Carnegie Mellon University
Research Showcase @ CMU

Robotics Institute | School of Computer Science

2004

Accurate and Flxible Simulation for Dynamic, Vision-Centric Robots

Jared Go
Carnegie Mellon University

Brett Browning
Carnegie Mellon University, brettb@cs.cmu.edu

Manuela M. Veloso
Carnegie Mellon University, veloso@cs.cmu.edu

Follow this and additional works at: http://repository.cmu.edu/robotics

Part of the Robotics Commons

Published In

.

This Technical Report is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Robotics Institute by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Accurate and Flexible Simulation for Dynamic, Vision-Centric Robots

Jared Go Brett Browning Manuela Veloso

November 19, 2004
CMU-CS-04-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213

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: Robotics, Simulation, High-Fidelity, Dynamics, Vision-Based, Latency.

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 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.

*Authors can be contacted at jgo@cmu.edu, {brettb,mmv}@cs.cmu.edu

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 8 km/h (3.5 m/s), a latency of 100ms in vision systems (common latency for systems without dedicated hardware) can induce a positional error of more than a third of 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.

3 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). 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.

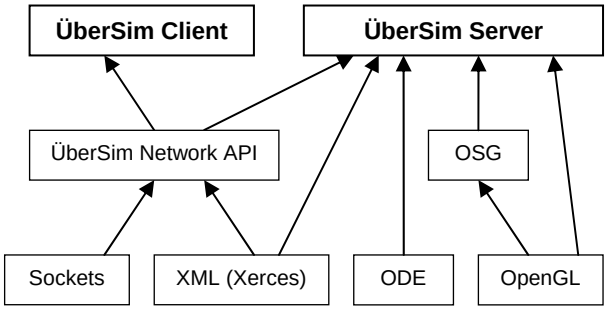

Figure 2. Overview of components in the ÜberSim framework.

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 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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.

4 Ubersim Server Code Architecture

The Ubersim Server was written in C++ using Microsoft Visual Studio 2003. The core solution file is named Ubersim2.sln and contains a single C++ project stored as Ubersim2.vcproj. While a Linux port has not yet been performed, the process of porting should be easy as the OSG and ODE libraries are both multiplatform and the Ubersim code only uses windows-specific code to create the display window. In this section, we describe in detail the structure of the code and how the different components interact to create the working simulator.

4.1 Sim Class

The core simulation class is the Sim class, defined in the files Sim.h and Sim.cpp. This class is responsible for handling all client connections, storing and ticking the simulated environment, and rendering the output to the screen. Usually, execution requires only one instantiated Sim object during the lifetime of the simulator.

When a Sim object is first instantiated, it is passed the timestep to use for the simulation as well as the port to listen on for client connections. The timestep and port cannot be changed at runtime. During instantiation, the Sim object creates a TCP socket, sets it to non-blocking, and listens for incoming connections on the specified port. It also initializes an STL list of client objects which is added to as clients connect. The material table is also initialized with appropriate parameters for The simulation then initialized ODE by creating a world and collision space and sets appropriate parameters such as gravity. Note that it may be desirable for these parameters to be stored in a configuration file instead. Finally, the OSG scene representation is constructed by creating a MatrixTransformation object as the root of the scene and placing a single empty Geode object into this scene.

At runtime, the surrounding application simply calls the *Draw()* and *Step()* functions in order to advance the simulation. Internally, network connections are updated and handled when the world is stepped.

4.2 Client Class

The Sim class maintains a list of Client objects which represent the connected clients. Each Client object contains various buffers for communication with the client, as well as a list of the simulation objects which are under the client's control. The Client class is defined in the files Client.h and Client.cpp.

When an incoming connection is detected by the Sim class, a Client object is created in the Initializing state and is added to the Sim object's client list. While in the Initializing state, the connected client has no physical manifestation in the simulation and is simply expected to transmit its self-instantiation XML document to the server. At every tick, the Sim object calls the *UpdateRemoteClient()* function on every connected client. Within each Client object, the socket is read from and the incoming data is stored into a buffer until the self-instantiation XML termination is found. In the current implementation, the terminator is two consecutive percent signs "%%".

