Accurate and Flexible Simulation for Dynamic, Vision-Centric Robots[1]

Jared Go, Brett Browning, Manuela Veloso
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA, 15213, USA
jgo@cmu.edu, {brettb, mmv}@cs.cmu.edu

Abstract

As robots become more complex by incorporating dynamic stability or greater mechanical degrees of freedom, the difficulty of developing control algorithms directly on the robot increases. This is especially true for large or expensive robots, where damage is costly, or where communication bandwidth limits in-depth debugging. One effective solution to this problem is the use of a flexible, physically-accurate simulation environment which allows for experimentation with the physical composition and control systems of one or more robots in a controlled virtual setting. While many robot simulation environments are available today, we find that achieving accurate simulation of complex, vision-centric platforms such as the Segway RMP or Sony AIBO requires accurate modeling of latency and robust synchronization. Building on our previous work, we present an open-source simulation framework, ÜberSim, and demonstrate its ability to simulate vision-centric, balancing robots in a realistic fashion. The focus of this simulation environment is on accurate simulation with high-frequency control loops and flexible configuration of robot structure and parameters via a client-side definition language.

Keywords

Simulation, High-Fidelity, Dynamics, Vision-Based

1. Introduction

As robotics research advances, robots are becoming increasingly complex in terms of kinematics and dynamics, with greater emphasis on walking or balancing robots. Concurrently, there is an increasing use of vision, whether monocular, stereo, and/or color based, as a primary sensing modality. Developing the algorithms and techniques to control such complex robot platforms operating in dynamic worlds becomes a difficult challenge. For expensive, large, or otherwise fragile robots, this is especially true.

It has been well established within the agents and robotics literature that simulation can be a powerful tool for speeding up the development cycle for robot control systems. Simulation achieves this impact through its potential for faster, or slower, than real-time simulation, more powerful debugging facilities, and independence of robot hardware. For a simulation to be useful, however, it must capture the *important* characteristics of the physical world, where importance is a function of the problem in question.

We consider the problem of highly dynamic, multi-robot environments where vision is the primary sensor. In our prior research, we have spent considerable time investigating such domains with a particular emphasis on the use of simulation as a mechanism to improve the speed of the development cycle. We have found that for to have impact upon development in such environments, a number of key features are desirable. These desirable features are; physical accuracy, flexible hardware configurations, vision-based sensor interfaces, ability to simulate at faster or slower than real-time, distributed execution, high density debugging support, and easily defined client interfaces for new robot hardware. Although there have been numerous simulation engines developed within the community, with varying levels of realism, to the authors knowledge no simulator successfully contains all of these features for complex, potentially dynamically balancing robots operating in dynamic multi-robot domains.

In this paper, we report on our progress developing ÜberSim, a simulation engine targeted to address these problems. We report on the techniques we have developed to address the challenges for physically accurate simulation of a robot operating in a dynamic environment with vision as its primary sensor. Concretely, we have implemented our techniques to create a simulator for the Segway RMP robot, a

AAMAS '04, July 1-2, 2004, New York City, NY.

1 A demonstration of this system will be available to be shown at the conference.

dynamically balancing robot based on the commercialized Segway scooter that uses vision as its only external sensor (see Figure 1). We are using this robot in a human-robot soccer problem – a highly dynamic, multi-robot domain where simulation can be a powerful tool to develop robot control algorithms.

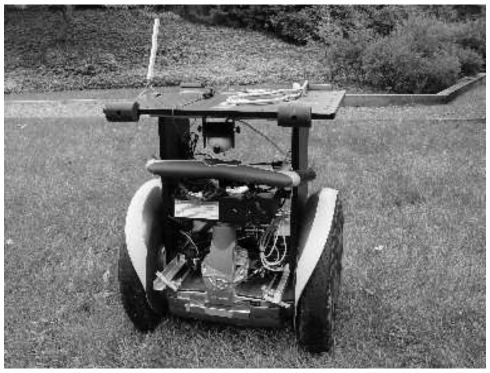

Figure 1. The Segway RMP robot.

