ÜberSim: A Multi-Robot Simulator for Robot Soccer

Brett Browning[*] and Erick Tryzelaar[**]
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA, 15213, USA
+1 412 268 6021

[*]brettb@cs.cmu.edu, [**]erickt@andrew.cmu.edu

ABSTRACT
A realistic simulation engine can be a powerful tool for speeding up the software development time cycle for robot control systems. To be useful, the simulation engine must capture, at a suitable level of resolution, the interfaces, structure and dynamics of the real world that are important to the performance of the control system. In this paper, we present a new multi-robot simulation engine, called ÜberSim, for simulating games of robot soccer. The goal of ÜberSim is to create a simulation environment that enables control systems to be developed rapidly and transferred to the real system with minimal change in behavior. For robot soccer, where dynamics play an integral role in robot behavior, ÜberSim must capture at a reasonable level of resolution the physical interactions between the robots, field, and ball. Thus, ÜberSim has been built on top of a high-fidelity physics simulation engine that models friction between object surfaces, elastic collisions between objects, and finite accelerations of objects with non-negligible mass and rotational inertia. We present the details of ÜberSim in its current form and describe its use for developing robot control systems. As a means to evaluate the simulation, we present some empirical comparisons between the performance of robots operating in the ÜberSim simulation and robots operating in the real world.

Keywords
Simulation, Multi-robot, Multi-agent, Robot soccer, RoboCup

1. INTRODUCTION
Developing new control software for robot teams can be a difficult and challenging task. Testing and debugging robot software is often a long and tedious process. Whether it is from limited, or no, communication bandwidth or the extra effort required keeping robots running through battery changes, code development on a robot system is often much slower than development within a simulation environment. Clearly, the ability to rapidly prototype software within a simulation environment can be of great benefit to developing robot control if the resulting software can be transferred from simulation to the real thing with minimal overhead.

For software developed in simulation to be transferable to the real robot, the simulator must capture the characteristics of the environment that are important. Of course, what characteristics are important is dependent upon the task the robot control software, and the environment.

In this paper, we are primarily interested in adversarial multi-robot domains, or more specifically, small-size robot soccer [7]. Small-size robot soccer is a fast moving game played by autonomous heterogeneous robot teams. In such an environment, to reliably develop control software in simulation that is to be transferred to the real system, the simulation engine must simulate realistic dynamical interactions between the different objects in the environment. Secondly, the heterogeneous nature of the robot teams means that it should be easy to add and reconfigure robots in the simulated environment. Nearly all freely available multi-robot simulation environments use low-fidelity dynamics models and rarely consider collision dynamics. Hence, most robot soccer teams develop custom simulation environments to meet their needs. However, developing simulation engines with high-fidelity dynamics models and collision models that can be simulated in real-time is a non-trivial problem. Moreover, if the simulation environment does not provide facilities to easily add and reconfigure robot types, its use by other researchers will be limited making the exercise a high cost to reward venture.

To address this issue we have begun an Open Source project, called ÜberSim, to develop a publicly available robot soccer simulation engine with high-fidelity dynamics and collisions models and extensible robot classes. The ultimate goal for ÜberSim is to produce a useable simulation engine capable of simulating a wide variety of robot types ranging from small-size soccer robots to legged robots such as the Sony AIBO. In this paper, we describe the ÜberSim approach and delve into the details of the first release of ÜberSim. The current version of ÜberSim, available at [11], implements a small-size robot league simulation engine. It provides a high-fidelity simulation environment and re-configurable robot classes for differential drive robots and three-wheeled omni-directional robots. The simulation engine interconnects with the CMDragons'02 small-size robot software, as used at RoboCup 2002, which is available at [11] (see [3,4] for more details about the CMDragons'02 software). We then present some empirical demonstrations of the simulator in action.

The following section describes the motivation and approach for ÜberSim. Additionally, the section specifies the requirements that ÜberSim must meet to be useful. Section 3 describes the technical details of the ÜberSim implementation. Section 4 presents some empirical results used to evaluate the simulator and demonstrate its operational capabilities. Section 5 describes the similarities and differences between ÜberSim and the related literature. Finally, section 6 concludes the paper.

2. APPROACH AND SPECIFICATIONS

In this section we describe the motivation for ÜberSim, the approach we have taken, and the concrete specifications for the small-size implementation that is the focus for this paper.

2.1 ÜberSim Motivation and Approach

Many robotics researchers make use of simulation tools, whether developed in-house or publicly available, to speed up the software development cycle. We argue that such simulators are really only as useful as their ability to simulate the characteristics of the world that are important to the control task at hand. If a simulator does not suitably capture the interactions of the simulated robot with its surrounding environment, then the ability to develop control systems that can be easily transferred to the real robot will be hindered. What makes a simulator useful is its ability to support meaningful code development that can be transferred to the real robots easily. This means the more realistic a simulator is, without adversely affecting its ability to simulate at real-time speeds or faster, the more useful a simulator is. What characteristics of the environment need to be simulated in high fidelity depends upon the environment, the task requirements, and the robot's action and perception capabilities. ÜberSim is primarily intended for use as a high fidelity robot soccer simulator that enables rapid development of software control systems that can be transferred to real robots with a minimum of overhead.

