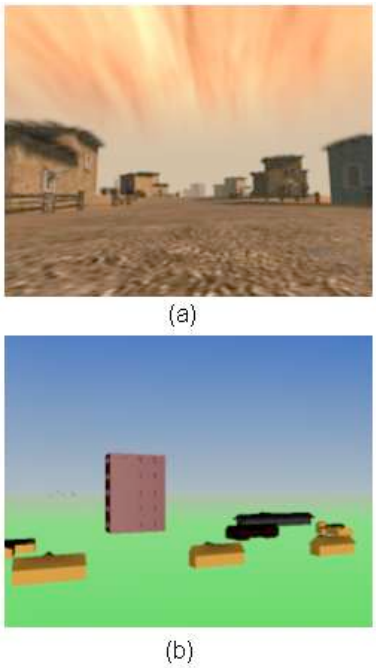


Fig. 1. Comparison of scenes from UT (a) and the VR toolbox (b).

In many prior works that used high quality visualizations, the vehicles navigating in the environment were represented by very simplistic models that may not adequately capture the vehicle's full behavior at a sufficient level of fidelity. Our system offers a high quality interactive visualization of entities/environments driven by high fidelity models of the vehicles operating in these environments—all in the same system.

In the UAV context, full vehicle dynamics (including aerodynamic and propulsion data obtained from wind tunnel experiments) are easily modeled in MATLAB®/Simulink®. When this is connected to USARSim/UT, it results in a truly powerful design, training and visualization tool for a variety of applications.

Another advantage of the MATLAB® interface is the ease with which preexisting systems may be integrated with the USARSim environment. Code does not need to be reformatted or otherwise altered – a few lines added, an inexpensive game, a free download and the system is operational and ready to run.

Given the advantages, we expect that many teams currently using either Simulink® or MATLAB® will decide that our approach will allow them to create interactive visualizations, with realistic underlying vehicle dynamics, quickly and easily.

III. Implementation and Architecture

The work presented here was done using MATLAB® 2007b, and requires The Mathworks'™ Instrument Control Toolbox (ICT), Unreal Tournament® 2004 and the corresponding version of USARSim. The Samarai simulation which has been integrated with USARSim and UT is written in Simulink®, however interfaces for both Simulink® and MATLAB® have been included in this paper, as it is thought that they are equally useful.

MATLAB® and Simulink®, while complementary and easily integrated with each other, require different coding structures. Accordingly, the following discussion has been broken down into two main sections, with a short additional portion that describes how to create a hybrid of all three.

To aid the following explanation, we assume a scenario in which a user wishes to send a drive command to the bot that will result in forward motion. As depicted in Figure 2, this is done by generating a velocity command, formatting that command into a form accepted by USARSim, sending the command, and then receiving the results, as shown by an updated visualization and sensor data returned to the user. While simplistic, the block diagram is an accurate representation of the steps involved in the integrated system.

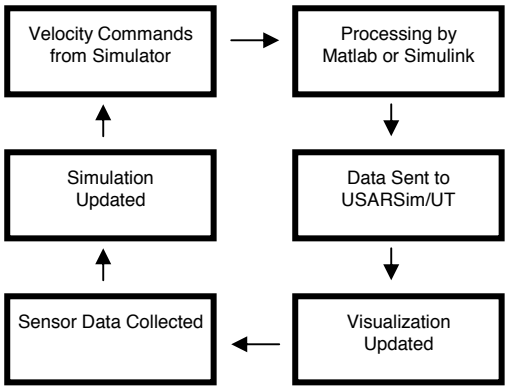


Fig. 2. Block diagram of scenario full system path.

The simulation in Figure 2 refers to the entire system within MATLAB® and Simulink® which represents the vehicle dynamics and input responses, with the exception of the portion that processes the commands to send to USARSim/UT. As mentioned in the introduction, it is a highly complex and project specific model and is outside the scope of this paper. The processing of the commands by MATLAB® or Simulink®, as well as the method of sending

the data to USARSim/UT is discussed in the next few sections.

The visualization updates as well as the collection of the sensor data are both automatically performed by USARSim/UT. For instructions on accessing various types of sensor data, see the section on sensors in [10]. Any sensors attached to the bot in USARSim will automatically pass back information, which may be collected using the "fscanf" command on the TCPIP connection, as shown in the Checking For Message Data section below. For the purpose of the Samarai project the data returned to the simulator is solely images produced by the internal UT "camera" rather than the more usual range or odometry information. As such, this data is passed to MATLAB® and Simulink® through different channels. The USARSim project offers an image server to collect and return the data to the user, but in its current form it does not interface as anticipated with MATLAB®/Simulink®. While a rudimentary method of collecting and sending the images has been implemented, the final version is still in progress and will be shared when complete.

A. MATLAB® to USARSim/UT Interface

The MATLAB® interface is straight forward. First, one creates a TCPIP connection to the proper port, and sets the terminator to carriage return / line feed. Because using other terminators will cause message parsing to fail, it is imperative in this scheme to either set the type in the script (recommended) or with the dialog box produced by the inspect(*connectionName*) command.

The standard method of putting variables into a print statement does not work with this interface. Commands to the bot, such as how fast to drive, are sent by "printing" to the TCPIP connection. However, when conversion specifiers and variables are used, it is printed as is, and cannot be properly parsed. Instead, the commands must be assembled as a string, with each variable converted to string form and concatenated in the correct order. An example of the required formatting may be found in the "Commanding the Bot" section of the sample code.

SAMPLE CODE:

SETTING UP CONNECTION
```
Connection2UT = tcpip('localhost', 3000)
```

SETTING THE PROPER TERMINATOR
```
set(Connection2UT, 'Terminator',
{'CR/LF','CR/LF'});
```

OPENING A CONNECTION
```
fopen(Connection2UT);
```

CHECKING FOR MESSAGES OR DATA
```
fscanf(Connection2UT);
or
ConStat = get(Connection2UT,'Status');
```

INSTANTIATING A BOT
```
fprintf(Connection2UT,'INIT {ClassName
USARBot.bot_Type}{Location 0,-50,-
20}{Name bot_Name }');
```

COMMANDING THE BOT
```
s1 = 'Drive {Name bot_Name}
{LinearVelocity ';
s2 = int2str(variable1);
s3 = '}{LateralVelocity ';
s4 = int2str(variable2);
s5 = '}{AltitudeVelocity ';
s6 = int2str(variable3);
s7 = '}{RotationalVelocity ';
s8 = int2str(variable4);
s9 = '}';

CommandString =
[s1,s2,s3,s4,s5,s6,s7,s8,s9];
fprintf(Connection2UT,CommandString)
```

B. Simulink® to USARSim/UT Interface

There is no Simulink® block to connect to USARSim, so an S-function must be used in its place. S-functions provide a way to embed a script inside a Simulink® block and are used where the functionality provided by the usual block diagrams are either inadequate or would be too time-consuming to implement. The script may appear complex, but by following several rules outlined at [11], the distinctive syntax becomes familiar. S-functions can be written in almost any language, but it was found that the level 2 m-file type is the easiest to use, as it is written in native MATLAB® script and can use the built-in editor. It also gives the user control over the type and dimensionality of the inputs and outputs. The code that goes within the S-function is identical to the MATLAB® interface, however due to the iterative nature of Simulink®, it has a few extra peculiarities.

Ordinarily, Simulink® will reset all variables inside a block each time it is called. This will result in the simulation losing track of where the messages are supposed to go, even though the connection is still open, thus, the connection variable must be set to "persistent." Similarly, it is important to name the bot and to use that name when sending commands, otherwise commands will not be paired to the correct entity.

To ensure that each bot is only instantiated once, it is necessary to set a flag to be checked by an IF statement within the S-function. The "Stop Instantiating" flag should be set after the initial iteration to prevent any more instantatiation of bots (in one test, it was observed that neglecting this step resulted in almost instantaneous creation of 8 bots, which overloaded the system and caused it to crash!).

The default update rate of Simulink® is faster than UT can handle, and every message sent will be stored in a queue, resulting in a lag of a few minutes between a user's command and its execution. This may be resolved by setting the simulator step size to an appropriate number in the

Simulink® configuration parameters or, in the case of the drive command, by down-sampling the velocities being generated by the controller to produced real-time execution and visualization.

C. MATLAB®/Simulink® to USARSim/UT Interface

Linking Simulink® and MATLAB® for simulation purposes is identical to linking them normally. The two options are to use the "to workspace" block or the "assignin" command to get the data into memory where it can be generally accessed, gathered up by MATLAB® and then dealt with in the standard MATLAB® manner.

D. Logging Data in USARSim to Aid Debugging

While the setup of USARSim and its attachment to UT is not within the scope of this paper, there is one step which can be useful for debugging the interface with MATLAB®/ Simulink®. When creating the batch file to start UT, include the line

```
-log=usar_server.log
```

that will produce a real time log of messages received and actions taken, such as that shown in Figure 3.

SAMPLE BATCH FILE CODE:

```
Start C:\UT2004\System\ut2004
DMNAV2?game=USARBot.USARDeathmatch?
spectatoronly=1?TimeLimit=0?quickstart=t
rue -ini=usarsim.ini
-log=usar_server.log
```

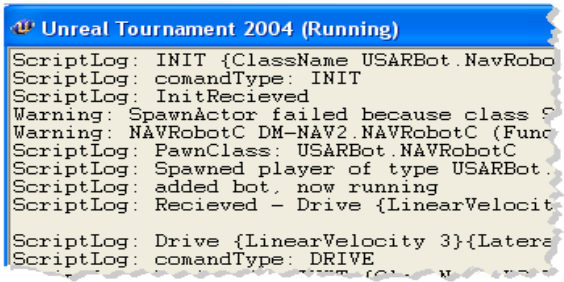


Fig. 3. UT Runtime Log.

For more details on how to set up a batch file and why this is useful, see the section on running the simulator in the USARSim manual [10].

If a command has been properly received and parsed it will be shown in the log (INIT {ClassName..), followed by the program acknowledging what sort of command it was (commandType: INIT) and then that is has been received (InitRecieved), as shown in the first three lines of Figure 3. If the log does not show this, check to make sure the terminators have been properly set and that the syntax of the command is exactly, space by space, that of the examples in the USARSim manual [10].

E. Matlab to Image Server Interface

USARSim does not currently include a camera-type sensor which can return data for manipulation, instead, images must be collected using an image sever which connects directly to UT and USARSim. Carnegie Mellon's Intelligent Software Agents laboratory has released a user friendly image server, UPIS, which may be downloaded at [12]. Unlike prior image servers, UPIS is run from the batch file, and can be fully incorporated into the USARSim side of the system, simplifying system start-up and runs.

SAMPLE BATCH FILE CODE:

```
upis.exe -l
"C:\ut2004\System\ut2004.exe" "DM-
NAV2?spectatoronly=1?game=USARBot.USARDe
athMatch?TimeLimit=0?quickstart=true -
ini=usarsim.ini"
```

UPIS is capable of sending images of varying quality; for more information on how these are set in the batch file, see the interface description portion of [12].

While the connection between MATLAB® and UPIS is also conducted via sockets, unlike the previous connection this one uses java sockets to access the data stream that would otherwise overload the TCIP protocol.

MATLAB® allows a form of java to be used within the m-files, and to be interspersed with the MATLAB® code – however it is not standard java, but rather a sort of hybrid of MATLAB® and java syntax and commands that is not terribly intuitive. For more information, see the documentation in [13].

MATLAB® does not recognize the primitive data type required for the java command "readBytes", forcing the use of "readByte" in its place. This makes real time integrated use of the image server impossible for some applications, as approximately 30 seconds are required to read in an image. As a result, an alternate method of collecting image data for immediate use in MATLAB® has been developed, and will be referred to as the Fast Unintegrated Image Collection (FUIC), as opposed to the Image Server, which refers to the integration with UPIS. FUIC takes a partial screen shot [14] bounded by coordinates chosen by the user. As such, the UT must always be open, on top and remain in the same place during the entire run.

IMAGE SERVER SAMPLE CODE:

ACCESSING JAVA IN MATLAB®

```
import java.net.Socket
import java.io.*
```

SETTING UP CONNECTION

```
input_socket = Socket('localhost', 5003);
```

RECEIVING DATA

```
input_stream = input_socket.getInputStream;
d_input_stream = DataInputStream(input_stream);
```

GETTING IMAGE INFORMATION

```
imageType = d_input_stream.readUnsignedByte();
imageSize = d_input_stream.readInt();
bytes_available = d_input_stream.available;
```

READING IN AN INDIVIDUAL BYTE

```
aByte = d_input_stream.readByte;
if sign(aByte) == -1
     aByte = - aByte +128;
end
```

IMPLEMENTATION NOTES

The byte stream should be read as a unit8s, and there should be a brief pause between reading the image size and checking the number of bytes still available on the stream.

There are a number of jpeg decoders available on the MATLAB® Central Exchange, however they were not immediately applicable and time was not available to modify them as needed for this application.