In the following section, we first review the background and motivation for out work, as well as the relevant related work. In section 3, we then overview the approach we have utilized in our work with ÜberSim, as well as the the technical details for simulating a dynamically balancing robot operating in a dynamic world. In section 4, we examine the empirical results and proceed to examine related work in section 5. Section 6 concludes and describes our future work.

2. Key Challenges

When building a simulation environment that is useful and realistic for vision-centric robots such as the Segway RMP, several main challenges must be considered. These issues include:

1) realistic robot dynamics and simulation accuracy,
2) support for high-frequency control loops,
3) accurate vision synthesis with occlusion and artifacts,
4) latency modeling,
5) flexible and extensible robot specification.

The first two challenges in building a simulation environment for a robot such as the Segway RMP relate to dynamics and control loops. By default, the Segway RMP exists in an unstable state and is only kept stable via a tight control loop that solves the inverted-pendulum control problem in real-time. As this behavior is intricately tied to the dynamics of the robots, simulating this behavior requires *both* the ability to simulate accurate dynamics while also accurately simulating the effects of changes in actuation at high frequencies. Simulations that fail to meet these two criteria will be unable to generate physically plausible motion for simulated robots such as a Segway RMP.

Vision synthesis is also equally important as it allows for correct occlusion and visual artifacts, unlike other methods of passing scene data directly to the simulated robot. This markedly increases the applicability of the simulation to the real-world, as the simulated client's control loop no longer needs to incorporate an alternate scene representation, and thus simulated control algorithms can translate more directly into real-world control algorithms.

Another key element of a simulation is the inclusion of latency modeling. The effects of latency are often omitted in simulations, but are essential to the faithful simulation of large and unstable robots, such as the Segway RMP. Since the Segway RMP is capable of moving at up to 20 km/h (5.5 m/s), a latency of 100ms in vision systems (common without dedicated hardware) can induce a positional error of more than half a meter which must be accounted for in the simulation in order to match behavior in the real world. Furthermore, unaccounted latency in the actuation or sensing components of the Segway balance mechanism can lead to simulated results that quickly diverge from reality even under the same control code.

Another challenge in building an effective simulation environment relates to the interface for building virtual robots and integrating their control code with the simulation. Ideally, the integration should be seamless and should maximize reuse of existing control code in the simulation. Furthermore, the specification of the virtual agent's composition should be done in a simple and intuitive manner.

High-Fidelity Simulation – Our Approach

ÜberSim addresses the various challenges with a combination of mature, proven technologies as the cornerstones for the simulation framework (see Figure 2).

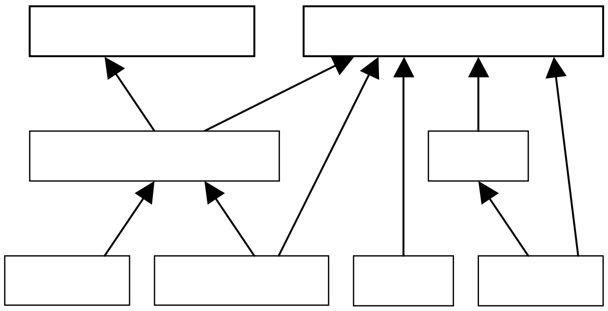

Figure 2. Overview of components in the ÜberSim framework.

ÜberSim provides realistic dynamics and simulation accuracy by its use of a rigid-body physics engine known as the Open Dynamics Engine (ODE). This library simulates high-fidelity dynamics and interactions between rigid bodies, which allows ÜberSim to generate physically plausible simulated motions.

High-frequency control loops are handled in ÜberSim by the simulation and network protocols, which simulate forward conservatively so as to ensure that the client and server components of the simulation remain synchronized at all times. The simulation can also be run at a configurable rate which can be tuned to match with the frequency of sensors in the simulation, allowing users to select a balance between accuracy and speed depending on the requirements of their simulated environment.

In order to address the issue of latency, ÜberSim provides a variety of latency models which can be easily configured for each simulated robot and allow for correct simulation of sensor and actuator latency. This further reduces translation time between virtual robot and real-world robot as the latency is built into the simulator and does not need to be added manually into the client control code.