RoboCup robot soccer [7] is an adversarial multi-robot research domain where autonomous teams of robots compete against one another in a game of soccer. The domain has a broad range of leagues including small-size robots (< 18cm), mid-size robots (<50cm), Sony AIBO's, simulated agents, and humanoids. Our group has participated at RoboCup competitions virtually since its inception. Throughout, we have made heavy use of simulators as a tool for rapidly prototyping software. A simulator can enable one to develop software without the unnecessary overhead of running real robots and then transfer the software to the real system for final testing and fine-tuning. Additionally, a simulator limits resource conflicts, as the robots are a shared resource, from slowing down development and allows development to continue even if the robots are temporally unavailable. Clearly, simulators can have a powerful impact on development productivity. Indeed, we argue that the utility of a simulator is a function of how much it improves the development rate.

Robot soccer, the target domain for ÜberSim, is a highly dynamic task with a broad range of heterogeneous robots even within a single league. Moreover, it is a very competitive domain meaning that robots are typically pushed to the limits of their capabilities and new hardware innovations are constantly appearing. Thus, realistic environment dynamics and extensible, re-configurable robot classes are the keys to building a useful robot soccer simulator for developing robot control software. Realistic dynamics include motions and physical interactions between robots and the environment, robots and the ball, robots and other robots. Realistic dynamics are required to enable control systems to be developed in simulation that explore the boundaries of what a robot is capable of. If there is a mismatch between a robot's simulated capabilities and its real capabilities, then it is most likely that control software will work as designed in simulation but fail on the real thing. Extensible, re-configurable robot classes are required because of the diversity of robot types and capabilities even within a single league. Moreover, new hardware is always being developed so the simulated robot configuration and capabilities need to be regularly revised.

The first question that one naturally asks is: "Are the simulators available that fit these requirements?" To our knowledge, the answer is no. There are no publicly available simulators that allow simulation of robot soccer at a high enough fidelity to support useable software development. Indeed, many teams across the competition traditionally develop in house simulators to fit their requirements. The problem with this approach is that it is inefficient in terms of cost to benefit. As a result many poor quality simulators are developed rather than a few high-quality simulation engines.

The ÜberSim approach to this problem is distinctly different. Firstly, we created ÜberSim as an Open Source project in the hope that wide availability of the simulator will concentrate development to produce a more complete, broadly useable, simulator as opposed to many partially useable simulators. Secondly, ÜberSim is built around a maturing Open Source high-fidelity simulation package, the Open Dynamics Engine [9]. By using an existing physics simulation engine some of the development effort is negated, and we ensure that ÜberSim will have realistic, correct, and fairly complete physics models. Clearly, the latter point is critical to accurately simulating the world in a high-fidelity manner. A third part of the ÜberSim approach is robot extensibility. ÜberSim is targeted towards providing parameterized robot classes that are easy to extend and re-configure. As ÜberSim is built on a rigid body simulator, the robot shapes and actuators are generic enough to allow simulation of a wide range of robot types. Hence, one goal of the ÜberSim project is to provide a structure that makes adding new robot types or parameter modification of an existing robot types easy.

Ideally, we would like ÜberSim to become a cross-league development tool in the long term. In the shorter term, we desire ÜberSim to be useful enough to operate as a generic small-size robot soccer simulator. In this paper, we present a first step towards this goal by building a simulator for an existing small-size team. In the next section we describe the specific requirements for ÜberSim to operate as the simulator for our small-size team CMDragons [4]. This naturally leads to section 3 where we describe the ÜberSim implementation for this task.

2.2 ÜberSim Small-Size Specifications

The small-size robot domain consists of two teams of five robots playing soccer with an orange golf ball on a 2.8m x 2.3m carpeted field. Each robot must fit within a 18cm diameter cylinder that is 15cm tall and may have color markings for identification purposes. The small-size league is characterized by its allowance of off-field computers for processing and overhead cameras for 'global vision' perception purposes. Although not individually autonomous, each team is an autonomous entity where the off-field computer(s) communicate to the robots via radio. Figure 1 shows a typical set up.

The task for ÜberSim is to provide an identical interface to the small-size robot control software such that control software cannot distinguish between reality and simulation. Three interfaces are required: Perception and action interfaces, and a control interface for specifying the environment configuration and controlling simulation execution. The first part of the specification is to determine at what level the perception and action interfaces

operate. The choices range between simulating the raw input and outputs of the system (i.e. camera input and voltage output to the actuators) to simulating the interfaces at some higher level of perception processing and action generation.

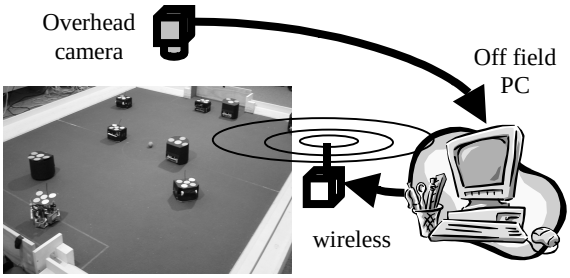

Figure 1. A typical small-size robot soccer game set up.