FUIC SAMPLE CODE:

RECORDING THE POSITION OF THE MOUSE

```
aCorner = get(0,'PointerLocation');
```

TAKING A SCREEN SHOT

```
[ScreenShot,colormap]=getscreen(coordinates);
```

IV. RESULTS

The current version of the integrated simulator uses the Simulink® model of the Samarai dynamics and responses, produces velocity commands, sends them into USARSim/UT and controls the motion of the bot within the simulation environment. Image data is collected and stored by a separate MATLAB® program. Another program to collect image data is being developed to work with the USARSim Image Server produced by Carnegie Mellon.

The integrated simulator allows the user to switch almost instantaneously from a simulation environment to the real world by simply removing a single block (Simulink®) or commenting out a single function (MATLAB®). There are no limits to the types of scenarios that may be modeled, nor to the robots which may inhabit them, as anything which can be mathematically modeled is possible. Due to the ubiquity of MATLAB®, one may easily share code with fellow researchers who are not using USARSim, as well as adapting prior work for simulation with little more than cut and paste.

V. CONCLUSION

While many possible applications exist for the integrated simulator, the two most useful to date for the Samarai project are the updated optic-flow based obstacle detection/avoidance and the ability to demonstrate the system to its potential user community. Development and tuning of optical flow algorithms are significantly aided by the accurate modeling of the physical capabilities of the platform in MATLAB®® and the richness of visual data produced by the image server of Unreal Tournament. When demonstrating Samarai to potential users, having an interface that allows users to "fly" around an environment gives them a realistic experience than charts, graphs or video clips.

The interface between MATLAB® and the UPIS image server is still a work in progress. While time was not available to continue with it in order to overcome the difficulties involved in the MATLAB®/Java interface prior to publication, we anticipate an alternate method will be found, and the reader should either look forward to sharing or hearing about it.

Overall, we find the interface between MATLAB®/ Simulink®, USARSim and UT to be easy to set up and use. It will simplify the simulation process for teams coding robots in either MATLAB® or Simulink®, as well as adding a great deal of functionality, such as sensor feedback in a complex environment, that would not be available.

ACKNOWLEDGMENT

We would like to thank The Mathworks™ for their assistance with this project, as well as Tom Moore of ATL and Prasanna Velagapudi of CMU for their thoughts on working with UPIS.

REFERENCES

[1] C. Scrapper, S. Balakirsky, and E. Messina "MOAST and USARSim - A Combined Framework for the Development and Testing of Autonomous Systems," in Proc. SPIE Defense and Security Symposium, Orlando, 2006.
[2] "Player/stage project," http://playerstage.sourceforge.net, 2005.
[3] "Gamebots," http://gamebots.planetunreal.gamespy.com/, 2009.
[4] "Gamebots," http://gamebots.sourceforge.net/, 2009.
[5] "Pyro," http://pyrorobotics.org/, 2009.
[6] "The Mathworks™ ," http://www.Mathworks.com/, 2009.
[7] S. Carpin, et al., "USARSim: a robot simulator for research and education," in Proc. IEEE International Conference on Robotics and Automation, Rome, Italy, 2007, pp. 1400-1405.
[8] "Unreal Tournament® at Epic," http://www.epicgames.com/, 2009.
[9] S. Jameson, et al., "Samarai Nano Air Vehicle – A Revolution in Flight," in Proc. AUVSI's Unmanned Systems North America, Washington, 2007.
[10] "USARSim Manual v.3.1.1" http://sourceforge.net/sourceforge/usarsim/USARsim-manual_3.1.1.pdf 2009.
[11] "Writing Simulink S Functions, Mathworks Simulink User Guide," http://www.Mathworks™ .com 2009.
[12] "Intelligent Software Agents' USARSim page" https://athiri.cimds.ri.cmu.edu/twiki/bin/view/UsarSim/WebHome 2009
[13] "Mathworks External Interfaces Documentation" http://www.mathworks.com/access/helpdesk/help/techdoc/index.html?/access/helpdesk/help/techdoc/rn/f4-998197.html 2009
[14]"getscreen, at MATLAB® Central Exchange, by Matt Daskilewicz" http://www.mathworks.com/MATLAB®central/fileexchange/22031 2009

Neuromorphic System Testing and Training in a Virtual Environment based on USARSim

Christopher S. Campbell, Ankur Chandra, Ben Shaw, Paul P. Maglio, Christopher Kello

***Abstract*—Neuromorphic systems are a particular class of AI efforts directed at creating biologically analogous systems functioning in a natural environment. The development, testing, and training of neuromorphic systems are difficult given the complexity and implementation issues of real physical worlds. Certainly, creating physical environments that allow for incremental development of these systems would be time-consuming and expensive. The best solution was to employ a high-fidelity virtual world that can vary in task complexity and be quickly reconfigured. For this we chose to use USARSim---an open source high-fidelity robot simulator with a high degree of modularity and ease-of-use. We were able to accelerate our testing and demonstration efforts by extending the functionality of USARSim for testing neuromorphic systems. Future directions for extensions and requirements are discussed.**

I. Introduction

Artificial intelligence (AI) has always been a vastly broad and multidisciplinary field including everything from decision making agents in economic models to pattern recognition systems, to learning models, to natural language processing algorithms to name a few (see [1]). While some approaches are biologically inspired, many treat the system like a black-box such that only the observable behavior need show intelligent or human-like qualities. In contrast, neuromorphic systems are a particular class of AI systems aimed not at just biologically inspired models, but biologically and psychologically analogous systems. Neuromorphic systems can model many levels including intra-cellular processes, neuron growth and learning, as well as whole brain systems and connections among them.

Manuscript received July 17, 2009. This work was supported in part by the U.S. Department of Defense Advanced Research Projects Agency (DARPA), Defense Sciences Office (DSO) under "Cognitive Computing via Synaptronics and Supercomputing (C2S2)", DARPA Contract No. 11-09-C-0002. October 2008 – July 2009. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited) by DISTAR Case 14006.

Christopher S. Campbell is a Research Staff Member at the IBM Almaden Research Center, 650 Harry Rd, San Jose, CA 95120 (408-927-1784; e-mail: ccampbel@almaden.ibm.com).

Ankur Chandra is a Research Software Architect at the IBM Almaden Research Center, 650 Harry Rd, San Jose, CA 95120 (408-927-2455; e-mail: achandra@us.ibm.com).

Ben Shaw is an Associate Researcher at the IBM Almaden Research Center, 650 Harry Rd, San Jose, CA 95120 (408-927-2659; e-mail: shawbe@us.ibm.com).

Paul P. Maglio is a Senior Research Manager at the IBM Almaden Research Center, 650 Harry Rd, San Jose, CA 95120 (408-927-2857; e-mail: pmaglio@almaden.ibm.com).

Christopher Kello is an Associate Professor in the School of Social Sciences, Humanities and Arts, at the University of California, Merced, 5200 North Lake Rd., Merced, CA 95343 (e-mail: ckello@ucmerced.edu).

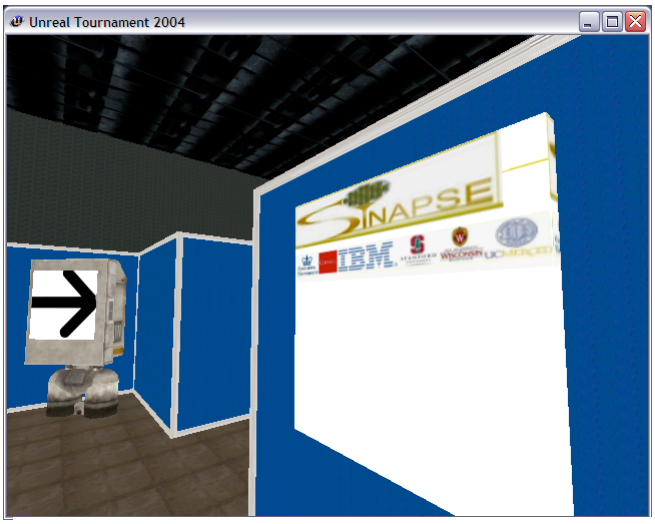


Fig. 1. Synapse 3D Virtual Environment.

However, due to the complexity of many nervous systems (i.e., mammalian brains), neuromorphic systems work usually involves an attempt to model only a part or subsystem of an entire nervous system.

Neuromorphic systems research in the software domain has focused on creating simulations of biological systems at all scales. A strong push in this direction was seen in the explosion of research on neural networks and parallel distributed processing in the 1980s [2]. Much of this work was inspired by theoretical considerations of neural organization going back to the 1950s. Such is the case with the Perceptron [3] (a simple feed-forward network of simulated neurons) and Hebbian learning theory [4]. A digital simulation of a "large" 512 neuron neural network was also conducted in IBM Research at this time [5].

In contrast the software simulation, neuromorphic systems research in the hardware domain involves building actual circuits that represent a brain or brain subsystem. A growing interest group has been in existence from the 1980s and the bulk to research focuses on intelligent robotics, low-level perception and motor control.

Neuromorphic technology that replaces programmable systems with learning or adaptive systems would be a significant step forward. If we subscribe to Turing's idea that intelligent systems merely need to demonstrate intelligent behavior, then programmable systems will be sufficient. However, there is a growing consensus that intelligence involves more than just the demonstration of intelligent behavior through specific task performance.

Rather intelligence requires that good outcomes in task performance are accompanied by additional characteristics such as flexibility, efficiency, generalizability, and creativity. In other words, intelligence is not necessarily just solving a problem, but also the manner in which the problem is solved. Thus neuromorphic systems solve problems in a desirable manner which produces a host of useful qualities:

- managing complex, real-world, dynamic environments
- efficient energy use in terms of power
- efficient information processing
- robustness to damage
- self-organizing and scalable systems
- creative and adaptive use of the environment

The development of neuromorphic systems is without question challenging and complex. Even the simplest mammalian nervous systems have tens of millions of neurons and thousands of interconnected brain structures. To build such a system would require an unprecedented multi-disciplinary team that can work in areas such as computational neuroscience, artificial neural networks, large-scale computation, neuromorphic VLSI, information science, cognitive science, materials science, unconventional nanometer-scale electronics, and CMOS design and fabrication.

II. Building A Neuromorphic System

The Cognitive Computing via Synaptronics and Supercomputing (C2S2) project is a large multi-organization and multi disciplinary effort aimed at creating both the hardware and software components of a neuromorphic system on the scale of a small mammal (i.e., rat). The goal is to create hardware components that behave like biological synapses so the term *synaptronics* is used to refer to the hardware. These collaborating organizations include at IBM Almaden Research Center, Stanford University, Cornell, Columbia University, The University of Wisconsin-Madison, and The University of California-Merced. Each organization may have multiple teams. This effort is in response to a DARPA BAA (i.e., DARPA-BAA 08-28) called Systems of Neuromorphic Adaptive Plastic Scalable Electronics (Synapse) requesting proposals for the development of a neuromorphic system.

There are four main areas if the project and these various teams may work in one or more of these areas. They include:

- *Hardware*: The hardware teams are responsible for building the circuitry for the synaptronic brain from the materials all the way to the full-scale system. They are responsible for creating components that mimic biological synapses showing spike-based information encoding and spike time dependent plasticity (STDP).
- *Architecture:* The architecture teams will evaluate and compile the literature on brain anatomy, physiology, and function. They are responsible for designing the architecture of the synaptronic brain such that it approximates the connectivity, modularity, hierarchical organization, self-organization, reinforcement, and inhibition systems of a biological brain. Processing will should also be distributed, inherently noise-tolerant, and robust to damage.
- *Simulations* : The simulations teams are responsible for creating software to test and explore subsystems to ensure they perform as expected before development of the synaptronic brain. While many of the subsystems can be developed and standard workstations, the large scale simulations will require the use of a supercomputer.
- *Environments*: The environments teams will develop an environment to test, train, and benchmark both the software simulations and the final synaptronic brain. The environments teams will also need to create tests incrementally increase task complexity and intelligent response to evaluate the progress of the project.

The initial coordination plan for C2S2 includes the following. While the hardware team evaluates materials and designs to approximate synapses and brain sub-systems, the simulations and architecture teams start to build a fully simulated prototype of the neuromorphic system. In parallel, the environments team is to create a virtual environment

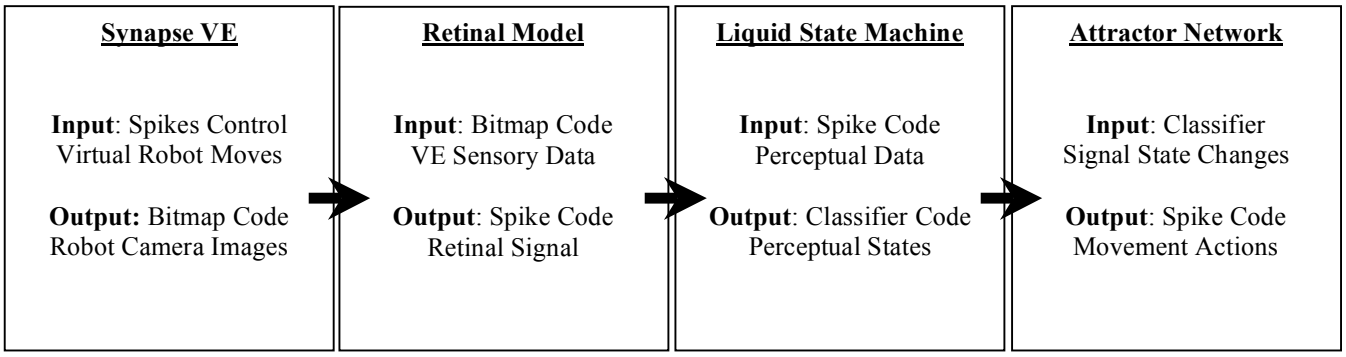


Fig 2. C2S2 team coordination and data flow through several quasi-biological neuromorphic systems. This effort is an attempt to approximate the future final system design with a software prototype. Each box is a different team.