Finally, in order to present a streamlined interface, ÜberSim provides an XML description language for specifying the composition of a virtual robot. Modifying the robot's composition requires only small changes to the robot's XML definition and does not require recompilation of the server. Furthermore, modification and recompilation of client code is also unnecessary if the changes do not affect the existing control algorithms, such as is the case for most modifications to physical structure. New components, such as different types of sensors and motors, can be easily added to the server and subsequently used by the client.

3.1 Core Simulation Engine Components

The ÜberSim simulation engine integrates several open source technologies for graphics and simulation. The Open Dynamics Engine (ODE) provides accurate collision detection and rigid-body dynamics while the Open Scene Graph (OSG) library provides the scene graph and representation. OpenGL is also used directly in combination with OSG's rendering pipeline in order to achieve other effects such as rendering views from simulated camera sensors.

ODE provides the ÜberSim server with accurate dynamics, including varying surface and material properties. Objects in the simulated world are tagged with a particular material, and a user-defined material matrix in the server determines the properties of contacts between various materials. Given the set of properties, contact joints are created in ODE and the collisions are resolved appropriately. The library also provides fast, two-phase collision detection routines which we make use of in ÜberSim.

The OSG library provides a robust and powerful scene representation and provides built-in features such as culling and scene traversal mechanisms. It is also highly extensible and can make use of newer graphical features such as vertex and pixel level processing, which can be leveraged to generate realistic renderings. OSG also allows for direct access to the underlying OpenGL routines, which is advantageous for implementing graphical techniques that have not yet been integrated into OSG.

Finally, the ÜberSim server uses the Apache Xerces XML parser to perform all XML parsing. Specifically, the library is used to load configuration files and to parse XML sent by ÜberSim clients.

3.2 Vision Synthesis

Vision synthesis is handled by the combination of OSG and OpenGL, the former of which is used to maintain the graph of objects in the scene as well as the rendering properties of the various objects. When a robot requires a view of the world, we use the Open Scene Graph rendering pipeline but render to an offscreen surface (P-Buffer) and read back the appropriate data. This ensures that as the visual simulation quality improves through the addition of more complex textures, shaders, and lighting models, the simulated view will become correspondingly more accurate. Additional effects such as noise and radial distortion can be easily added to the rendering pipeline for to achieve greater simulation fidelity.

3.3 Client/Server Architecture

ÜberSim maintains the traditional client/server paradigm found in many other simulations, but shifts the responsibilities of the client and server in order to provide increased interchangeability between the simulation control code and the physical robot control code. In our architecture, the server source code contains primitives, sensors, and actuators which have various parameters that help maximize reuse. For example, primitives include solids such as rectangular prisms, spheres, and capped cylinders which can have various mass, extent, and material properties. Sensors include such devices as cameras and tilt sensors, while actuators include various kinds of translation and rotation motors and constraints. Once the primitives necessary for a particular robot have been integrated into the ÜberSim server, the server can remain unchanged during client development and testing.

The ÜberSim client program's responsibilities essentially involve two components. The first involves a process we define as *self-instantiation*, whereby the client establishes a TCP socket connection to the server and transmits an XML document that describes the various primitives, sensors, and actuators that compose the simulated robot. The second involves processing incoming sensory messages and sending the correct actuation messages as determined by the robot's control code.

The ÜberSim server's responsibilities are similarly two-fold. When a client first connects, the server is responsible for converting the client's XML document into the actual simulated components. As long the server is running, it is also responsible for simulating forward, using ODE to determine the next world state. At each simulation step, for every sensor primitive in the world, the server determines if the sensor should generate output and if so generates the appropriate output and sends it to the client in a predefined form. For example, a camera sensor might generate a rendering of the scene and send the compressed image to the clients, while an orientation sensor might pass orientation information back to the client.

3.4 Latency and Synchronization

Much work has been done on simulation synchronization architectures and protocols. As with many other simulators, ÜberSim is a conservative simulator in that the system guarantees the same simulation regardless of client and server speed, with differences in speed simply reducing to differences in the rate at which ÜberSim can simulate. This allows for the simulation of high-frequency control loops on machines of varying processor speed and bandwidth.