Due to the use of global vision and off field processing in small-size robot soccer, a nature choice is to simulate the output of high-level vision and the radio interface used to send commands to the robots. There are a number of motivating factors for this choice. Firstly, many small-size teams make use of a vision server that produces fairly similar output information (e.g. [4]). Similarly, many teams use some form of radio server program that accepts robot relative velocity commands for each robot. Typically, each robot locally runs servo control loops of some kind to maintain a commanded velocity making the robots essentially remote controlled vehicles. Figure 2 shows the resulting interfaces that ÜberSim must simulate.

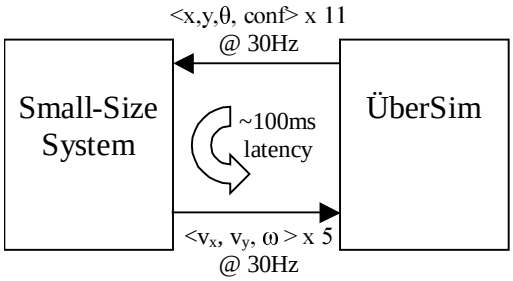

Figure 2. ÜberSim block diagram

In addition to providing the appropriate parametric interface, ÜberSim must simulate the appropriate dynamical properties of the system. For the interfaces ÜberSim must limit the I/O rate and provide non-zero latency to match that of the real system. The parameters that control the latency and data rate will vary according to the system being modeled. For the CMDragons'02 system high-level vision produces frames at 30Hz and a total system latency of around 100ms (the latency distribution across the system is unknown). [3] describes the details of the vision output and its noise artifacts. [4] describes the motor commands which are also shown in Figure 5.

There are two types of robots: differential drive robots and three-wheel omni directional drive robots. Each of the robots has a different shape, mass, and rotational inertia. Each robot has different acceleration and peak velocity capabilities by virtue of their different masses, motors, wheel types and configurations. Finally, each robot is equipped with a dribbler mechanism (a rolling bar coated in rubber used to spin the ball backwards and thereby control it), and a kicker. Again, the differential drive robots have different dribbling and kicking capabilities to the omni directional ones.

3. ÜBERSIM IMPLEMENTATION

In this section we describe the implementation of ÜberSim to simulate small-size robot soccer for the CMDragons'02 small-size software. We focus on the technical aspects of the simulation engine. The source code for both ÜberSim and CMDragons is available, under the GNU Public License, on line at [11].

3.1 Simulator Overview

Figure 3 shows the modules for ÜberSim. Essentially, ÜberSim is built around the Open Dynamics Engine physics simulation library to provide the physics simulation and parameterized object classes to define each physical object in the simulator. The scene graph forms the primary data structure for containing all the ODE relevant and simulation relevant information. A collision manager provides efficient potential rigid body collision detection and using the low-level primitives provided by ODE updates the scene graph to produce the appropriate collision response. The main controller is the orchestra conductor for the system. The controller maintains the communication interfaces to the robot software, generates simulation and collision detection calls at the appropriate time, and performs system configuration as required.

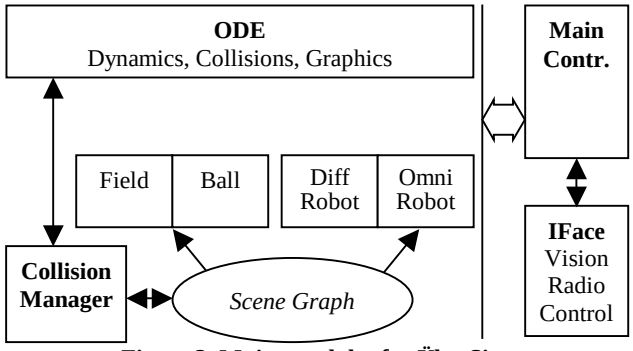

Figure 3. Major modules for ÜberSim.

3.2 Open Dynamics Engine

The physics simulation engine is the key to making ÜberSim useful in terms of high performance robot control. The physics simulation engine must be able to model, with appropriate parameterization:

- Contact collisions between polygonal rigid body objects with different elasticity
- Static and sliding contact between two surfaces and the transition between static contact and sliding
- Rigid body motion with non-negligible mass and rotational inertia

Although there are a number of Open Source simulation engines available, most focus on producing fast pseudo realistic simulations for use in computer games. These engines are therefore fast, but produce motions that look good as opposed to being accurate. In contrast, there exist a number of simulation engines for rigid body motion that are useable for simulating the mechanical interactions of rigid parts. These simulators vary in terms of their integration engines used to produce a forward model of how the rigid parts move dynamically, and in the collision detection/handling mechanisms. We desire an accurate but fast simulation engine for ÜberSim that models rigid bodies, elastic collisions between rigid parts, and contact surfaces that have both static and dynamic friction.