(VE) to bind coordination efforts and integrate simulations.

III. Environment for Benchmarking and Training

A. Motivation

Our goal as part of the Environments and testing team was to create a testing and training environment to support all other teams in the development and benchmarking of their components of the system. Our mission was that the environment had to support

1. fast environment development
2. accelerated training
3. incremental testing
4. benchmarking
5. wide range of environment fidelity
6. wide range of task complexity

In response to these requirements, we created the Synapse Virtual Environments Server (Synapse VE Server) as a common interface layer for all the teams working on the C2S2 project. The Synapse VE Server is integrated with a system called the Unified System for Automation and Robot Simulation (USARSim) [6] which provides a framework for creating, controlling, and interacting with a robot in a VE (see Fig 1). While it is true that each C2S2 team could use USARSim to do their own development and testing, providing one common interface has several benefits:

- removes redundant work,
- standardizes the training and benchmarking,
- provides a unified look-and-feel to the project
- serves as a point of integration for the neuromorphic system components

Using USARSim was logical given that it solves many of the problems we would encounter attempting to interface with a VE. Another benefit is that USARSim provides a way to control and monitor a player in a commercially available game called Unreal Tournament 2004 produced by Epic Games [7]. This game is a standard multi-player networked combat-oriented first-person for both Windows and Linux operating systems. Commercially available VEs tend to be of higher quality---e.g., more realistic, provide development tools, and have a pre-built physics engines. The labor and cost of developing our own VE of similar quality would be far beyond the resources of this project. No doubt, if we were forced to created our own VE, it would not have the complexity and realism necessary to test and train neuromorphic systems. Another benefit is that every improvement in the Unreal Tournament game can be realized quickly to the benefit of the C2S2 project. Any new objects and art created for the game can be used immediately.

B. USARSim

USARSim was created to be a research and education tool

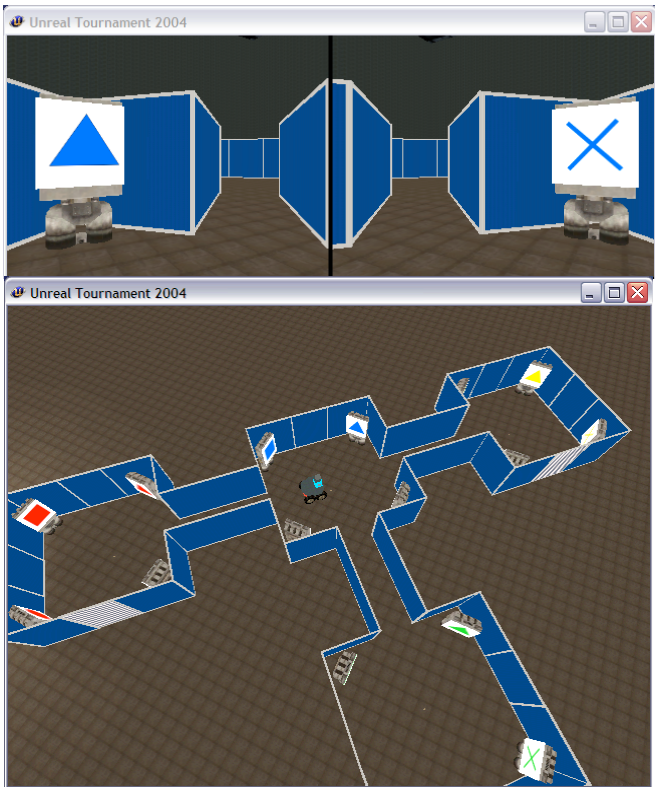


Fig. 3. Top image shows stereoscopic camera view from robot perspective. Bottom image is robot position in a low complexity task.

to provide researchers with an easy-to-use robot controller (e.g., Human Robot Interaction [8]) and automation interface. USARSim also makes it easy for students to learn and explore controlling robots. USARSim is currently heavily used in the Robot World Cup Initiative (i.e., Robocup) community---an international community of researchers and educators working to foster intelligent robots research [9]. This is achieved by providing a standardized problem (the Robocup competition) around which a range of technologies can be developed and benchmarked. The Robocup competition has three main components: 1) Robocup soccer, 2) Robocup rescue, and 3) Robocup Junior. USARSim has been developed with these uses in mind so the tool comes with many prebuilt robots, sensors, and arenas. Much of these materials are robot specific such as sensors for sonar distance and laser range finders, for example. Also included is preprioceptive feedback like robot battery power and wheel speed. These are hardly of interest for developing neuromorphic systems yet, they provide an excellent starting point for, say, more biologically relevant proprioceptive feedback (i.e., muscle tension and vestibular).

USARSim is already being used for benchmarking robots performing in a search and rescue environment. The National Institute of Standards' (NIST) Reference Test Facility for Autonomous Mobile Robots for Urban Search and Rescue was designed as a physical benchmarking environment. USARSim has the robots, environments, and

sensors to recreate this testing facility to a high degree of realism in the virtual world. Realism in an urban search and rescue environment usually includes damaged walls, chairs, and other objects found inside of buildings. It also includes rubble blocked paths, and injured people. The NIST USAR virtual test environment has three levels of increasing difficulty (just like the physical facility) including yellow (i.e., easiest), orange, and red (i.e., most difficult).

Specifically, USARSim uses the Unreal Engine 2.0 through an interface called Gamebots, This interface allows an external application to exchange information, control and monitor, with the engine. While the internals of Unreal Engine 2.0 are proprietary and closed, Epic Games does provide a modding capability for extending "classes" of objects in that run in the engine and the Unreal Virtual Machine. This comes in the form of a Javascript like language called Unrealscript in which objects in the game can be defined and subclassed.

IV. Synapse VE Server

A. Motivation

While USARSim is could be used by each team on the C2S2 project individually, the DARPA requirement for project integration required us to create a special layer---i.e., the Synapse VE Server. The purpose of this layer is to provide a simple ready-to-run virtual testing environment where research teams could select task complexity and training runs. Then they would process the input with their part of the system (neuromorphic subsystem) and output control signals to the Synapse VE server. The Synapse VE provides all of the sensors, effectors, environments, and

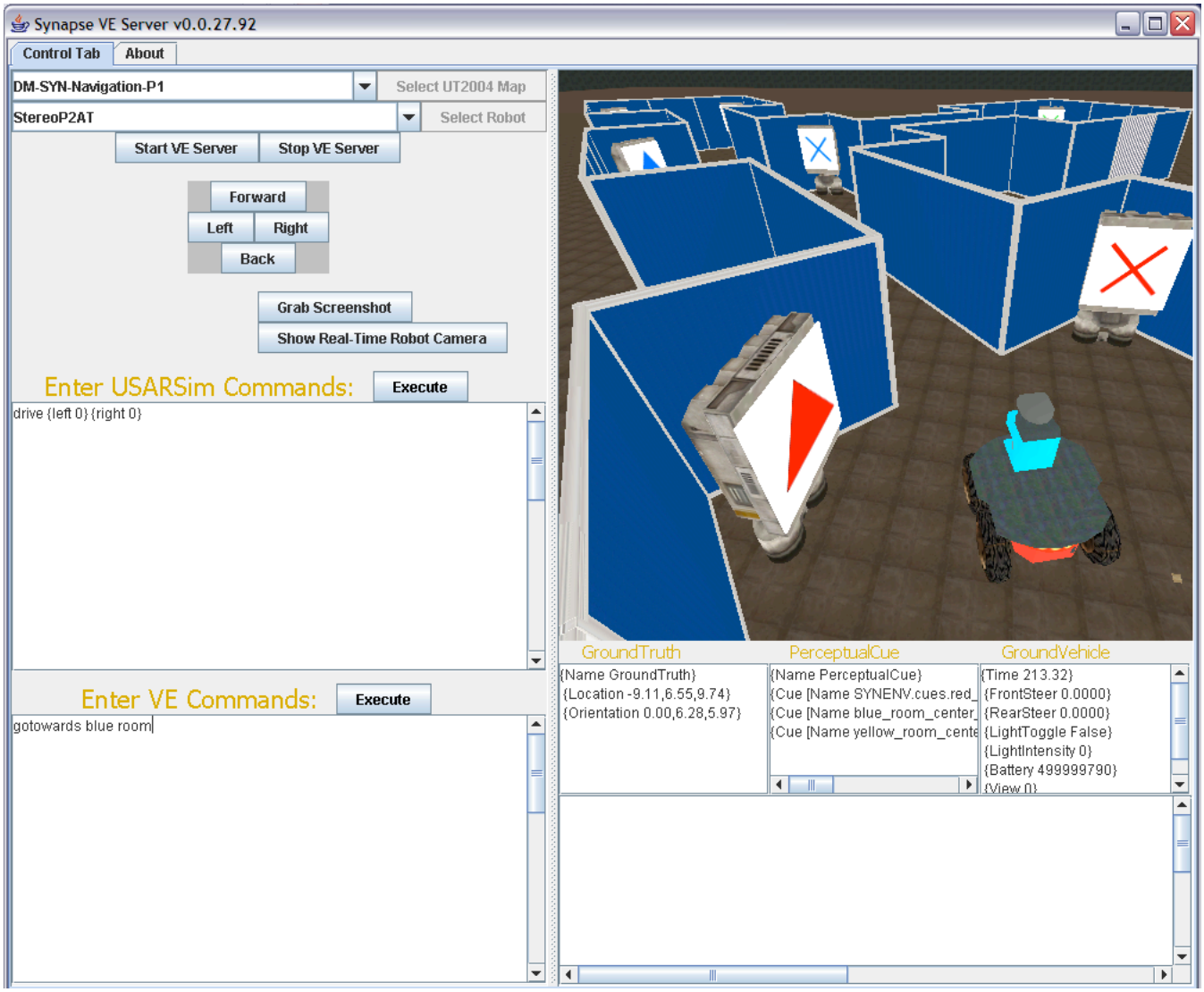


Fig. 4. Screenshot of the Synapse VE Server GUI. In the upper-left corner the complexity of the task can be selected by the map drop-down list. The robot can also be selected. On the left side of the GUI is all the control functions. The robot can be controlled manually in real-time for testing purposes with the forward, left, right, back buttons. Output is shown on the right with real-time video (third-person or robot camera) and sensor data output at 15 updates per second.

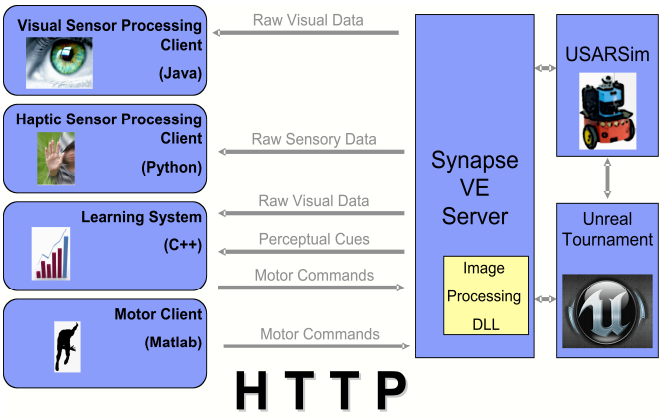


Fig. 3. Synapse VE Server general architecture.

testing experiments needed while gathering all the customized project elements into one place.

Development of the neuromorphic system has necessarily required a layered team approach in which one team develops a base layer (i.e., the VE). Another team takes data from the environment into a retinal model for processing. Another team takes the data from the retinal model into a liquid state machine classifier. Another team takes output from the classifier into a navigation attractor network and so on. As shown in Fig 2, the teams are broken down into general layers to create a software model of the neuromorphic system. Additional layers, details, subsystems, and microcircuitry will be added later.

Another problem with each team creating their own task and training environments is that creating environments is rather complex and time consuming. While Epic Games provides the Unreal Editor (UnrealEd) in order to graphically create environments, the process is really quite complex and error prone. If customized geometry (3D objects) and textures need to be created, it becomes even more difficult requiring a additional set of tools and applications. Each team would probably have to dedicate one person to VE development.