Synchronization between the client and server simulation is handled by timestamp messages sent by the network protocol. For each simulated frame, the server sends out a small message containing a floating-point timestamp of the current simulation time. The client program should store this timestamp and, when sending actuation commands, prefix them by a timestamp message telling the server at which instant the messages were sent. The client must acknowledge the processing of each frame with a response message informing the server that the client has completed the necessary processing for that frame.

We model latency by maintaining a server-side delay queue for each client. As the server receives actuator messages from the client, it places them in the delay queue with an execution timestamp equal to the message's timestamp plus the system latency parameter. The server then actually executes the actuator message when the simulation time matches the execution timestamp. Latencies can also be added to sensors to delay the sending of sensory messages. As a side benefit, the server-side latency model is also advantageous as it eliminates the need for every client to implement a delay queue to simulate the effects of latency.

3.5 Configuration and Client Interface

In designing the ÜberSim client interface, our main goals were to present an interface and network protocol that are generic enough to support all forms of sensory and actuation messages, while remaining unified and extensible.

The model that ÜberSim adopts with respect to virtual representation is that each client, upon connecting, describes itself by a set of uniquely named *components* of varying types, each of which has a set of associated properties per-instance and defines a common input and output interface. These components are divided into two categories – *primitives* and *joints*.

The first category of components, primitives, are affixed to a single rigid body and are appropriately transformed and simulated in accordance with the motion of that rigid body. It is important to note that one rigid body may contain more than one primitive, allowing for the composition of simple objects to form more complex ones. Examples of primitives include physical solid components, such as rectangular prisms or spheres, whose extents and masses together define the rigid body's collision behavior. The primitive category also includes sensors such as inclinometers and cameras whose sensory output depends on the current position and orientation of the sensor in the world.

Joints, on the other hand, are not directly placed on any particular rigid body and are considered not to have an actual position. Instead, they represent a constraint or motor connecting to two rigid bodies. Examples of joints include slider motors, hinges, ball-and-socket joints, and angular motors.

The component architecture is advantageous as each component can potentially send sensory messages or receive actuation messages, so long as the component is compiled into the server with the appropriate sending or receiving behavior. This allows for tremendous flexibility and extensibility of components; as an example, a slider motor could be programmed to send periodic messages which describe its current extension length, or a camera could be modified to allow it to accept actuation messages that change the brightness or field of view of the camera at runtime.

A virtual robot is composed of several rigid bodies, each with one or more primitives, along with a set of joints that describe the connections between the rigid bodies. The actual interaction between client and server, after initial configuration is complete, consists of sensory and timestamp messages being sent from server to client and actuation messages being sent from client to server. These messages follow a simple and generic network protocol, loosely modeled around Remote Procedure Call protocols. This protocol is extremely lightweight and

Each network message consists of a header and a body. The header is simply a 4 byte unsigned integer which denotes the length of the body in bytes. The body itself begins with a null-terminated string indicating the name of component that the message comes from or is intended for. The remaining body contents, up to the header-specified length, are simply bytes whose meaning depends on the component type.

3.6 Self-Instantiation XML

The self-instantiation XML forms the foundation for the client API by determining the initial configuration of the components that compose the virtual robot. The XML document contains a single top-level node, named *Client*, which in turn has two children nodes, *CSettings* and *Entity*.

The *CSettings* node contains data relevant to the high-level client configuration, such as the name of the client, as well as the system latency to be used when processing actuator messages, specified in milliseconds.