For ÜberSim we chose the Open Dynamics Engine (ODE) (version 0.035 was used for this paper). ODE is an Open Source rigid body simulation engine, developed by Russell Smith, available at [9]. It has been used in a number of other projects (see the site for details), is reasonably documented, and has reached a maturity level ensuring that the code is stable and useable. ODE is essentially a simulation library that provides support for rigid body motion, with finite mass, rotational inertia, and non-even mass distributions (defined via the moment of inertial matrix). ODE provides support for rigid body collisions. Contacting surfaces can take a number of configurations with static friction, or sliding contacts with adjustable levels of Coulomb friction. Finally, ODE provides for a small, but ever growing, number of joint types. Although there are only a few joint types, they cover the range of mechanisms that have been used in any of the leagues at RoboCup. Finally, its simulation engine provides fairly accurate integration using Euler integration, that is fast and more importantly, stable. As a bonus feature, the ODE library includes Open GL routines to render the 3D simulated environment. Although cross platform development is not currently a goal of the ÜberSim project, ODE is a cross platform development making future extensions of ÜberSim to multiple platforms quite possible. Put together, ODE meets the requirements for the simulation engine. Its main limitation stems from its collision detection package, which only provides a small set of primitive object shapes rather than a general polygon soup.

3.2.1 ODE Details

Within ODE, objects consist of rigid bodies, geometries, and joints. Rigid bodies are dynamical objects and therefore have mass (with optional mass distribution), rotational inertia, and momentum. As rigid bodies are dynamical objects their motions are calculated during simulation updates by numerically integrating their equations of motion based on the forces acting on the bodies. In contrast, geometries and joints are not dynamical objects and are used in the integration step. Instead, geometries are used to determine when and where collisions occur and what resultant forces are transferred to the connected rigid bodies. Joints are special geometries used to specify how two connected rigid bodies can move in relation to one another. Joints can also be powered, which provides the primary mechanism for controlling how the connected body behaves.

ODE groups connected rigid bodies into *islands*. Disconnected islands are simulated independently during the motion integration step. In the context of ÜberSim, each robot is a separate island, a connected group of rigid bodies. Rigid bodies interact with one another, and with other geometric objects such as the ground, whenever they make contact. Contact between objects is a collision detection process, discussed in more detail below, based on the geometry data structures. Whenever contact between two parts is made forces are introduced at the contact point, line, or surface. The resulting forces acting on the rigid bodies as a result of the contact is a function of the properties of the contacting surfaces. ODE stores these properties in a surfaces data structure.

ODE supports a range of different types of contact. Contacting surfaces can either be hard or soft, where soft surfaces are essentially deformable meaning some penetration depth is allowed in collisions. In contrast, hard surfaces allow no deformation and therefore no penetration of either rigid body. Contact collisions can have some elasticity, whereby the restitution coefficient can be independently set. Contacting surfaces can impart frictional forces on one another where the friction coefficients for these interactions can be specified. If the friction force exceeds the stiction limit (ie. the force exceeds $\mu|F_N|$ where F_N is the magnitude of the force normal to the contact surface), then the contacting surfaces slip. As with friction coefficients, ODE allows the slip coefficients to be specified. Finally, for contacting surfaces, ODE allows the friction and slip coefficients for two orthogonal directions to be in specified independently. This is useful because omni directional wheels have the unique property of rolling freely in the direction of its axis, while operating as a normal wheel in the direction perpendicular to its axis.

3.3 Building a Simulation Environment

ODE provides a basis for building a realistic, 3D simulation engine for multi-robot problems. However, ODE is only a library for simulation, the main part of the simulator must still be built. Similarly, for ÜberSim to be useful and to meet the goals of its specification, it needs to be structured in such a way that new robots can be easily incorporated into the simulation package. Additionally, the parameters that define a robot configuration need to be easily changeable to support design modifications as well as to operate as a design tool. In this section we describe how ÜberSim is structured around ODE, how the extensible robot classes are defined and parameterized.

3.3.1 Storing the World State via the Scene Graph

At the core of ÜberSim is the scene graph. The scene graph stores all the information about the simulation environment and is the major data structure used by ÜberSim. The scene graph hierarchically stores the geometric and physical information about each object in the environment. The scene graph is a tree consisting of linked nodes. Each node stores a bounding sphere for collision detection where the bounding sphere encloses all the bounding spheres contained in its children. Each node may optionally store an ODE geometry object. The geometry object may also be augmented with a rigid body object, or a joint object, or a surface object for collision contacts. If a node contains a rigid body, joint, or surface, then it must contain a geometry object. Finally, each leaf in the scene graph must contain at least an ODE geometry object. Figure 4 shows an example scene graph for a simple robot with no dribbler or kicker and a ball.

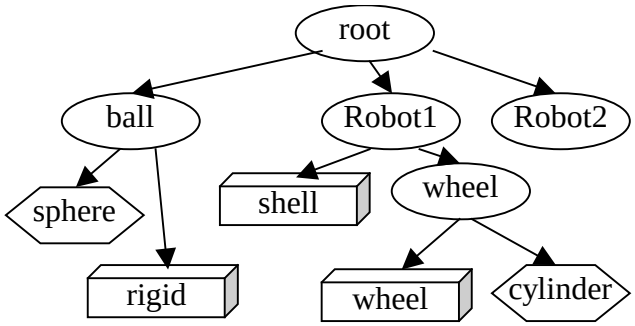

Figure 4. A simple scene graph. Rigid bodies are boxes, nodes are ovals, and hexagons are geometries.