B. *Server Overview*

The Synapse VE Server was designed to provide multiple channels of output from the VE and allow for multiple channels of input (see Fig. 5). This supports ground-truth testing at almost any level of system development. Ideally, the neuromorphic system should be able to transduce raw data into spike-pulse codes and output spike-pulse muscle responses. But, until all the low-level component systems are built this is not a reality. Are we to wait for the entire system to be completed before testing? This seems inefficient. So, the server supports channels of input and output with pre-transformed data. For example, the Synapse VE Server provides a channel of output from the VE that is a raw image showing what the robot is currently viewing. Another simultaneous channel of output is the current visual objects or perceptual cues that the robot is viewing. Any team working on perception and classification can use these channels to test or train their system. Likewise, a team that is working on robot navigation can use the perceptual cues as input and system testing without needing a working perceptual system component to transform the raw image.

C. *Server architecture*

The Synapse VE Server is a stand-alone Java application that was built on a standard client-server model---the neuromorphic system or subsystem is the client. The client connects to the server via HTTP at port 8080 to send commands in plain text (i.e., post) and receives the requested data. This architecture allows clients to be written in any programming language that can interact with HTTP (most modern languages and systems). This approach also allows the work of the client to be distributed to multiple machines, and also allows for geographic separation of client and server if desired.

A "pull" or request model of data exchange was used as the processing requirements/capabilities of the client would not be known ahead of time. Additional benefits of this model include, 1) off-loading of computation to a separate machine that runs the VE and 2) different APIs are not required for each client language. The Synapse VE Server, however, must run on the same machine as USARSim and Unreal Tournatment as it starts both the Unreal server and client with the needed settings. The server communicates with USARSim through the standard message port (i.e., 3000).

A custom C++ DLL was created to provide raw image data to clients and to improve performance. This was achieved using Hook.dll to pull video data directly from the Microsoft DirectX framebuffer for the Unreal Tournament client thereby capturing the robot camera. The Java Native Interface provides a way for the Synapse VE Server to interface with the custom DLL. Performance tests show that 640x480 24bit image arrays can be captured as fast as 25 times per second (fps) or near real-time. Improvements will be required in this performance as the retinal model becomes more sophisticated. Given that there are about 126 million rods and cones in the human retina, the spatial resolution will have to be much higher.

Thus far, only one customized sensor has been created to detect visual (perceptual) cues in the VE. This sensor, called the *perceptual cue sensor*, was written in Unrealscript as a subclass of the USARSim Sensor class. It was added to the StereoP2AT robot in the USARBots.ini configuration file. This sensor runs in the game engine and scans through all the visible (i.e., the robot's field of view) staticmeshes (i.e., 3D objects) and reports back on the ones with the keyword *cue* in the *label* field. Thus, one can make any staticmesh in the environment a visual cue that is reported by the perceptual cue sensor simply by changing the label to have the word *cue* somewhere in the string. The perceptual cue sensor also computes the angle off center, the absolute x,y,z location, and the x.y.z distance of the visual cue. This information is included in the sensor message that is returned. Figure 3 shows the robot field of view on the top

for each of the two cameras and its location in the environment on the bottom.

One of the task environments is shown in Fig. 3 on the bottom. This indoor maze in with four rooms and corridors was created using a set of wall "objects" and other staticmeshes. This environment provides a template to quickly construct a maze with any number of rooms and branches. The Unreal Editor allows whole branches and sections to be selected, copied, and pasted. Also, the environments team can take any image file and put it in the maze as a perceptual object. Thus, almost any type of task or level of complexity can be rapidly created using this template environment.

D. *Interface*

The server has two interfaces. The first is a standard GUI that supports debugging and monitoring for the server (see Fig. 4). The second is a web-based interface that supports remote manual and programmatic control.

The GUI displays a real-time robot camera view at the upper right by acquiring video from the game engine's display window. Movement commands (left, right, forward, back) are used to move the robot. Both the GUI window and the game engine's display window are dynamically updated. The GUI can also be used to send specific commands to the robot and the real-time environment through the two command windows. To send USARSim Commands, the user enters commands in the appropriate window and presses "Execute" (for a list of all commands, see the USARSim manual. To send specific VE Commands, the user enters commands in the window and presses "Execute" (commands include: "face <cue>," which instructs the robot to rotate until it is facing the named perceptual cue; and "gotoward <cue>," which instructs the robot to rotate as above and then proceed toward the named perceptual cue). The GUI also displays sensor information, including "Ground Truth," which displays the robot location (in meters) and orientation (in radians) with respect to the (x,y,z) axes; "Perceptual Cue," which is a list of all labeled cues in the robot's unobstructed field of view with location of each cue (x,y,z) in the environment; and "Ground Vehicle," which contains additional detail on various parameters related to the robot's operation and state.

A web server embedded in the Synapse VE Server allows commands to be sent to the robot in the VE via HTTP through a browser window or programmatically. To use this interface, the user launches a web browser and types the following URL: `http://{host_name}:8080/.` The response "welcome to the Synapse VE Server" indicates the server is working correctly. Commands are passed to the robot using the following request syntax: `http://{host_name}:8080/command` where "/command" can be any one of the following: `/robot` to obtain details about the robot, such as location and orientation. Location is given in meters and orientation is in radians on the (x,y,z) axes; `/cues` to obtain a formatted list of all the labeled cues in the robot's unobstructed field of view; `/image.png or /image.jpg` to fetch either a PNG or JPEG image from the current game engine display window; `/forward, /backward, /left, /right or /halt` to move the robot forward, backward, left or right—note that motion continues until a halt command is issued.

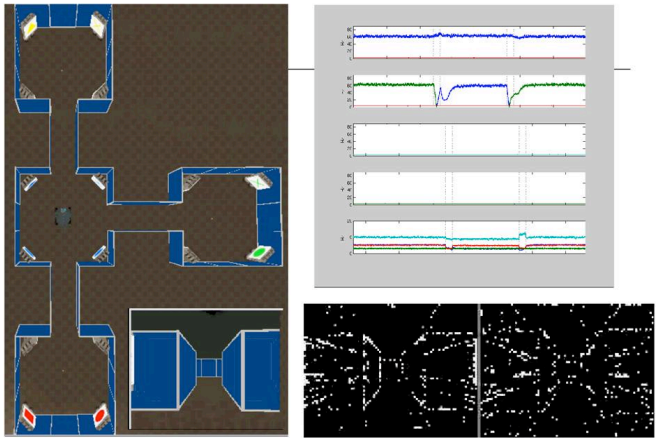


Fig 6. Testing of retinal model and spiking navigation systems in the VE with a low complexity four room/state task. The response of spiking neurons for navigation is shown right-top and the retinal spike response is shown right-bottom.

V. VE Task Complexity

Sensory-motor interaction with an evolving, changing environment is a key to intelligent behavior, as all intelligent must be both situated and embodied. The behavioral tasks for the neuromorphic system fall into three broad cognitive categories, highlighting problems of perception, planning, and navigation. Though the original proposal outlined three separate environments (one for each category of task), we revised the plan such that a single 3D virtual world was used to develop tasks that highlight each of the three kinds of problems. This change made for a uniform and elegant conceptual design of the VE. All tasks are conceived as traversals on state-space graph, with perception as state identification based solely on the current state, navigation as action selection based on the current state and previous states, and planning as action selection based on prediction of the consequences of future possible actions. In the state-space framework, all tasks are versatile, extensible, indefinitely scalable in complexity, and are amenable to objective, quantitative, and comparative performance evaluation. The tasks can be extended to provide interaction over a wide range of space and time scales, and can offer comparison to behavioral studies.

Figure 6 shows a relatively simple task environment with only four states (decision points) shown by the four interconnected rooms. The retinal spiking model takes the raw image data and transforms it into edges and perceptual objects (right-bottom part of Fig. 6). The attractor spiking network takes the perceptual objects as input (spiking input)

to detect a) current state and b) next desired state. The attractor network response is shown in the right-top part of Fig. 6.

While the current maze-like task environment appears somewhat artificial, it may be desirable in future work to allow the actual VE to take on different visual characteristics of other, more realistic-naturalistic environments. We have formulated an approach whereby the graph traversal formalization (and its attendant benefits of comparability and quantification) can be maintained and applied to more naturalistically-rendered environments. Such is diagrammatically illustrated in Fig. 7. In this an environment, task complexity is conceived in terms of three dimensions: the number of different perceptual states processed by the agent, the degree of memory (history) and/or prediction (anticipation), and the level of symbolic abstraction involved in the perception-action relation. Actual traversals would involve tasks of systematically-varying perceptual difficulty and complexity, with local and global orientation cues made available or obscured in a controlled (and perhaps dynamic) manner.

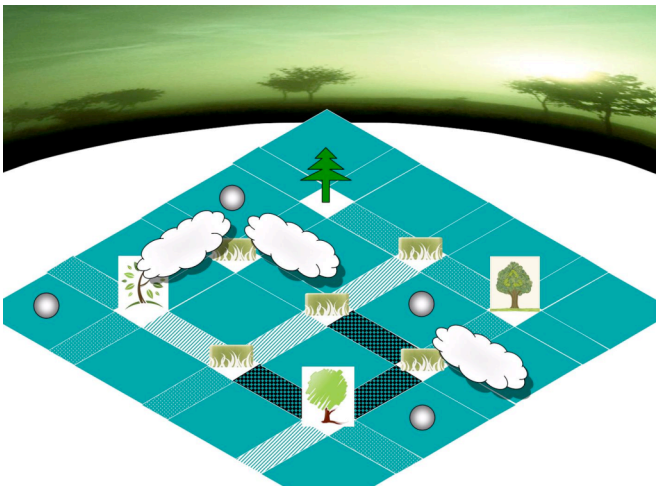


Fig 7. Illustration of a possible "gameboard" structure to underlie future VE renderings of more naturalistic environments while preserving the formalization of state-space graph transitions.

VI. Conclusion & Future Work

Our goal as part of the Environments and testing team was to create a benchmarking and training environment to support all other teams in the development of their components of the neuromorphic system. Our mission was that the environment had to support

1. fast environment development
2. accelerated training
3. incremental testing
4. benchmarking
5. wide range of environment fidelity
6. wide range of task complexity

We have made significant strides in achieving fast VE development, incremental testing, and high VE fidelity given the use of the USARSim framework. Because parts of the task environment can be copied and pasted, components are be reused and new task configurations can be quickly created. Environment development is clearly faster than using a physical test facility. VE fidelity too can be easily decreased or increased depending on what would provide a challenge for any part of the neuromorphic system. For example, the lighting can be manipulated so that there is no directional lighting or shadows (low fidelity) or there is only directional lighting with many different types of lightsources (high fidelity). Having this capability means that incremental testing is possible.

Still, further technical work is needed to increase fidelity in certain areas. One such area includes increasing the number of input channels to more than just vision and some types of proprioceptive feedback. This would mean adding sensors for hearing, touch, olfactory, and taste channels. Also, these modalities may have to be added to the Unreal Tournament environment as it was not designed to provide olfaction simulation, for example. Another area in which fidelity could be increased includes adding reward and energy (e.g., food, heat) feedback channels. Such objects would need to be added to the VE along with internal functions to the robot or game actor.

This would also mean that the data rate of information for each of these channels was high enough to simulate real-world stimuli. As stated earlier, the current image data rate is only a 640x480 24bit pixel array 25 times per second. That is merely 23 Mbps---a bitrate that is only a small proportion of the data entering the eye. To improve performance the system architecture may need to be redesigned with a faster UDP-based protocol such as RTSP instead of HTTP.

Accelerated training is the least known among all the goals. Technical research needs to be conducted to see 1) if the Unreal Engine can be accelerated and by how much, 2) how USARSim sensors and other components can be accelerated in a unified and consistent manner, 3) how this acceleration interacts with performance characteristics of the server hardware, 4) how much can the input and output data rates be accelerated?

We have defined the environment in terms of transversals in a state-space graph. This graph then could lead to a method for measuring task complexity. However more theoretical work is needed before a quantitative measure of task complexity can be achieved. Also, providing a quantitative measure for task complexity would be central to efforts at benchmarking neuromorphic system performance.

Finally, future work will be aimed at creating a player in the VE that resembles a mammal rather than a robot. The mammal should have articulated joints with fairly realistic skeletal muscle control. This will allow for the development of neuromorphic motor cortex and cerebellum subsystems.

ACKNOWLEDGMENT

We would like to thank Stefano Carpin and the USARSim team for development support and suggestions. We also thank Brian Wandell, Bob Dougherty, and the Stanford team for work on the retinal model. Finally, we acknowledge Stefano Fusi and Daniel Ben Dayan Rubin for work on the attractor network model.

REFERENCES

[1] Simon, H. A. 1961. Modeling human mental processes. In *Papers Presented At the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference* (Los Angeles, California, May 09 - 11, 1961). IRE-AIEE-ACM '61 (Western). ACM, New York, NY, 111-119.

[2] Rumelhart, D.E., J.L. McClelland and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*, Cambridge, MA: MIT Press.

[3] Rosenblatt, F., The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, *Psychological Review*, 65: 386-408" (November, 1958).

[4] Hebb, D. *Organization of Behavior*, Wiley, 1949 .