Once the terminator has been read, the document is converted from a char array into a STL string, and is read using the Xerces XML library by using a MemBufInputSource object which points to the stored string in memory. The XML nodes are parsed and analyzed, and RigidBody objects are created which represent the parts of the robot which have a physical manifestation and must be handled in collision checking and rigid body physics simulation. Other robot components such as cameras or motors have no physical manifestation and are thus simply stored in a list of Component objects. After creating the objects and components, the appropriate joints (as described by the self-instantiation XML) are created in ODE to bind the different rigid body pieces together. Finally, all the created objects are added to the simulation.

Each Client object also contains a NetMessageSocketIO object which is used to handle the sending of sensor message and the receiving of actuator messages from the client. This class handles message parsing and also can be configured to simulate various latencies in sensing or actuating.

4.3 RigidBody and Component Classes

The RigidBody and Component classes form the backbone of the simulation, and are tightly integrated with both ODE and OSG. These classes are defined in RigidBody.h, RigidBody.cpp, Component.h, and Component.cpp. Essentially, an instantiated robot in the simulation consists of several rigid bodies, connected by joints. Each rigid body consists of one or more components, which define the physical extents of the rigid body and the accessories attached to the body. The RigidBody class should not be subclassed by users as it is designed to be standalone and operate with any configuration of components. The Component class, on the other hand, should be subclassed for every different type of component that the user wishes to use on the simulated robot. Examples of components in the existing implementation include solids such as boxes, spheres, and cylinders, as well as sensors such as camera and inclinometers.

The class Component derives from the osg::Group class, which means that it can be attached directly into an OSG scene graph and can contain one or more children. Each component can be marked with a name in the self-instantiation XML, and this name is stored as an STL string inside the Component class. The Component class also contains pointers to the owning client as well as a cached world transformation matrix which is stores the current transform for the component in the simulated world. This transformation matrix is recomputed at each frame during the simulator tick, and is cached here since components such as sensors use the matrix when generating their appropriate sensory data. The most important part of the Component class is three virtual functions which determine the behavior of the component with respect to instantiation, simulation, and actuation.

The first of these virtual functions is the *ParseAdditionalPropertiesFromNode(xercesc::DOMElement *node)* function. This function is called during the parsing of the self-instantiation XML after the properties common to all components have been parsed. This function should be overridden for all components that need to read additional parameters from the XML; for example, a Box primitive may read width, height, and depth fields out of the XML node. A camera sensor may read frame rate and field of view parameters out of its XML node. In general, almost every component will need to override this call to read in additional information out of the XML.

The second function is the *Tick(float dTime)* function. This function is called on every simulation tick and is passed the specified simulation timestep. Components which need to update themselves at each step of the simulation should override this function. Note that no code relating to rigid body simulation, etc. should be placed here as this is already handled at a higher level inside the simulation. An example of a component that would override this function might be a sensor that needed to fire sensory messages to the client at a specified interval.

The third and final virtual function in the Component class is the *ProcessCommand(char* bytes, int len)* function. This function is called when the client sends an actuation message to this component. The binary data can take on any user-

defined meaning, and often consists of a message ID and some associated parameters. This interface allows complete flexibility in the actuator interface for different components. Some examples of actuation message might be setting desired velocity for motors, or changing parameters on sensors.

The RigidBody class combines several components into a single simulated rigid body. Each RigidBody object corresponds to a single dBody object in ODE; the class contains a pointer to a dBody object used for the rigid body simulation as well as a pointer to a dSpace object which contains the collision hash-space for ODE's collision detection routines. During client initialization, the Component objects which compose each rigid body are parsed and are added to a single osg::Group scenegraph node. These components are then visited using custom OSG node visitor objects, which traverse the subgraph using the visitor pattern.

The two node visitors used are called GeomVisitor and MassVisitor and are defined in NodeVisitors.h and NodeVistors.cpp. All components that are solid and have a physical presence which affects the rigid body simulation are descendants of type UB_Solid and have associated rigid body parameters such as inertia tensor matrix and orientation. The MassVisitor finds all of these solid components within the rigid body and composites the mass properties of each component to determine the total composite mass properties of the rigid body. The center of mass of the components is also computed by the MassVisitor object and is used to offset the components to place them with respect to the rigid body's center of mass. The GeomVisitor object simply traverses the subgraph and collects the ODE collision geometries of type dGeom that are associated with each component. These collision geometries are inserted into the ODE dSpace collision space of the rigid body, and are also offset appropriately to account for the center of mass of the composite rigid body.