The scene graph is the main data structure used by ÜberSim. It compactly encodes:

- The geometric properties of each object in the environment

- The physical parameters for each rigid body that partakes in the integration step
- The contact surface parameters: friction, bounce etc.
- The geometry information required for optimizing and performing collision detection

The remaining components of ÜberSim are dedicated to creating and maintaining the scene graph, using it to perform operations such as collision detection, using it to drive the actuators in the system, or extracting geometric information from it to report to form the output of the simulator. Finally, the scene graph forms the main object used to store the data required by ODE to execute the simulation and by ODE's graphical visualization engine used to draw the simulated objects via Open GL.

The scene graph is constructed and maintained through base object classes. Currently, there are three types of objects: the field, the ball, and robot classes. Each object type, upon construction, adds nodes to the scene graph. The root node of the scene graph branches to different object types which in turn branch to nodes or leaves with geometry, and possibly rigid body information. Once created, the role of each object class is to provide a useable interface for handling simulator events and extracting geometry information about an object. ÜberSim events include manipulation commands and actuator commands for robots, Manipulation commands are for manually moving objects around in the simulated environment, for example when a user is moving objects via a GUI. Actuator commands are commands sent to drive a robot around the field or to use its other actuators. The final role of the object class is to provide a useful interface for extracting object pose information in order to generate perceptual output from the simulation engine.

3.3.2 Collision Handling

Collision are handled using a Collision Manager (CM) developed for ÜberSim. The CM uses the scene graph data structure and the ODE collision detection components. Essentially, the CM uses the bounding spheres encoding in scene graph to determine which geometries are potentially colliding. Once found, the CM uses the ODE collision detection routines relevant to the colliding objects types, to determine if there is a collision.

To determine what nodes are potentially colliding, the CM uses the bounding spheres encoding in the scene graph. Each node in the scene graph stores a bounding sphere that surrounds all of its child nodes and so on recursively through the tree. The first step of collision handling is to determine which bounding spheres intersect. To test for intersection, the bounding sphere for each node is compared to each other node in the tree that it is not its ancestor or descendent. The ancestor/descendent constraint is required because the node's bounding sphere is contained within each of the ancestor bounding spheres. Likewise, all descendent bounding spheres are contained within the node's bounding sphere. If a node intersects with another node in the tree, other than its ancestors or descendents, then the child nodes are checked recursively until the colliding nodes are found. If a node does not intersect with any other node in the tree, then none of its children intersect with any other node, thus no further collision checking is required for that node or its children. Once all intersecting nodes have been located, and the ODE collision routines for the object types in each intersecting node are called. These routines determine if there is a collision, and using the surface properties (if defined) for the contacting surfaces, ODE specifies what forces result. These forces then act on the colliding parts in the next simulation steps to produce the appropriate response.

The CM search for collisions is fast because of the tree search using the hierarchical bounding spheres. In the types of simulations ÜberSim is used for, the number of colliding objects is limited and often significantly less than the size of the scene graph. Hence the tree-based search provides a powerful tool for quickly finding the few objects that are colliding.

3.3.3 Base Object Class, Field and Ball Objects

The base object class is the *RigidBodyEntity* object. This class encapsulates the notion of a rigid body with geometry. The base class provides the base operations for adding the object to the scene graph, accessing its position, and for handling manipulation commands from ÜberSim. All other objects, the field, ball, and robot classes, are derived from *RigidBodyEntity* and therefore inherit its basic capabilities. The basic structure for objects derived from *RigidBodyEntity* is shown in Table 1.

Table 1. Pseudo code describing a physical object.

```
Class SomeObject
  Derived from RigidBodyEntity
Methods
  Constructor
  HandleEvent(Event e)
Data
  Node root_node
  Other data
Constructor
  Create root node for SomeObject and add to scene graph
  For each part in SomeObject
    Create [Geometry, RigidBody, Joint, Surface, Node]
    Set parameters from configuration file
    Add to scene graph
  End
End
HandleEvent(Event e)
  Switch (e)
    SomeEvent:
      Handle the event
    Default:
      Call default RigidBodyEntity handler
  End
End
```

The field object is a static geometry. As the field is essentially non-moving, it consists of only geometry objects. The geometry objects define the position and shape of the field walls and ground surface. Once created, the field object is unused and its role in simulation occurs via the collision management engine and its defined geometries in the scene graph.

In contrast to the field object, the ball is a moving object but contains no actuators. Thus, the ball object contains a single spherical geometry that describes its surface and a rigid body object to describe the dynamical properties of the ball. The rigid body object contains the parameters that define its mass, mass distribution, and rolling friction. A surface object describes the contact properties of the ball. Specifically, the contact properties are the friction coefficient of the ball on carpet, the elasticity of the ball for collisions, and the coefficient of friction for the carpet

when it is in slipping mode. The latter is required when the ball is driven by the dribbler and spins backwards on the carpet against the robot kicker plate. Once created, the ball object handles velocity manipulation commands from ÜberSim and passes position commands to its ancestor *RigidBodyEntity*.