[5] Rochester, N., J. H. Holland, L. H. Haibt and W. L. Duda, Tests on a Cell Assembly Theory of the Action of the Brain Using a Large Digital Computer, *IRS Trans. on Information Theory*, IT-2; #3, (September, 1956) .

[6] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, C. Scrapper, USARSim: a robot simulator for research and education. *Proceedings of the 2007 IEEE Conference on Robotics and Automation*, pp. 1400-1405. 2007.

[7] Epic games, http://www.unrealtechnology.com/

[8] M. Lewis, J. Wang, and S. Hughes (2007). USARSim : Simulation for the Study of Human-Robot Interaction, *Journal of Cognitive Engineering and Decision Making*, (1)1, 98-120.

[9] Robot World Cup Initiative, http://www.robocup.org

Acoustic Sensing in UT3 Based USARSim

Steven M. Nunnally, Stephen Balakirsky, *Senior Member, IEEE*

Abstract— **This paper presents the implementation of an acoustic sensor for Unreal Tournament 3 based USARSim. The sensor is able to provide the bearing, volume, and delay to specific sound sources and is also able to eavesdrop on general game sound. General information on the addition of a sensor to UT3 based USARSim as well as specific aspects of the acoustic sensor and sound server are presented.**

I. Introduction

THE Unified System for Automation and Robot Simulation (USARSim) is a high-fidelity simulation of robots and environments based on the Unreal Tournament game engine[1]. The USARSim project was originally created in 2002 by Carnegie Mellon University (CMU) and the University of Pittsburgh to study human-robot interaction in the area of Urban Search and Rescue (USAR) (4, 5). One main feature of USARSim that sets it apart from other simulation systems is the extensive effort that has been performed in validating the robot and sensor models used in the simulation (2). This validation has allowed numerous robotic controllers to be developed in simulation and directly ported to real robotic hardware (1). In addition to providing valid models of existing sensors, the simulation environment allows developers to imagine new sensors that have yet to be created. The utility of these sensors may then be approximated through the simulation without incurring the expense of developing the actual hardware.

USARSim concentrates on providing a consistent application programmer's interface (API) and validated models that are supported by a world-class graphics and physics engine. Therefore, as time progresses it becomes necessary to change the underlying simulation engine. The USARSim developer's community is currently porting the body of software to the newer UT3 game engine from Epic Games. This paper outlines the general development scheme that must be followed to add a new sensor to the UT3 version of USARSim. It then follows the specific implementation of a novel acoustic sensor framework that has been made available to the general community for test and comment. This framework consists of an acoustic sensor, a sound generator, and a client/server application for the receipt of observed acoustic signals.

To date, the simulation has not had a realistic acoustic sensor. Therefore, the entire concept of working with sound in USARSim was a new task for the project. This required an understanding of how sound worked in UT3 and how sound could be manipulated. The objective was to give the operator the ability to hear sound emulating from the environment, and to also pick out specific classes of sounds that would receive additional processing to determine attributes related to the sound source.

To complete this objective, the UT3 system, how it deals with sounds, and techniques for modifying UT3 must be understood. The UT3 system is designed to be modified with an API into the game engine being provided by Epic Games. Game modifications are performed through the use of Unreal Script code which is the basis for developing in UT3. Unreal Script is much like java, c++, or any other object oriented programming language. The main problem with coding for the game API in Unreal Script is the lack of documentation. However, finding the correct function in Unreal Script allows for full use of the system's design.

Trying to figure out how the engine deals with sound is an important part of creating an acoustic sensor. While Unreal Script is fairly accessible for developers, there is a lot of hidden code to allow Epic Games to protect its product. This means that many of the calculations that UT3 performs are in hidden code and cannot be accessed by developers. This was the case for the sound calculations. Therefore, when hearing sounds, the new sensor had to perform similar calculations as the game engine to create similar effects.

The remainder of this paper is organized as follows. Section two explains the steps of adding a sensor to USARSim. Section three explains the different techniques used to add the Acoustic Sensor while Section four compares these techniques. Section five addresses problems with the current implementation. Finally, Section six draws conclusions from the work.

II. USARSim Sensors

A. Implementing Sensors in USARSim

Without sensors, robots cannot obtain any information about their surroundings or self, and will have difficulty in operating to meet goals and satisfy objectives. Thus, integrating sensors into a robotic simulation is a critical step. Once one adds data from sensors, the construction of a world model as well as reactive behaviors are possible. Operators and developers gain the ability to obtain information from

Manuscript received August 20, 2009. This work was supported by a grant from the department of Homeland Security HS-STEM Internship Program.

S. Balakirsky is with the National Institute of Standards and Technology (NIST), Gaithersburg, MD 20899 USA Phone: (301) 975-4791 Fax: (301) 990-9688 e-mail: stephen@nist.gov)

S. M. Nunnally is with the National Institute of Standards and Technology, Gaithersburg, MD 20899 USA and Roanoke College, Salem, VA 24153 USA (e-mail: smnunnally@roanoke.edu).

[1] Certain commercial software and tools are identified in this paper in order to explain our research. Such identification does not imply recommendation or endorsement by the authors, nor does it imply that the software tools identified are necessarily the best available for the purpose.

the robot's surroundings and can work to meet the objective of developing automated algorithms.

To add a sensor to USARSim, a developer has to first understand what information is trying to be obtained from the environment or the robot. Then, one may look through Unreal Script API's to attempt to understand how the gaming engine stores that information. Online information is available on the API from sites such as (3, 6). Careful examination of the API will lead the developer to the classes they will want to use to obtain the necessary information when he/she begins developing the code for the sensor.

When adding a sensor to USARSim, two classes must be added to the code. One class is directly below the Sensor class. This class, which we will refer to as sensorChild, performs all of the calculations necessary to obtain the information, and output it. The SensorChild must follow some guidelines to work with the rest of the USARSim code. The class must have a *PostBetginPlay* function, a periodic "timer" function, as well as a *GetData* function. The *PostBeginPlay* function initializes the sensor and sets the interval that this sensor will use when gathering its data. This is accomplished by installing a periodically triggered timer function. Each call of the timer is based on the interval set in this function. The timer function sets all of the data that this sensor finds and stores it in a global string variable that the class owns. The timer function keeps adding the data to the string until the USARSim network accesses the *GetData* function in this sensorChild. When called, the *GetData* function adds necessary header information to the global string and returns it for transmission over the USARSim socket[2]. Other functions for returning sensor configuration information and setting operating parameters are also usually provided.

After following these steps, the developer has created a new sensor for the simulation, but there is still work that needs to be done. A visual representation of the sensor must be modeled and added to the simulation so that the sensor shows up when it is attached to the robot. This class, which we will refer to as sensorGrandchild, extends below sensorChild and is the class that will set the skeletal mesh to use as the model for the new sensor. SensorGrandchild only has default properties that modify the operation of the sensorChild class. Next, the developer must test the sensor to validate it. The simulation loses its realism and cannot meet its objectives if the sensor does not return realistic results. This validation requires the developer to check not only the accuracy of the information returned, but to also validate that the sensor returns the correct amount of noise. Sensors all have some inaccuracies and it is the developer's responsibility to add these inaccuracies to ensure realistic results from the sensor when other developers try to use these results to create their own automated algorithms.

[2] USARSim provides for all commands into the system and data out of the system to be transmitted over a TCP/IP ASCII based socket.

III. Acoustic Sensor

A. Acoustic Sensor Specs

For our acoustic sensor, it was desired that the sensor could detect a specific sound with a volume of 140 dBA within 8000 m of the sensor. This figure is based on providing realistic results to aid robots and operators in finding victims in an USAR disaster scenario. To meet this objective, listening for human noises is very important. In addition, it was desired that the sensor provide an "eavesdrop" mode that would allow direct playback of any sounds that are detectable by the robot. The sensor is made up of an array of microphones which will only report sounds produced from another newly created sound generator class. Sounds resulting from hazards, (e.g. gas leaks), and sounds from distressed humans (e.g. cries, or heartbeats) have been instantiated in USARSim. This array of microphones gives the volume of the sound, the amount of time that speakers recreating the sound must delay the sound (in seconds) before playing it to account for distance, and the direction, as a unit vector, that the sound came from. This is a prototype of what future acoustic sensors could do, since no sensor is known to the authors that filters data based on searching only for hazards or distressed humans. The functions of the acoustic array can be tuned so that this class of acoustic sensor can be reproduced when it becomes available.

Figure 1 depicts the graphical representation of the sensor that was created for the simulation. This representation contains three coplanar microphones, and one additional microphone located above the plane. This combination of microphones allows for the detection of the direction of a sound in three dimensions. The four microphones are modeled after the ICP Array Microphone, Model 130A40 from PCB Piezotronics Vibration Division.

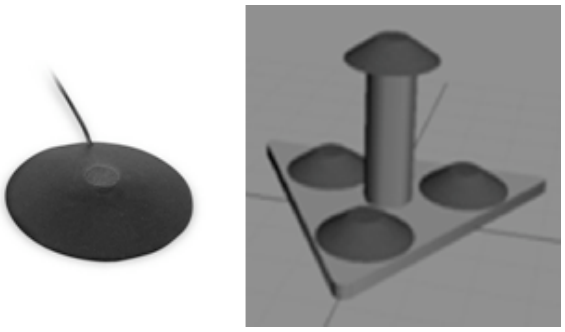


Figure 1: The left side shows the real ICP Array Microphone, Model 130A40 from PCB Piezotronics Vibration Division. The right side shows the virtual example of an acoustic array sensor.

B. Acoustic Array Implementation Method 1

The first attempt, referred to as method 1, at completing this task was to derive acoustic information from the Controller class. The Controller class is a child of the Actor class, as shown in Figure 2. This class can be used to get feedback from an individual Pawn's senses (e.g. touch and hearing). Pawns are the class below Actor which represents

all vehicles or characters in UT3. Actors produce sounds by making a call to a *MakeNoise* function. The Controller class has a *HearNoise* event that is supposed to be triggered anytime the *MakeNoise* function in the Actor class is called to produce a sound. Data included with this event includes the actor object that made the sound and the loudness of the sound (as a float). The loudness value is based on a parameter that is specified in the *MakeNoise* function's call.

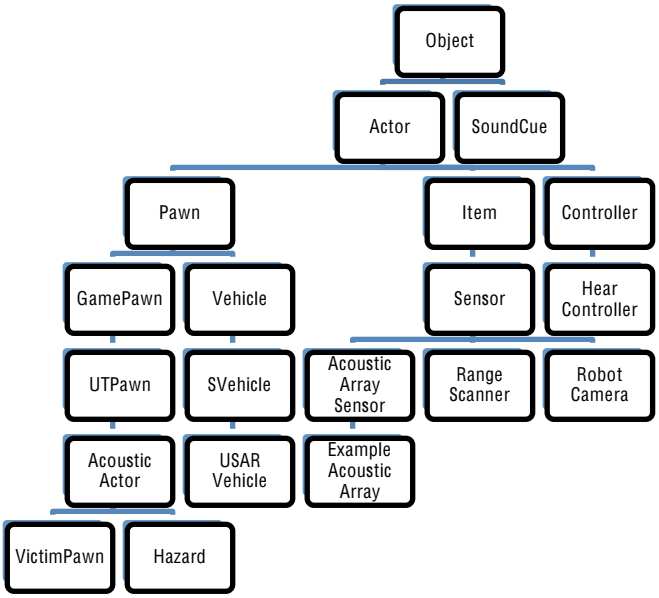


Figure 2: The Hierarchy of classes used for the acoustic sensor.

To access this data, a class called *HearController* was created that extended the Controller class. This class had a global array to hold the actors which made noises and the noises' loudness. The *HearNoise* event was overridden to simply store the data that triggered the event into this global array. Another function was created to remove the data from this array and to reset the variables after the data was extracted.

The *AcousticArraySensor* class, which as shown in Figure 2 extends from USARSim's Sensor class, contains the working code for Acoustic Sensors. This class must spawn a *HearController* object and set necessary variables. For the Controller events to work properly, some variables inherited from the Pawn class must be set. To set this variable, some knowledge of the inner workings of USARSim, UT3, and polymorphism are necessary. As shown in Figure 2, the USARVehicle class may be traced back to a Pawn, so the variables are accessible from the USARVehicle class. In addition, sensors may be traced back to Items that are attached to robots, and the robot object which this sensor is attached to may be accessed through inheritance from the Item class. Through using this object, the necessary variables from the Pawn class can bet set.

During operation, each call to the timer function causes the sensor to access the data that is stored in its *HearController* object, reset the object's arrays, and perform the necessary calculations with the accessed data.

C. *AccousticArraySensor Implementation Method 2*

As described in the previous section and depicted in Figure 2, the *AcousticArraySensor* extends from the basic sensor class. This corresponds to the concept of a sensorChild. The actual model of this sensor is provided in the *ExampleAcousticArray*, which corresponds to the concept of a sensorGrandChild.