4.4 Material Class

The Material class is designed for users to be able to control the interaction between different bodies in the simulation intuitively and at a high-level. Each solid component descending from the type UB_Solid can be marked with a Material parameter in the self-instantiation XML. This material is simply a string referring to the type of material, such as "Rubber" or "Glass". The Material type maintains a static user-defined internal matrix that contains a pointer to MaterialInterface object for combinations of two contacting materials. Currently, this MaterialInterface class contains only the coefficient of friction, but it has been designed to be highly extensible. During simulation, for every collision between two rigid bodies, the material types of the two bodies is determined, and the Material matrix is used to lookup the properties that should be used for the ODE contact joint connecting the two bodies. The matrix is currently set up using hardcoded values in a static initializer, but can easily be modified to read in the material data from a server configuration XML file.

4.5 NetMessage API

The NetMessage API used to pass sensory and actuation message between the client and server is designed to be simple and lightweight. Any component that wants to send a message to the client should create a NetMessage object, which simply consists of a char* pointing to a string with the component's name, as well as a pointer to a char buffer of binary data to send and an integer which marks the size of the binary data to send. This message can be sent using the Client object's *SendMessageToClient(NetMessage* msg)* function, which internally uses the NetMessageSocketIO object inside each Client to send out the message through the client's socket.

Every message sent over the socket connection from client to server or vice versa has the following format: A 32-bit unsigned integer representing the total length of the message, followed by a null terminated string indicating the name of the component that the message originates from or should be sent to, followed by the binary data to send to that component. Note that actuation messages can only be sent by the client after the client has finished the instantiation phase.

4.6 Current Ubersim Components

There are three main classes of components implemented in Ubersim. These include solid components, sensors, and joints. The following sections describe the components in each category that have already been implemented within Ubersim.

4.6.1 Solid Components

Currently, Ubersim features a range of solid components (subclasses of UB_Solid) which can be used to define the physical structure of the robot. These components are UB_Box, UB_Cylinder, and UB_Sphere, and map directly to primitives in ODE and OSG. Upon creation of these nodes, corresponding ODE geoms are constructed using the dCreateBox, dCreateCylinder, and dCreateSphere functions. The renderable OSG drawables are added to an osg::Geode node which is attached to the solid component. The addition of other primitives is contingent on added support from the ODE and OSG libraries. Currently, supported primitives which have not yet been added to Ubersim include triangle meshes and capped cylinders.

4.6.2 Sensor Components

The sensors implemented in Ubersim inherit from the UB_Sensor class which provides them with the basic framework for firing sensory messages back to the client at specified intervals. To make use of this framework, subclasses should override the virtual function SendSensorUpdate(). When the Tick(float dTime) function is called by the simulation, the component keeps track of the elapsed time and call the SendSensorUpdate() function after a specified interval has elapsed. The rate at which the sensor fires can be modified by changing the floating point member "rate" defined in all sensors. This value can either by hardcoded to different values or can be read from the self-instantiation XML.

Two different types of sensors are currently implemented in Ubersim. The first of these is the UB_Inclinometer, defined in the .h and .cpp files of the same name. This sensor sends back sensory messages in the form of a single 32-bit floating point number which represents the degree of inclination of the sensor from vertical, in radians. The second sensor is the UB_CameraSensor, which sends back sensory messages in the form of uncompressed pixel data. By default, the camera is set to generate images at 20Hz with a 45 degree field of view in QVGA resolution (320x240). The actual generation of the image involves using an OpenGL extension called P-Buffers, which allows rendering of the scene to be done to an offscreen surface which can then be read by the CPU and sent to clients. The exact process of using P-Buffers in combination with OSG is described in section 4.1.9.

4.6.3 Joint Components

Joint components in Ubersim are handled differently from other components in that they are not actually placed into the scenegraph. Instead, joint components are simply stored in a list inside each Client object. This separation comes from the fact that joints and motors have no physical manifestation in ODE and simply represent constraints between pairs of rigid bodies. Most of the joints in Ubersim have been directly translated from ODE and are named as follows: UB_JointBallSocket, UB_JointHinge, UB_JointHinge2, and UB_JointSlider.

The UB_JointBallSocket component simulates a ball and socket hinge between two components. The only parameter read from the XML is the anchor position, which describes the point relative two the two bodies which should be kept at the same position in both body frames. Note that the relative position of this anchor point to both bodies is defined by the initial configuration of rigid bodies as described in the XML. This type of joint does not respond to actuation messages.

The UB_JointHinge component simulates a hinge joint between two components. For this joint, both anchor position and hinge axis are read from the XML. This joint also does not currently respond to actuation messages.