3.3.4 Robot Objects

One of the core challenges in the ÜberSim concept is the problem of how to enable new robot configurations to be easily added and/or parameters changed easily. In terms of robot hardware and capabilities, RoboCup is a very dynamic domain. Robots are often completely rebuilt from one year to the next. Hardware innovations may result in completely new drive configurations, new complex ball manipulation devices, and at the very least substantial changes in the parameters that describe the physical characteristics of a robot. For the ÜberSim concept to be useful to a team, or to be capable of providing a simulation environment for comparing two teams, there must be mechanisms to add new robot types easily. Moreover, given the argued need for simulation accuracy, there must be a mechanism to easily adapt robot parameters to closely match those of the physical robot.

To achieve this goal, ÜberSim uses multiple robot objects where each object encapsulates a particular robot type. Each robot object is derived from the usual *RigidBodyEntity* object giving it the base abilities for manipulation. Each robot object encapsulates a parameterized robot configuration, where the parameters describe the physical characteristics of a robot. In the current implementation of ÜberSim, two robot objects are defined: *OmniRobot* and *DiffRobot*. Each robot type, with appropriate parameters, can represent nearly all the robots found in small-size RoboCup competitions. The two types are distinguishable based on the drive configuration where the *DiffRobot* uses two wheels, while the *OmniRobot* uses three specialized omni wheels. Each robot has an optional dribbler and kicker mechanism.

The robot types define a generic differential or omni directional robot base with a kicker and dribbler (if selected). The particular parameters that define the dimensions and positioning of each part, the physical properties of the major robot components (mass, inertia etc.) and their contact parameters are read from human readable/editable ASCII text configuration files. Configuration files are ASCII text files that allow comments (preceded by a '#' character) and parameter assignments. Parameter names are ASCII strings (with no spaces and limited special characters) assigned to arbitrary but non-zero length vector of values. An example is:

KICKER_SIZE = 120 3.175 20 # *<x, y, z> mm*

A configuration file reader parses each file. If the parameter is being used, and is stored in the file, the configuration reader extracts the vector information based on the parameter type (real, integer or string), and stores the resulting array in the converted format. The converted array is then accessed directly making future data access very fast. By using the combination of configuration parameter files, new robots within in a given class can be quickly added if its physical parameters are known.

In the CMDragons system, like many other small-size league teams, each robot is commanded to move by specifying robot relative velocity commands. Typically, each robot locally implements velocity control servo loops, using for example PID compensation. As mentioned above, ÜberSim must be able to receive similar commands. Thus, each robot object incorporates event handlers to receive velocity commands, as well as kicker and dribbler binary commands.

All velocity commands $\underline{v} = (v_x, v_y, \omega)^T$ sent to the robot are defined in a robot relative reference frame (see Figure 5). To translate the velocity commands into wheel motor angular velocity for each of the N wheels (ie. to generate $\underline{w}=(\omega_1, \ldots \omega_N)^T$) the velocity command is transformed using the inverse of the forward kinematic transform T. In other words:

$$\underline{w} = T^{-1}\underline{v}$$

The inverse transforms for the different drive robot T^{-1}_{Diff} and omni directional robot T^{-1}_{Omni} for a wheel of radius r and a wheel base of $2R$ (see Figure 5), are given by:

$$T^{-1}_{Diff} = \begin{pmatrix} -\frac{1}{r} & 0 & \frac{R}{r} \\ -\frac{1}{r} & 0 & -\frac{R}{r} \end{pmatrix}, \quad T^{-1}_{Omni} = \begin{pmatrix} -\frac{1}{r\sqrt{3}} & \frac{1}{3} & R \\ 0 & -\frac{2}{3} & R \\ \frac{1}{r\sqrt{3}} & \frac{1}{3} & R \end{pmatrix}$$

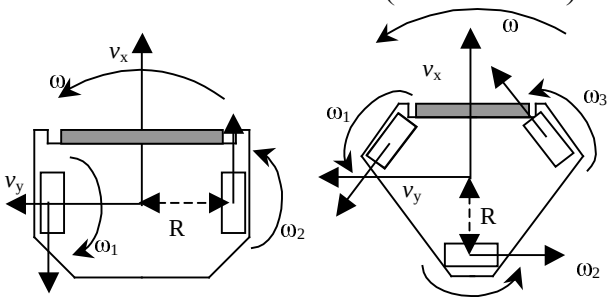

Figure 5. Reference frames.

3.4 Control Interface

Following our approach with our previous 2D simulator, configuration of what robots are on the field utilizes an identical interface to the CMDragons high-level vision configuration. That is, high-level vision must be told which robots are currently on the field so that it uses the appropriate models to recognize them. ÜberSim uses an identical interface to configure what robots, and what type, are on the field.

3.5 Main Simulation Loop

The main simulation loop essentially branches to each part of the simulation process in sequence. Essentially, the main simulation loop checks for any new configuration commands and creates or moves objects as required. If there are any robot velocity commands, they are sent to the appropriate robot object handler. The simulator is then progressed by the simulation step size δT and the process continues. When new perceptual output is ready, the raw position information is extracted from each major object in the scene graph, transformed to the appropriate format, and sent to the client programs. Perceptual output is sent every ΔT seconds, while simulation steps occur at a finer resolution of δT seconds ($\Delta T = k.\delta T$, with integer k) to ensure integrator stability. For the CMDragons system, ΔT is defined to be 33ms corresponding to a 30Hz frame rate. The simulation step size δT was set to 16ms such that $k = 2$ to obtain accurate simulations.