The *AcousticArraySensor* class calls its *GetSoundData* function every timer cycle. This function loops through all of the detected sounds produced by a special *SoundActor* class and calculates all of the specified information from the previous section (e.g. volume, delay, and direction). This information is appended to its global string (the soundStr variable). This string is packaged and transmitted over the USARSim socket by the *GetData* function, after which it is reset and cleared. This cycle continues as long as the sensor exists.

D. *Sounds in USARSim for Method 2*

A sound that is detectable by the *AcousticArraySensor* is made when a class that extends from *AcousticActor* creates a sound. Sound creation mimics the standard UT3 sound mechanism. The class hierarchy for *AcousticActor* may be found in Figure 2. USARSim currently instantiates *Victims* and *Hazards* as AcousticActors. These *AcousticActors* have *SoundCue* objects that are generally set in their properties when placed in the world. These sound cues are the special sounds that the sensors are designed to react to (i.e. provide the volume, delay, and direction of).

A sound is propagated from an *AcousticActor* to the *AcousticArraySensor* through the use of the *AcousticActor*'s *PlayAcousticSound* function. This function cycles through all instantiations of the *AcousticArraySensor* class and calls the *AcousticArraySensor*'s *HearSound* function which computes if the sound is able to be heard. The *AcousticActor* is responsible for setting an initial volume that is scaled correctly by the *AcousticArraySensor*. Scaling is based on a value of 1.0 representing the volume of a gunshot at close proximity, which is approximately 140 dBA. If necessary, the *HearSound* function in the *AcousticArraySensor* adds the information from the sound to its sound array. This array is made up of a structure called *SoundInfo*, which stores the sound's volume, location, duration, and timestamp all relative to the time and location that the sound was made.

E. *AcousticArraySensor's Calculations*

Both methods described above have to make calculations to extract the data from the sounds that the sensor "heard". Recall that all detected sounds are placed in a global sound array. If this sound array contains data, then the timer will cycle through the array calling *GetSoundData* for each item. The functionality of *GetSoundData* is illustrated in Figure 3. This function computes data for each sound and adds the data to soundStr for future transmission over the USARSim socket.

```
function string GetSoundData{
    string result;
    // Get distance from sound source to sensor
    // Calculate the time to delay playing the sound
    // Calculate the attenuation of the sound's volume
    //   from the source to the sensor
    // Calculate the direction from the sensor to the
    //   source as a unit vector
    result = // Calculated values and headers for the data
    return result;
}
```

Figure 3: This is pseudo code for the *GetSoundData* function where all of the calculations are completed.

In *GetSoundData* the sensor calculates the initial distance, based on the location of the sensor and the location of the sound. In order to apply an appropriate delay to the playing of a sound, the SOUND_SPEED constant and the computed distance are used to formulate the correct delay that occurs between sound generation and the sensor receiving the sound. Without this computation, the sensor will know of the sound as soon as it is played, which is unrealistic. This equation also takes the time that the sound was played into consideration, so if the sound should have already been played, then the time is automatically set to 0 thus causing the sound to be played immediately.

The function must also calculate the volume of the sound relative to the sensor. The code first checks to see if the sound needs to be muffled. This is done by checking to see if there is a world obstruction between the sensor and the sound location. If necessary, it "muffles" the sound by doubling the distance the sound must travel. While not perfectly accurate, this technique removes the requirement of computing multi-path and actual material sound attenuation. The sound drop-off is then calculated so that the loudness relative to the sensor is determined. This is shown by Equation 1 below.

$$[1]\ RelLoud = l_i - ((\frac{DROP\_OFF\_RATE}{GUN\_LOUDNESS}) * log_2(\frac{d}{o_u}))$$

Equation 1: This equation determines the volume of the sound at the sensor's location.

In Equation 1, l_i is the initial loudness (1.0 for guns), GUN_LOUDNESS is a constant currently set to 140 dBA A-weighted decibels, and the DROP_OFF_RATE is an environmentally dependent constant that is currently set to 4.5 dBA every time you double the distance. d is the distance from the initial sound in Unreal Units (UU), and $o_u = o_r * UU\_TO\_M\_CONVERT$, where o_r is the radius of no drop-off (this value needs to be validated, but until a test is developed 1 m is used). The simulation uses a conversion constant UU_TO_M_CONVERT, which is currently 250 UU = 1 meter, so o_u is the radius of no drop-off converted into UU.

The sensor also calculates the direction to the generated sound by computing a vector from the sensor to the sound's location. Using this vector and the direction that the sensor is currently facing, the function is able to calculate the sound's direction relative to the sensor's heading by using a rotational matrix. This information is formulated as a unit vector, giving the direction in a unit vector whose origin is centered at the sensor's location and oriented so that x+ is straight ahead, y+ is to the right, and z+ is straight up relative to the sensor. This is shown in Figure 4, all relative to the sensor's heading. The function also stores the sound's duration.

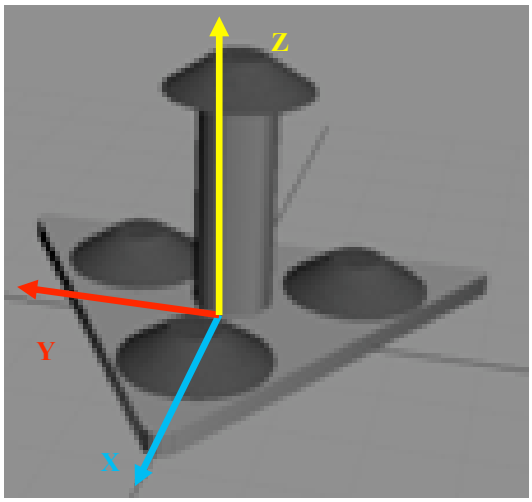


Figure 4: Representation of the axes of the sensor's unit vector towards the sound source. The front of the sensor is in the positive x direction. Note that it is the responsibility of the static mesh designer to set the "front" of the mesh when designing the graphical representation of the sensor.

After storing this data in the global soundStr, the soundInfo is removed from the array and the loop is restarted until all sounds since the last call to timer have been calculated. All data is added to the string which is returned and cleared when called on by the *GetData* function for use in other programs.

ExampleAcousticArray is the actual sensor. The model package is included as AcousticArraySensor.upk in USARSimSkeletalMeshedVehicles that is available from the SVN repository of the USARSim site.

F. USARSim Sound Server

The next task was to decide how to realistically allow the operator to actually hear the sounds in the robot's environment. The game engine automatically handles playing the sounds correctly over the system's speakers from a player's position. This extends to USARSim's robots when the operator is viewing what the robot sees. However, many more issues arise when considering the needs of the users.

Many users run the simulation system on one computer and operate the robots from another. This is facilitated through the socket interface that is provided by USARSim which allows for connections from any application that can

open a TCP/IP socket. Currently, the simulation supports this relationship for sending/receiving commands/data to/from the server and for allowing camera images to be sent out over a separate socket. However, since sound was not a part of the simulation, there was no support for sound transmission over a socket interface. This need required the creation of a new server that runs concurrently with the simulation, which we call the SoundServer. This server provides the audio output stream from the computer and sends it to multiple connected clients known as SoundClient programs. This data exchange between the client and server needed to be completed with as little lag as possible. However, in actual systems there is lag between sound receipt by the robot and receipt by the control station. This lag may be modeled in a future release in order to increase the realism of the sound simulation.

Once the client and server were completed, sound could be played across a network. This provided a system independent technique for users to hear the sound from the simulation. However, one more issue with regard to realism now remained. Through the use of the sound server/client, users could hear sounds even if they did not have an acoustic sensor attached to the robot.

This was addressed by creating a way to mute the system's sound. In the *USARVehicle* class, a boolean variable was added to determine whether or not the robot had a sensor of type Acoustic. If the robot has an acoustic sensor, the volume is normalized. If the robot does not have an acoustic sensor, then the volume is muted within UT3 so that no sounds will be played. This works well when only a single robot is being utilized in the simulation. However, if multiple robots are desired (which of course they are!), then this check will have to take place when changing views from one robot to another.

IV. Experimental Results

Method 1 appeared to work with early tests since the sensor was picking up data from a UT3 character in the room that was creating sounds by shooting and jumping. Unfortunately as testing proceeded, the *MakeNoise* function was not the only factor that determined when the *HearNoise* event was triggered. There are more requirements for triggering this event, but these requirements are hidden and it was determined that this method will not work for the sensor.

This problem created the need for method 2, which gives developers more control over what sounds are heard and what data is passed to the sensor. This is backwards from real life since the sound determines whether or not it is heard by the sensor and what data is passed, but this is the only way USARSim developers can be sure that the sound will be heard by the sensor.

V. Future Work

This newly acquired ability works well for a single robot, but further progress is still necessary to obtain the full potential of this sensor. Firstly, the sensor must be validated. Currently all of the data that is output is 100 percent accurate, which makes this an unrealistic sensor. Noise must be added to the sensor to try to gain the realism expected by the users of USARSim.

The calculation on sound attenuation, as well as sound delay, and the way in which the sensor muffles obstructed sounds must also be validated. These are important calculations which can allow for more realism with the sensor, so determining ways to test these equations will be an important part of validating this sensor. It is not expected that the current techniques will produce values that are consistent with actual sound transmission. However, knowing that shortfalls exist, and what form these shortfalls take, is important for determining limitations on the use of the new sensor.

The efficiency of the code behind the sensor must also come in question. Each time a sound is made, the *AcousticActor* class searches through all actors looking for *AcousticArraySensor* objects. This is not very efficient, but works with the current set up. Understanding method 1 and the efficiencies embedded into it could help fix these inefficiency problems. Another option would be to understand how events work in Unreal Script and creating an event that does something similar to the *HearNoise* event used in method 1, but would give the developers more control.

Finally, the sensor must be extended to be compatible with the use of multiple robots. Sound switching for eavesdropping must be developed as well as the ability to listen to multiple robots simultaneously.

VI. Conclusion

This paper presented a novel acoustic sensor, sound generator, and sound client/server application for use with USARSim. In addition, the general methodology for adding a new sensor to UT3 based USARSim was examined.

The ability to detect sounds will greatly increase the realism of USARSim with respect to USAR scenarios. Hearing sounds in a USAR scenario is important for operators to find victims or hazards. It can save first responders from searching an area that is leaking hazardous gases and can help locate victims that are out of sight. Adding this ability gets this simulation one step closer to accomplishing the goal of becoming a training tool for USAR squads. It might also spur a creation of algorithms that use sound to coordinate multiple robots, or the creation of new sound detecting hardware

Acknowledgment

The authors would like to thank Joe Falco for his assistance with the USARSim class structure, Nathanial Weinmann for his assistance with 3D modeling, and Fred Proctor for his assistance with Windows Batch commands.

REFERENCES

[1] Balakirsky, S., Proctor, F., Scrapper, C., Kramer, T., "An Integrated Control and Simulation Environment for Mobile Robot Software Development", *Proceedings of the ASME Computers and Information in Engineering Conference*, 2008.

[2] Balaguer, B., Balakirsky, S., Carpin, S., Lewis, M., Scrapper, C., "USARSim: A Validated Simulator for Research in Robotics and Automation", *Proceedings of the Robot Simulators: Available Software, Scientific Applications, and Future Trends Workshop at IEEE/RSJ*, 2008.

[3] BeyondUnreal Forums – Powered by vBulletin. 22 Aug. 2009 <http://forums.beyondunreal.com/>.

[4] Lewis, M., Sycara, K., and Nourbakhsh, I., "Developing a Testbed for Studying Human-Robot Interaction in Urban Search and Rescue," *Proceedings of the 10th International Conference on Human Computer Interaction (HCII'03),* 2003.

[5] Wang, J., Lewis, M., and Gennari, J., "A Game Engine Based Simulation of the NIST Urban Search & Rescue Arenas*," Proceedings of the 2003 Winter Simulation Conference*, 2003.

[6] "UnCodeX." Index. 22 Aug. 2009 <http://www.cdfunk.com/highnoon/ut3/>.

USARSim – Porting to Unreal Tournament 3

Stephen Balakirsky, *Senior Member, IEEE*, Joseph Falco, Frederick Proctor, *Member, IEEE*, and Prasanna Velagapudi

***Abstract*—. This paper presents current efforts to migrate USARSim, an open source high fidelity robot simulator, from the Epic Games UT2004[1] gamming platform to UT3. This software "port" is a non-trivial effort which includes software recoding to support unreal script syntax changes, use of a better physics engine and new graphics representations. In addition, this porting effort has marked an ideal opportunity to improve the USARSim implementation through code clean-up/consolidation, architecture redesign, improvements to client/server distributed processing, and a more detailed simulation environment characterization. This major effort provides many benefits to the USARSim user community including detailed graphics, improved physics effects, a more organized software structure, faster processing of distributed objects, support for manufacturing environments and improved validation of the simulation environment.**

I. Introduction

THE National Institute of Standards and Technology is leading an effort to upgrade the USARSim (Unified System for Automation and Robotic Simulation) [1] software implementation. This upgrade will ultimately provide more detailed graphics, improved physics effects, faster processing of distributed objects, and improved validation of the simulation environment.