The *Entity* node contains data which describes the virtual robot, and contains two nodes, *RigidBodies* and *Joints*, which describe the primitive and joint components of the robot respectively. The *RigidBodies* node contains a list of named *Body* objects, each of which contains one or more named primitives (see Figure 3). Each body object is treated as the union of the various solid primitives and sensory primitives that compose it. The primitives within each rigid body are specified in a user-defined global model space, and the center of mass and inertia tensor matrix are computed automatically. This frees designers from having to manually arrange primitives so that their center of mass coincides with the origin. The *Joints* node, on the other hand, simply contains a list of joint components which internally denote the bodies that they connect (see Figure 4). These joints are setup using the initial position of the rigid bodies that they connect, further simplifying the setup process for the designer.

```
<Body name="MainBody">
    <Box name="Platform">
    <Mass> 15.0 </Mass>
    <XSize> 0.4 </XSize>
    <YSize> 0.28 </YSize>
    <ZSize> 0.08 </ZSize>
    <Position>
        <x>0.0</x> <y>0.0</y> <z>-0.03</z>
    </Position>
    <Material> Plastic </Material>
  </Box>

  <CameraSensor name="Cam">
    <Fov> 42.0 </Fov>
    <Position>
      <x>0.0</x> <y>0.12</y> <z>0.55</z>
    </Position>
  </CameraSensor>
</Body>

<Body name="LWheel">
  <Sphere name="B1">
    <Mass> 3 </Mass>
    <Radius> 0.15 </Radius>
    <Position>
        <x>-0.36</x> <y>0.0</y> <z>0.0</z>
    </Position>
    <Material> Rubber </Material>
  </Sphere>
</Body>
```

Figure 3. Two sample body objects.

The primitives included with the base ÜberSim server are as follows. *Box* primitives are boxes with selected height, width, and depth, and with constant density and a specified total mass. *Sphere* primitives are defined similarly, but with a radius parameter instead of box dimensions. *Capped Cylinder* primitives are originally aligned with the z-axis and are defined by a radius and height.

As the original goal of ÜberSim was to simulate the Segway RMP, the current sensors integrated with the simulation are inclinometer and camera sensors. All sensors can be configured with a particular update frequency. The inclinometer sensor measures absolute pitch, and sends back the pitch as a binary floating point number in radians, in order to avoid inefficient string parsing. The camera sensor renders the scene, reads back the framebuffer data and sends it to the client in uncompressed BGRA form.

```
<Joints>
  <JointHinge2 name="LMotor">
    <Body1>MainBody</Body1>
    <Body2>LWheel</Body2>
    <Axis1>
        <x>0.0</x> <y>0.0</y> <z>1.0</z>
    </Axis1>
    <Axis2>
        <x>1.0</x> <y>0.0</y> <z>0.0</z>
    </Axis2>
    <AnchorPos>
        <x>-0.36</x> <y>0.0</y> <z>0.0</z>
    </AnchorPos>
  </JointHinge2>
</Joints>
```

Figure 4. XML Specification of a Hinge-2 Joint.

The joints included in ÜberSim correspond to the joints available in ODE. These joints include a ball-and-socket joint, a hinge joint, a hinge-2 joint, and a slider joint. The hinge-2 joint is particular to ODE and involves a car wheel-like joint in which one body (wheel) can rotate freely about its own axis (wheel axle) while being rotated or compressed along another axis (steering axle). Depending on the type of joint, the XML description specifies various axes or anchor positions for the joint. In addition, each joint specifies the two bodies which it connects, using the unique names assigned to each body.

At runtime, all of the joints currently implemented in ÜberSim have an identical actuator API and send no sensory information back to the client. The actuator message for these joints contains a total of eight bytes; the first four bytes identify the indexed property of the joint to modify, and the last four bytes contain the new floating-point value of the target property. Examples of properties include the target velocity of the joint (linear velocity for sliders, angular velocity for hinges and angular motors), the maximum force that the joint can exert, as well as the high and low limits of the joint.

4. Results

The ÜberSim server was implemented in C++ and a simple client was written in C#. The client simulates a virtual Segway RMP with an inclinometer and a camera affixed to the front of the unit. Dynamic balance is maintained using a simple PD controller where the controller gains were determined through experimentation. The message processing loop implemented in the client requires only a few lines of code and is extremely compact, owing to the simple network protocol between client and server.