3.6 Graphical Visualization

Finally, ODE is equipped with OpenGL drawing routines. ÜberSim makes use of these drawing routines, if Open GL is available, to provide a rich 3D visualization of the environment. Figure 6 shows an example screenshot from the simulator.

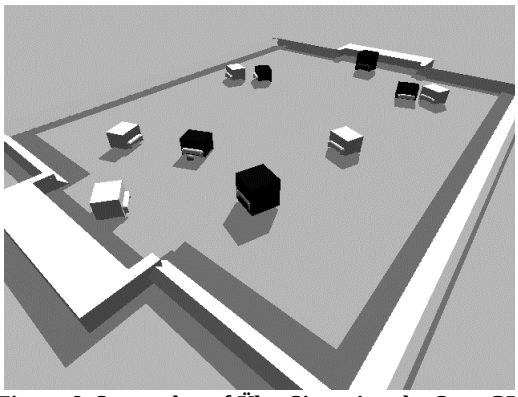

Figure 6. Screenshot of ÜberSim using the OpenGL visualization built using the ODE engine.

4. RESULTS & DISCUSSION

ÜberSim as described in this paper is able to simulate two teams of 5 robots and the golf ball playing soccer. With two simulation steps per frame of vision, running on a 1GHz Pentium processor, the simulator is able to execute at full frame rate (30Hz output, 60Hz simulation steps) using around 60% of the processor for two full teams playing against one another.

To examine the performance of the system, we have compared the trajectories generated by ÜberSim for a robot performing a set pattern and performing acceleration tests. The set patter is a fixed sequence of target points fed into the navigation/motion control system where the target speed at the destination is set to zero. For details on the navigation and motion control see [4]. Figure 7 shows the acceleration tests at 2 $m.s^{-2}$ for a differential drive robot in simulation and reality. The trials were repeated 5 times. Note that there will be some variation in the observed trajectory as the real system has noise and natural variation in the robot parameters. Figure 8 shows trajectory comparisons for a differential drive driving in a figure 8 at around 0.8 $m.s^{-1}$ for a simulated robot versus a real one.

In its current form ÜberSim is functional, but rather "bare bones". There are a number of limitations, and extensions to the architecture that are required before it becomes truly useful as a simulation engine for multiple teams or for comparing the control approaches of two different teams. Currently ÜberSim is limited to the primitive geometrical shapes supported by ODE. However, there are mature Open Source packages for fast collision detection libraries for general polygon shapes. Voronoi Clip (or V-Clip) (see [10] for a review and comparison) is one potential collision detection package that could augment the ODE substantially.

A second area of investigation is improvements to the robot class structure. Currently, new robot classes must be added at compile time. While adding a program class is an easy exercise it is not trivial. Thus future work is required to explore how best to extend, or replace, this approach. A final area of work relates to the robot software interfaces. Ideally, we would like the option of using a high-level vision/action interface or using a low-level perception model, and a low level motor model, to test all parts of the control system.

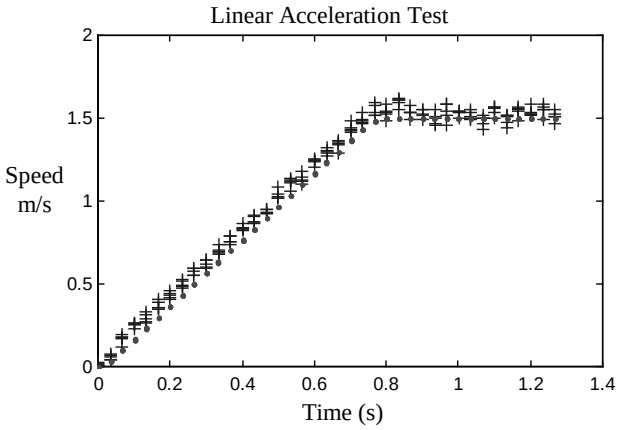

Figure 7. Acceleration comparison for a simulated Diff Robot ('.') versus a real robot ('+') repeated 5 times.

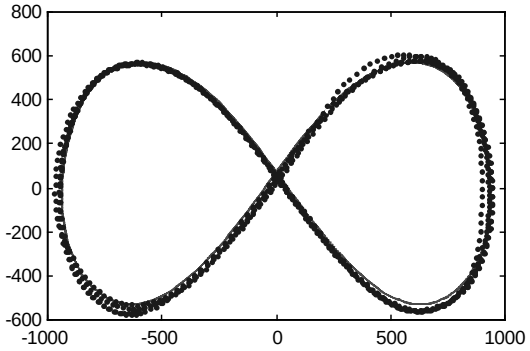

Figure 8. Trajectory comparison for simulator (line) and real robot ('.') performing a set figure 8 shape.