USARSim is a high-fidelity simulation of robots and environments based on the Unreal Tournament game engine. MOAST (Mobility Open Architecture Simulation and Tools) [2] is a control framework that aids in the development of autonomous robots. It includes an architecture, control modules, interface specs, and data sets. When used together, USARSim and MOAST provide a comprehensive set of open source tools for the development and evaluation of robotic systems where initial control development is validated in simulation with eventual transition to real hardware.

Epic Games' Unreal Tournament (UT) is a widely used and affordable state of the art commercial game engine. USARSim was originally developed using UT2003/UT2004 and Unreal Engine 2 (UE2). This paper describes an ongoing effort to port the USARSim UT2004/UE2 implementation to Unreal Tournament 3 (UT3). UT3 is Epic games' latest UT release that uses their new Unreal Engine 3 (UE3).

UT3/UE3 provides significant improvements through its change from using Karma Physics to the Nvidia PhysX engine, support for more sophisticated models, and enhancements to lighting, animation, and interactions with the simulation environment. However, these significant changes require major design changes to the USARSim implementation. In addition, this major overhaul of USARSim provides an opportunity to improve the overall USARSim implementation through code clean-up/consolidation, architecture redesign, improvements to client/server distributed processing, and a more detailed simulation environment characterization.

II. Overview of the Porting Effort

The Nvidia PhysX physics engine of UE3 will provide more realistic physics as compared to the Karma physics engine of UE2 and provides the opportunity to install a hardware accelerator to offload physics calculations from the CPU; ultimately producing a more realistic simulation. However, deprecation of Karma physics in UE3 has eliminated the UT2004 KVehicle class. This KVehicle class is the parent of the USARSim KRobot class from which all USARSim robots are derived. The new base class for vehicle implementation is the SVehicle class which provides many simplifications over the KVehicle class. However, this change of classes requires a redesigned KRobot, now called USARVehicle.

Another major effort of this port is remodeling. This includes changes in how both environment and vehicle component models are created. For environments, the editor for world creation in UT2004 used subtractive geometry. The environments start as a solid lump of mater and the designer carves out the world. The new UT3 editor uses additive geometry where the world is empty and objects must be added. Therefore, UT2004 worlds are

S. Balakirsky is with the National Institute of Standards and Technology (NIST), Gaithersburg, MD 20899 USA (e-mail: stephen@nist.gov)

J. Falco is with the National Institute of Standards and Technology (NIST), Gaithersburg, MD 20899 USA (e-mail: falco@nist.gov)

F. Proctor is with the National Institute of Standards and Technology (NIST), Gaithersburg, MD 20899 USA (e-mail: proctor@nist.gov)

P. Velagapudi is with the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15217 USA (e-mail: pkv@cs.cmu.edu)

[1] Certain commercial software and tools are identified in this paper in order to explain our research. Such identification does not imply recommendation or endorsement by the authors, nor does it imply that the software tools identified are necessarily the best available for the purpose.

not compatible with UT3 and must be remodeled from scratch in the Unreal Editor environment.

In UT2004, vehicles were constructed by composing multiple individual static meshes into a single object. In UT3, vehicles must now be modeled by using skeletal meshes. These skeletal meshes are modeled in a 3rd party software package such as 3D Studio Max or Maya, and then wrapped in a UT package using Unreal Editor. UT3 unreal script must then be created to support vehicle operation. The same effort is required for mission packages which are now also implemented using skeletal meshes.

Other tasks at hand include code clean-up/consolidation and architecture redesign. USARSim has been in existence since 2002 with involvement by many organizations and hundreds of individual code contributions via its open source development portal located on SourceForge [3]. This has resulted in many code formats that are often difficult to read and deprecated code that has never been purged from the implementation. A major aspect of this porting effort is to provide consistent code formatting and commenting as well as the elimination of code that is no longer in use. In addition, this port is implementing a repackaging scheme and improving naming conventions to provide a better organization of the USARSim implementation.

III. USARSim API and Units

While improvements are being made throughout the USARSim framework, a strong emphasis is being placed on maintaining backward compatibility with the previous USARSim API. USARSim currently uses SI units for all information exchanges and this policy will remain in place. In addition, the current scale of 250 Unreal Units (UU) equals 1 m will be maintained. The original intent behind this figure was to balance physics fidelity that improves with higher uu-to-m density against the maximum extent of a world (UT2 and UT3 support a rectangular world of at most 524288 UU per side). The value of 250 UU provides for a maximum world size of approximately 2 km per side.

In order to maintain this figure for UT3, gravity needed to be revalidated. Through freefall experiments, it was determined that the native relationship between UU and Meters is 104 UU = 1 m. To compensate for this, the gravity for any generated world must be set to -1245 by adding a physics volume into the map.

IV. Port Architecture & Object Communication

The class based Unified Modeling Language (UML) architecture diagram in Figure 1 depicts the progress of the current port. Green boxes with the yellow folders represent unreal script packages. Blue boxes represent native classes of interest while orange boxes represent unreal script that has been implemented as part of this porting effort. UML generalizations and associations are shown for major classes within USARSIM. Note that not all classes are shown to simplify the figure.

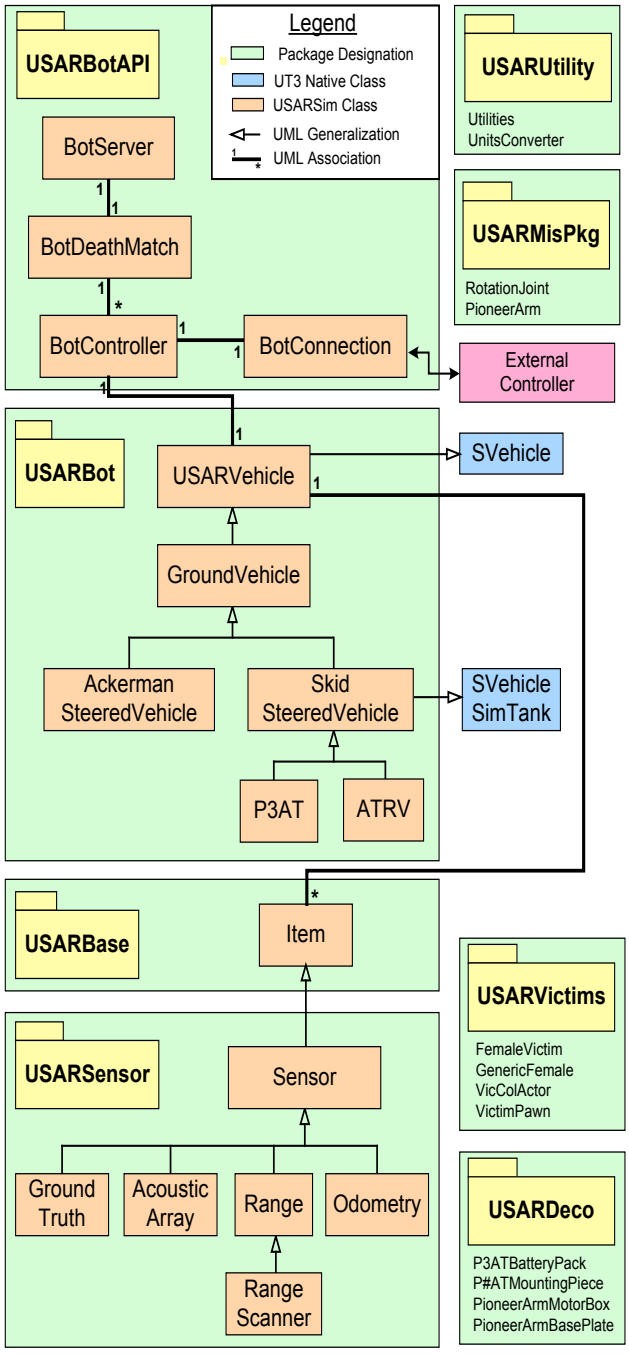


Figure 1: The UT3 Port Architecture shows the current status of the USARSim UT3 port.

This port results in the USARSim architecture being broken up into significantly more packages than the UT2004 implementation which only consisted of USARBotAPI, USARBot, USARMisPkg, USARModels, and USARVictim. The new packaging scheme

essentially breaks the massive USARBot package into smaller, more concise packages. Current packages include: USARBase, USARBot, USARBotAPI, USARDeco, USAREffector, USARMisPkg, USARSensor, USARUtility, and USARVictim.

USARBase contains base classes that define children classes appearing in more than one package. This package was created to compensate for the fact that unreal script does not support statements to makes types available within a compilation unit such as the C++ "include" statement or the Java "import" statement, but rather depends on a package compile order to recognize subsequent types. This compile order is configured in the DefaultEditor.ini file and was not an issue prior to repackaging, sensors, mission packages, and robot vehicles, when they all resided within the USARBot package. USARBotAPI contains the socket based communications API. USARDeco contains model packages for brackets and mounting plates that support sensors and mission packages. Keeping the models separate from the actual sensor and mission packages allows for reconfigurable mounting schemes. USAREffector contains gripper tools while USARMisPkg contains mission packages such as robot arms and pan/tilt units. USARSensor contains all sensors used in the simulation environment whether mounted to vehicles or a UT world. USARUtility contains all utilities such as conversion, random noise generators, and tool for querying world information. USARVictim contains all victim models used in urban search and rescue applications. Additional packages may be added as this porting activity progresses.

V. UT3 Vehicles

Vehicles in UE3 are rendered through the use of a skeletal mesh. Skeletal meshes consist of a series of bones and skins that are designed and assembled for each vehicle in a 3rd party package as shown in Figure 2. NIST is currently using 3D Studio Max for this effort [4]. A vehicle skeletal mesh is designed with a static chassis containing a root bone, weighted to the chassis, and additional bones for wheels and suspension that are direct children of the chassis bone [5]. The process of packaging a vehicle using Unreal Editor is a multi step process: (1) import the skeletal mesh (3D Studio Max requires use of the ActorX plug-in), (2) create a wheel controller, called a "SkelControlWheel", for each wheel or suspension bone and (3) assign a physics object to the vehicle chassis.

Step two above is accomplished by configuring an animation tree within the Unreal Editor. Animation trees contain "SkelControls" objects which are used to move individual bones around in the UT3 game. The UE3 vehicle system uses a special type of bone controller, a "SkelControlWheel" to move the wheel bones to match the vertical movement, roll and steering of the underlying simulation. Step three involves creating a physics asset for the vehicle where a collision shape and physical properties are assigned to the chassis. Only the vehicle chassis body is used for collision checking since wheels contact the simulation environment. The mass and inertia of a vehicle is calculated based on the volume of the collision geometry and the density property assigned to the chassis body.

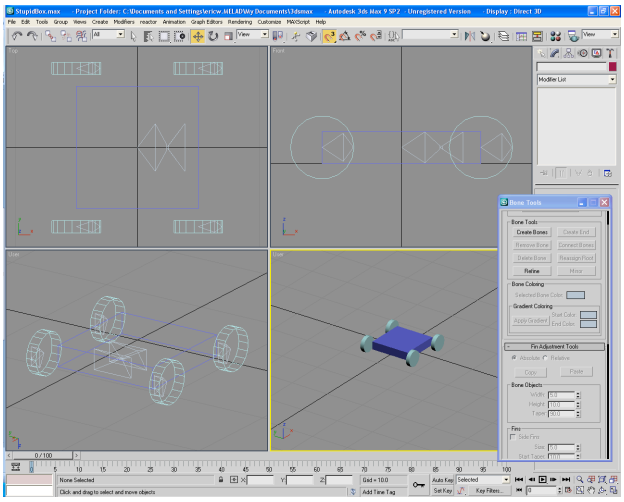


Figure 2: The 3D Studio Max software is being used to design a simple vehicle skeletal mesh for USARSim UT3 development work.

A major effort within the USARSim UT3 vehicle port is to develop a control strategy to support the same vehicle functionality that was achievable with the UT2004 implementation. UT2004 allowed for direct control of wheel speed while the UE2 computed the desired torques to maintain it. The UT3 SVehicle API does not support the direct control of wheel speed and only provides for torque commands to vehicle wheels. UE3 then determines the resulting speed, which will vary for a given torque depending on the friction and terrain. To solve this problem NIST has implemented a feedback controller and has initiated a study to enhance this controller by applying machine learning to generate a speed vs. torque model.

To calculate the torque needed to achieve a given wheel speed, a closed-loop control algorithm is needed that continually calculates the torque to be applied to the wheel based on the commanded speed and feedback from the wheel of its actual speed. This closed-loop algorithm is located in a vehicle's Tick function which executes every simulation cycle. To manage time, UT divides each second of game-play into "Ticks". A tick is the smallest unit of time in which all actors in a level are updated. A tick typically takes between 1/100th and 1/10th of a second, a time which is limited only by CPU power (i.e. the faster the machine, the lower the tick duration).