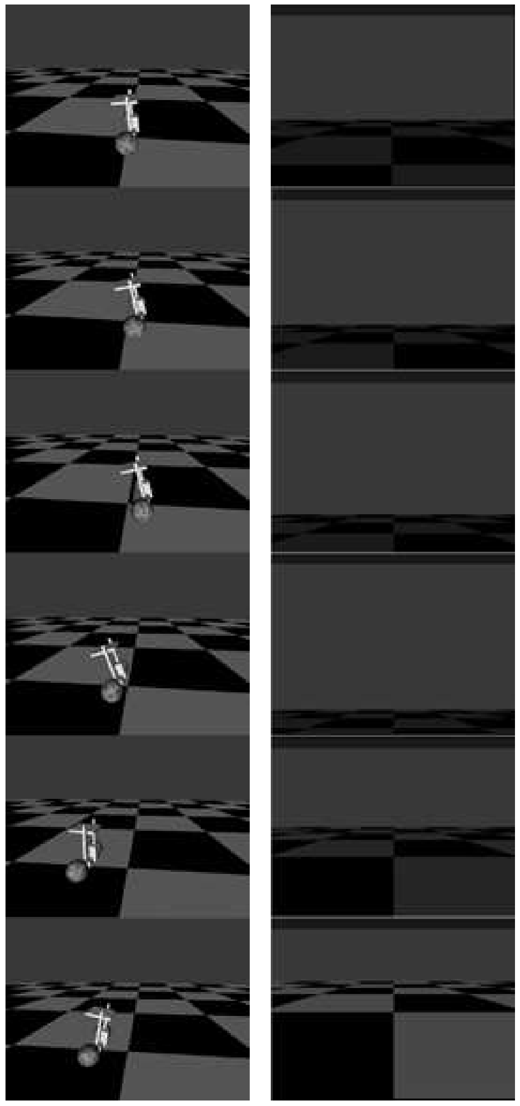

Figure 5. Simulation view of Segway RMP (left) and client-side onboard camera view (right) during balancing.

On startup, the client implementation reads in an XML file containing the robot's configuration and sends it to the server, followed by the appropriate termination characters to mark the end of the XML. Afterwards, the client implementation enters into a simple loop, responding to the inclinometer messages with appropriate wheel actuation messages determined by the PD controller. The client implementation also allows the user to control the desired balance angle of the Segway in order to cause it to move forwards and backwards.

Figure 5 shows a sequence of images during one simulation in which the robot balances fairly steadily for the first three images, with a slight forward velocity. The simulated Segway initially leans backward in order to balance as the simulated Segway's center of mass is towards the front of the robot. Afterwards, the user manually induces the Segway to balance towards the rear for a short period of time, and then towards the front, causing the behavior visible in the last three images. The time between subsequent images is approximately one second. The left images have been captured from the ÜberSim server visualization; however, the right images have been captured from the C# client implementation, which obtained the pixel data via camera sensory messages sent over the TCP connection.

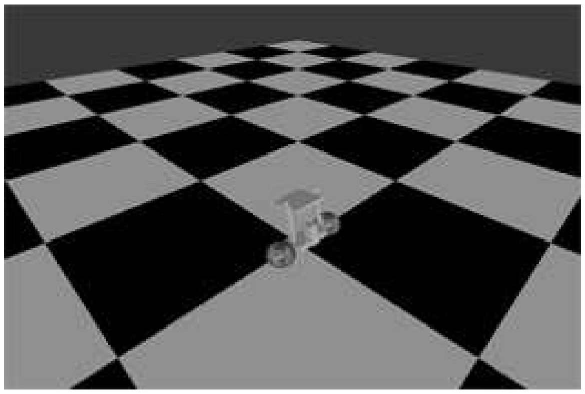

Figure 6. Another view of the simulated Segway RMP.

5. Related Work

Many simulation environments have already been developed with varying degrees of comprehensiveness and fidelity. For the purposes of this paper, we focus on those simulators that attempt to model the real world in sufficient level of detail to provide a good basis for developing algorithms for a balancing, vision-centric robot. As many simulators only provide virtual environments consisting of a 2D plane, this immediately removes these from consideration. Examples of such simulators include TeamBots [1,2], Stage [3], and the Soccer Server [8]. Although these simulators are of use in many robotics problems, they lack the realism required for robot domains where dynamics is an integral feature. As we are particularly interested in domains such as robot soccer with dynamically balancing robots, dynamics is a critical feature.