5. RELATED WORK

A plethora of robot simulators have been created and used to support robotics research. Although some are available in the public domain, most are not. Most simulators provide simulation engines for developing general-purpose multi-robot software. Prime examples of this approach are Player/Stage [6][12] and TeamBots [1]. Player/Stage is a distributed multi-robot simulator and can simulate large numbers of different robots interacting in a complex, structured environment using a range of conventional sensors. In terms of dynamics, the simulation package uses low-fidelity dynamics models limited to kinematic approximations. TeamBots is similar to Player/Stage. Written in Java, it provides a range of vehicle types and sensor types. Although TeamBots has been used to simulate games of robot soccer [2], the simulator again uses low-fidelity dynamics approximations. The Robot Soccer Server [8], which is the official simulator for the RoboCup simulator robot soccer league, provides a simulation environment dedicated to investigating the high-level control issues for a team of distributed soccer agents. In contrast to TeamBots and Player/Stage, the Soccer Server simulates each agent as a high-level abstract robot/human. Although the simulator provides dynamical interaction between the agents and the ball, it uses no realistic dynamical models for agent movement. Indeed, there is no collision detection so agents can drive through one another.

For robot soccer where the control code has to transfer to a real system, dynamics are an integral part of the problem. Hence, low-fidelity dynamic approximations limit the usefulness of a simulator for the development of robot control software. Dynamics are critical for motion control, as it is the dynamical limitations of motor torque, inertia, and traction that prevent a robot from instantaneously moving to where it wants to go. Similarly, ball manipulation is one of the primary challenges for robot control of a soccer robot. To develop detailed software control for ball manipulation in simulation, one most certainly needs a high-fidelity simulator to have any hope of developing software that is transferable to the real system.

Most robot soccer teams develop simulation engines to rapid prototype control software, but few publish details of the simulator or make the software publicly available. There are some exceptions, however. M-ROSE [5] is a novel simulation engine that uses a neural network (NN) to learn the forward dynamics model. That is, a conventional back-propagation NN is trained using results recorded from the real robots to approximate the movement model for the robot without latency. The simulation engine operates by receiving commands, delays them by the latency amount, and then using the current state of the robot with the forward model determines where the robot moves to for the next simulation time step. The authors use collision detection algorithms to handle ball collisions with robots and provide no modeling of robot-robot contact. This approach is distinctly different to the approach described in this paper. The advantages of the M-ROSE approach are that measurement of the physical parameters of the world is done implicitly through the data collection to train the forward model. One disadvantage of this approach is that it is unclear how robot-robot collision detection could be integrated with the forward model to produce at least partially realistic robot-robot interactions.

6. CONCLUSIONS

In this paper we have described the development of a new high fidelity simulation engine for robot soccer an adversarial multi-robot task. The simulation engine, ÜberSim, builds on top of an Open Source physics simulation library to provide fast, stable and accurate rigid body dynamics. We have developed the ÜberSim architecture to enable easy addition of new parameterized robot types. Furthermore, we have developed the architecture such that parameters are in ASCII text files for easy editing and manipulation. Using this architecture we have created both a differential and an omni directional robot class and simulated a game of soccer using our existing small-size robot soccer software. Much work still remains to be done to achieve the final vision for ÜberSim, in particular collision detection and more powerful tools to easily add new robot types and parameters.

7. ACKNOWLEDGMENTS

The authors would like to thank Prof. Manuela Veloso and Michael Bowling for the help and support for the development of the research described in this paper.

This research was sponsored by Grants No. DABT63-99-1-0013 and F30602-00-2-0549. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the funding agencies.

8. REFERENCES

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

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

[3] Bruce, J., & Veloso, M. Fast and accurate vision-based pattern detection and identification. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation* (ICRA'03), 2003, under submission.

[4] Bruce, J., Bowling, M., Browning, B., & Veloso, M. Multi-robot team response to a multi-robot opponent team. In *Proceedings of IEEE International Conference on Robotics and Automation* (ICRA'03), 2003, under submission. A previous version also submitted to IROS-2002 workshop on Collaborative Robots

[5] Buck, S., Beetz, M., & Schmitt, T. M-ROSE: A multi robot simulation environment for learning cooperative behavior. In H. Asama, T. Arai, T. Fukuda, and T. Hasegawa (eds.): *Distributed Autonomous Robotic Systems 5*, Springer, 2002.

[6] Gerkey, B., Vaughan, R., Støy, K., Howard, A.,Sukhatme, G., & Mataric, M. Most valuable player: A robot device server for distributed control. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001), pages 1226-1231, Hawaii, October, 2001.

[7] 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, L.N. on A.I., Springer Verlag, 1998, 1-19.

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

[9] Open Dynamics Engine main site: *http://www.q12.org/ode*

[10] Reggiani, M., Mazzoli, M., Caselli, S. An experimental evaluation of collision detection packages for robot motion planning. In *Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems*, (IROS'2002), Switzerland, October, 2002.

[11] ÜberSim and CMDragons'02 software downloads page *http://www.cs.cmu.edu/~coral/download/*

[12] Vaughan, R. Stage: a multiple robot simulator. *Technical Report IRIS-00-394*, Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, 2000.