The goal of the control algorithm is to calculate the applied torque $\tau_{applied}$, so that the correct vehicle speed is achieved and maintained. However, the applied torque is only one component of the net torque that determines the vehicle's performance. The net torque τ_{net} is the sum of the applied wheel torque and that of the external world through the axle

$$\tau_{net} = \tau_{applied} + \tau_{external}$$

where $\tau_{applied}$ is the torque applied to the wheel to attain a desired speed and $\tau_{external}$ is the torque induced by the external world. $\tau_{external}$ is unknown and will in general vary widely as the vehicle moves across varying terrain and encounters obstacles. In order for a vehicle to maintain a commanded speed, τ_{net} must be zero meaning that $\tau_{applied}$ must balance $\tau_{external}$

We assume that the UT3 wheel is modeled as a simple rigid body following $\tau = I\alpha$, where τ is the torque on the wheel axis, I is the wheels moment of inertia and α is the resulting angular acceleration. The resulting angular speed at the end of a simulation cycle of duration Δt is $\omega(\Delta t) = \omega(0) + \alpha\Delta t$, where $\omega(0)$ is the speed at the beginning of the simulation cycle. Solving for torque τ for a given desired angular speed $\omega(\Delta t)$, we get

$$\tau_{net} = I \bullet (\omega(\Delta t) - \omega(0)) / \Delta t$$

Since wheel inertial information is not readily available from the physics engine, we derive it by performing a simple experiment. In this experiment, the vehicle is flipped upside down to create the effect of a free-spinning wheel. A constant torque is then applied, and the resulting acceleration is observed.

We explored PID control as a closed-loop control algorithm to provide compensation for the external torque disturbance. PID is a control method where the error between the desired set-point value and the actual measured value drives the system toward the set-point. PID refers to the three tunable parameters, or *gains*. The 'P' term generates a contribution proportional to the error. This is often good enough but can leave some residual errors in systems that have some steady-state error source. One steady-state source of error in UT3 is torque from the downhill pull of gravity. To compensate for this, an 'I' term generates a contribution due to the cumulative or integrated amount of error. For a constant steady-state error, this term will build up over time, and pull the system back toward the set-point. In systems that tend to oscillate and need damping, the 'D' term generates a contribution due to the rate of change or derivative of the error.

The existence of an accurate system model means that the PID gains can be determined analytically given some figures of merit, such as allowable overshoot and steady-state error. More often, as in our case, the model is unknown and tuning the parameters is something of an art.

A C-language simulation and experimentation in the UT3 environment were used to optimize PID control. In our case, a PI (no D) controller implemented in UT3 vehicle control yielded reasonable tracking of speed commands. Experiments with the derivative D gain did not show any improvements since in there is no natural tendency of the rigid body wheel model to oscillate.

Figure 3 shows a log of vehicle speed in UT3 for PI values of 1 and 1, respectively. The commanded speed profile was zero until 22 seconds, 1 unit per second until 110 seconds, and finally zero. Note the oscillatory behavior around speeds of zero and the steadily climbing approaches to constant speeds. Figure 4 repeats the same test with the PI gains increased to 20 and 2, respectively. The speed tracking is significantly better.

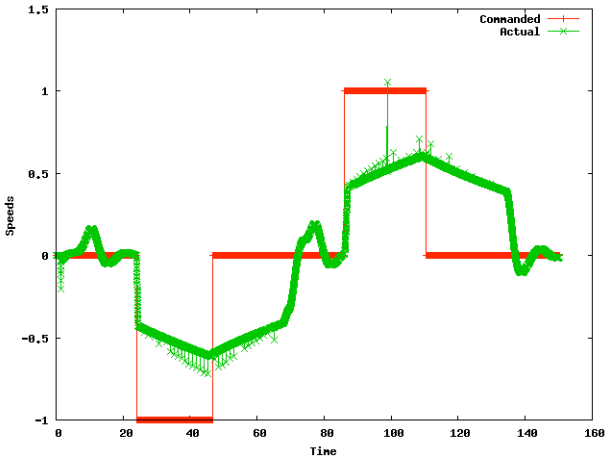


Figure 3: Speed tracking in UT3 using the PI controller, given nominal speed profiles of 1 unit/second in each direction. The PI gains were 1 and 1, respectively. Note the relatively sluggish behavior.

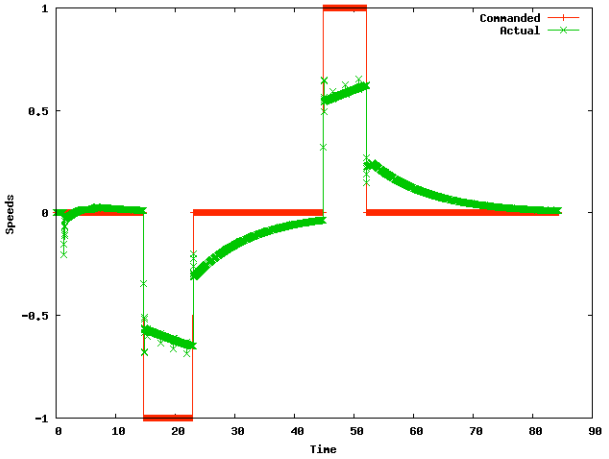


Figure 4: Speed tracking in UT3 for increased PI gains of 20 and 2, respectively. Note the improved tracking.

A problem often encountered with PI control is that the speed error must be present or have built up over time for any torque to be generated. This PI control strategy could be improved if a prediction of the necessary output can be made and added to PI terms. This control strategy is known as feed-forward control. NIST has initiated a study to implement this "feed-forward" strategy by applying machine learning to generate a speed vs. torque model. This model is being built based on the speed vs. torque relationship with respect to various UT3 terrain, obstacle and friction attributes.

VI. Mission Packages

Work is in progress to port mission packages to UT3. As with wheels, UT3 provides torque or force control of the joints connecting the individual bones that make up a mission package, such as a robot arm. If direct position control of these joints is possible, the mission package control problem is greatly simplified. In that case, the kinematic configuration of the mission package can be used to determine the joint position values needed to bring the mission package to a desired position and orientation in space. If position control is not possible, then torques (or forces) need to be generated to bring the mission package to its desired position and orientation. The torques necessary include those needed to balance the unloaded arm against gravity, plus any load at the end. As with wheels, the torque computation includes the nominal values to balance the mission package, plus contributions from a PI controller to compensate for the unknown loading on the mission package.

For either position- or torque control, the kinematic configuration of the links that make up the mission package are needed. In UT2004, the configuration was defined by a coordinate system convention in which rotational joints rotated about their Z axis, and sliding joints slid along their Z axis. The origin (position and orientation) of each link was expressed with Cartesian XYZ values for position, and roll, pitch and yaw values for orientation, with respect to the coordinate system of the previous link. This scheme will be adopted for UT3.

In this scheme, a position controller can build a kinematic model of the mission package that relates the joint positions with the resulting position and orientation at the end, and vice-versa. For torque control, the mass and center of gravity of each link are needed. In UT3 the mission package information will be supplemented with this information, extending the interface with four numbers, one for mass (in kilograms) and three for the XYZ position of the link with respect to its coordinate system.

Once the mass and center of gravity are known, a torque controller can combine this with the existing kinematic configuration and determine the torques necessary to balance gravity acting on the mission package itself. For each goal position and orientation, the torques are computed and commanded to the mission package. Ideally, this should balance the mission package exactly at the goal position and orientation. In practice, uncertainties in the environment and the presence of external loads on the mission package will prevent the arm from reaching its desired goal exactly. As with wheel control, a PI controller will be applied to generate additional compensating torques that bring the mission package close to its desired goal.

Figure 5: Actual step-test standard test method (left) and simulated version of test (right).

VII. Support for Non-Vehicle Systems

USARSim began as a project to study human-computer interfaces in urban search and rescue robotics [6]. In fact, the original acronym stood for Urban Search and Rescue Simulation. The main emphasis in the development of USARSim was placed on the creation of validated mobile robot platforms and sensors. Platforms were validated through comparisons of the simulated platform's performance on NIST's standard test methods vs. the real platform's performance on the same tests [7]. An example of a real test method and its simulated version are depicted in Figure 5. Sensors were validated with a variety of techniques that included comparisons of raw data output and comparisons of the output of seminal algorithms operating on both simulated and real data [8].

Since that time USARSim has matured to be used in many different research disciplines. This variety of research goals has driven a desire to expand the base of supported robot and sensor models. Currently, as shown in Figure 1, all robots are based on the UT3 SVehicle class. Robotic arms and sensors are then mounted to the vehicle as mission packages. While this technique works well for mobile platforms, it breaks down when one tries to emplace a fixed sensor network or stationary robot such as a robotic arm installed in a factory setting.

A current work in progress is the search for a new base class to use for a fixed robotic base. This base will allow for mission packages to be attached to a fixed location and will not have the overhead associated with a full vehicle implementation. Validation techniques also need to be developed for the new factory arms and effectors that will be included with this new class of robots.

VIII. Distributed Robot Processing

The processing required to simulate robots in USARSim can be divided into two major components. The first is the physics simulation of robots as they move around the simulated world. This entails detecting collisions, applying motor forces and joint constraints, and computing the resulting dynamics of the bodies comprising the robots. The second component is the generation of sensor data. This consists of processes such as casting rays to determine simulated range measurements, and rendering images for simulated cameras. In USARSim for UT2004, both of these processes are centralized to a single server machine. This approach simplifies implementation significantly, but limits the number of robots and sensors that can be accurately simulated. For each sensor that is added, the server must run an additional processing function per time step. This, in turn, increases the length of each time step, which decreases the accuracy of the physics simulation, ultimately leading to unstable rigid-body dynamics.

To address this concern, the design of the UT3 port is such that physics and sensor loads can be distributed across multiple machines. UT3 provides a mechanism for doing so through its built-in multiplayer system. The engine provides a client-server model with a "replication" system, which automatically creates and updates proxy objects on clients to represent "actors" on the server. By running the port of USARSim as a multiplayer game, several UT3 clients can connect to a single UT3 server and share the computational load of simulation.

In this multiplayer mode, users connect to a UT client rather than the UT server and send commands to initialize a robot model. This client requests that the physical model of the robot be created on the server. A lightweight proxy object is then passed back to the client by the UT3 engine. The position, rotation, and other relevant properties of the physical model are dynamically and automatically updated within this proxy object through the replication mechanism of the UT3 engine. With this information, clients can asynchronously simulate the output of the sensors within the model, leaving the server to solely focus on fast and accurate simulation of physics. This division of processing improves the fidelity and stability of both the physics simulation and the generated sensor data, allowing greater numbers of robots and sensors to be simulated concurrently.

IX. Conclusion

As may be observed from this paper, the original development of USARSim was performed by numerous contributors from a variety of organizations over a several year period. As a result, USARSim consists of a large number of sometimes disjoint software modules. However, the developers of these modules all share the common goal of creating a validated robotic simulation system with consistent programmer interfaces.

The move to UT3 has provided us with the opportunity to learn from previous efforts and enhance the maintainability and usefulness of USARSim. This IROS workshop is seeing the Beta launch of the new USARSim for UT3. It contains a limited set of as yet unvalidated robot and sensor models and does not yet support the full richness of the original API. As the previous USARSim development was a community effort, it is hoped that this new release will also involve a wide community. Help is needed in everything from rewriting the manual to adding and validating sensor and robot models. In addition, new areas are being explored such as end effectors and acoustic sensing. We encourage anyone interested in participating in this effort to contact one of the authors or to see the USARSim wiki located at http://usarsim.sourceforge.net/wiki/index.php/Main_Page for more information on how to become involved.

Acknowledgment

The authors would like to thank the many research groups and individuals that are contributing to this software port.

References

[1] B. Balaguer, S. Balakirsky, S. Carpin, M. Lewis, and C. Scrapper, "USARSim: A Validated Simulator for Research in Robotics and Automation", in *Workshop on "Robot Simulators: Available Software, Scientific Applications, and Future Trends: at IEEE/RSJ* 2008.

[2] S. Balakirsky, F. Proctor, C. Scrapper, and T. Kramer, "An Integrated Control and Simulation Environment for Mobile Robot Software Development", in *Proceedings of the ASME Computers and Information in Engineering Conference* New York: 2008.

[3] "USARSim Project", http://sourceforge.net/projects/usarsim/.

[4] UT3 Vehicle Skeletal Mesh Setup in 3DStudio Tutorial, http://usarsim.svn.sourceforge.net/viewvc/usarsim/Doc/Vehicle_3DStudio_UTEditor.doc.

[5] Unreal Tournament Web Portal, "Setting Up Vehicles", http://udn.epicgames.com/Three/Setting UpVehicles.html.

[6] Lewis, M., Sycara, K., and Nourbakhsh, I., "Developing a Testbed for Studying Human-Robot Interaction in Urban Search and Rescue," *Proceedings of the 10th International Conference on Human Computer Interaction (HCII'03),* 2003.

[7] C. Pepper, S. Balakirsky, C. Scrapper, "Robot Simulation Physics Validation", in *Proceedings of PERMIS 2007.*

[8] S. Carpin, T. Stoyanov, Y. Nevatia, M. Lewis, J. Wang, "Quantitative Assessments of USARSim Accuracy", *Proceedings of PERMIS 2006.*