One class of 3D simulators involves those derived from 3D game engines such as the Unreal^{TM} engine. Prime examples include Gamebots [4] and the Urban Search And Rescue (USAR) robot simulator [9]. Although visually compelling and offering a truly 3D environment, the dynamics in such simulations are only approximate at best. As such, these simulators lack sufficient accuracy to enable development of robot control algorithms to transfer from simulation to reality with any confidence.

Other simulators, such as Gazebo [6, 12] and Webots [11], integrate with the Open Dynamics Engine as we have used here, thereby providing dynamically accurate motion. Moreover, these simulators provide a variety of actuator primitives and support a wide range of sensor suites, although vision is not always supported. In the case of Webots, the product is commercial, which offers some advantages but also some disadvantages in terms of complete knowledge of the underlying algorithms. Additionally, these systems do not account for latency or provide integrated methods for modeling latency in robot actuation or perception. Some of these systems also use code-based model authoring, which can lead to programmatic errors that are difficult to find. ÜberSim maintains a clear separation between the specification and actuation components functions of the simulated client in order to maximize compatibility with existing control loop code.

6. Conclusions and Future Directions

We have presented ÜberSim, a simulation framework intended for high-fidelity simulation of robots in highly dynamic environments. We have also demonstrated how ÜberSim addresses several important challenges with respect to building simulations for dynamic, vision-centered robots.

Looking forwards, our next focus is to integrate the actual Segway RMP balance code with the simulated Segway and to examine the similarity between the simulated and actual motion under various conditions. Conversely, we are also working to develop control code on the simulated Segway and then transfer it to the actual Segway RMP to analyze the simulation applicability in greater detail. Information gained during these two experiments will be used to refine the simulator.

From a project perspective, we also intend to continue work to enhance the graphical realism of the simulator's rendered output, as well as examine the scalability of the system with larger numbers of simulated robots. As we are interested in simulating other robots such as the Sony QRIO, we will also be adding new actuators and sensor primitives which will complement the existing set of components in ÜberSim.

References

[1] Balch, T. *Behavioral Diversity in Learning Robot Teams*. Ph.D. Thesis, College of Computing, Georgia Institute of Technology, 1998.

[2] Balch, T. JavaSoccer. *RoboCup-97: Robot Soccer World Cup I*, Springer-Verlag, 1998.

[3] Gerkey, B.; Vaughan, R. T.; Howard, A. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor

Systems. *Proceedings of the 11th International Conference on Advanced Robotics*, June 2003 (ICAR'03), 317-323.

[4] Kaminka, G. A.; Veloso, M.; Schaffer, S.; Sollitto, C.; Adobbati, R.; Marshal, Andrew N.; Scholer, Andrew, S.; and Tejada, S. 2002. GameBots: the ever-challenging multi-agent research test-bed, In *Communications of the ACM*, January 2002.

[5] Kitano, H.; Asada, M.; Kuniyoshi, Y.; Noda, I.; Osawa, E.; & Matsubara, H.; RoboCup: A Challenge Problem for AI and Robotics. *RoboCup-97: Robot Soccer World Cup I*, Nagoya, LNAI, Springer Verlag, 1998, 1-19.

[6] Koenig, N.; Howard, A. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. *IEEE International Conference on Robotics and Automation*, under submission, 2004.

[7] Michel, O. Webots: a Powerful Realistic Mobile Robots Simulator. *Proceeding of the Second International Workshop on RoboCup.* LNAI Springer-Verlag, 1998.

[8] Noda, I.; Matsubara, H.; Hiraki, K; & Frank, I; Soccer Server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12, 1998, 233-250.

[9] 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)*, Crete, Greece, June 22-27, 2003.

[10] Kitano, H.; Tadokoro, S.; Noda, H.; Matsubara, I.; Takhasi, T.; Shinjou, A.; and Shimada, S. 1999. Robocup-rescue: Search and Rescue for Large-Scale Disasters as a Domain for Multi-Agent Research, In *Proceedings of the IEEE Conference on Systems, Men, and Cybernetics*, 1999.

[11] Cyberbotics Webots Home Page. *http://www.cyberbotics.com/products/webots*

[12] Gazebo Sourceforge Page. *http://playerstage.sourceforge.net/gazebo*