The final two types of joints, UB_JointHinge2 and UB_JointSlider, can both be actuated with messages to achieve the effect of a motor between two bodies. The hinge-2 joint simulates a car wheel-like connection where two hinges are connected in series. The first hinge acts like the steering axis and has finite stops on rotation, while the second hinge acts like the wheel axis and has no stops on rotation in either the positive or negative direction. The slider joint, on the other hand, simply constrains the motion of two bodies along some axis. The actuation interface for both these joints mirrors ODE's joint parameter setting interface. An actuation message consists of a 32-bit integer describing the joint property to be modified as well as a 32-bit floating point value which indicates the new value for the parameter. Allowed values for the property identifier are described in the ODE document, but some of the most useful properties are desired velocity for the joint (linear or angular), the maximum force exerted by the joint, and the high and low stops for the joint.

One issue of note is that high rotational speeds decreases the stability of bodies connected by a hinge-2 joint. For example, when the hinge-2 joint is used to simulate motors driving the wheels of a vehicle, it is common to see the wheels

wobble and become unaligned with the wheel axis. This issue appeared during testing of the Segway simulation and was corrected by using an additional UB_JointBallSocket to connect the wheels to the main body. Another possible solution to this problem would be investigating ODE's finite rotational mode flag which is recommended for use with bodies that have a high rotational velocity.

5 Simulation Execution Details

In this section, we describe how the Ubersim code executes at runtime and explain some of the interesting parts of the simulation. The core main routine is found in the Ubersim2.cpp file and begins execution simply by registering the application class and creating a window, as is standard in Windows applications. The next steps involve using OpenGL functions to create an alternate rendering context and loading OpenGL extensions using the GLEW library. The extensions are necessary for creating and using the P-Buffers needed for vision synthesis.

Afterwards, the core Sim object is created and the main simulation loop is entered. Within the loop, the simulation constantly steps and redraws the simulation. Within this loop, there are several key components of the simulation algorithm that are worth further explanation. These components are: P-buffer rendering, ODE and OSG synchronization, and the network timestamp and synchronization algorithm.

5.1 P-Buffer Drawing Code

The core P-Buffer management code can be found in the files PBuffer.h and PBuffer.cpp. These files define the class PBuffer which wraps up the creation and usage of PBuffers. Essentially, the class queries for a suitable pixel format for the PBuffer surface and uses the wglCreatePBufferARB() extension to obtain a PBuffer and rendering context. In the PBuffer class's Activate() function, the PBuffer's rendering context and device context are selected into OpenGL which causes any further drawing to be done into the new surface.

Inside of the UB_CameraSensor's SendSensorUpdate override, the PBuffer is activated and an OSG scene view is set up to render the scene from the viewpoint of the camera in the simulation. This scene view is also set to use an alternate render context, which is needed since the switch of rendering contexts necessitates OSG flushing some of its internal state. After the scene is drawn, the glReadPixels function is used to read the framebuffer back into the system memory, and the pixels are sent back to the client untouched in the form of uncompressed pixel data. At this point, distortions or other effects such as noise, etc. could be added into the image before the data is sent to the client.

5.2 ODE and OSG Synchronization

Since the ODE library provides the rigid body simulation for the environment while OSG provides the rendering and scenegraph, a critical component in Ubersim is the algorithm which maintains synchronization between the two libraries. This synchronization is also essential for doing accurate collision detection within the simulation. This synchronization takes place at the end of the Sim::Step() function and involves several steps.

The first step in synchronization involves calling the SyncDisplay() function for all RigidBody objects currently in the simulation. This function reads the ODE-computed position and orientation of every rigid body and sets these values as the position and orientation of the OSG subgraph belonging to each rigid body. After this step is complete, the OSG scenegraph accurately reflects the position and orientation of all objects in the ODE simulation.

The second step involves calling the UpdateGeomTransforms() function for all RigidBody objects in the simulation. Since the OSG subgraph has been updated, each node in the subgraph corresponding to different components of the rigid body now has the updated transformation matrix. This matrix is thus used to set the transformation on the corresponding collision-space geom objects in ODE.

Afterwards, the ODE collision detection function is called which detects any collisions in the simulation. The result of this operation is the creation of ODE contact joints representing all of the collisions at the current frame of the simulation. Immediately afterwards, the ODE simulation is stepped using the dWorldStep() function to advance the world to the next frame. The temporary contact joints are then removed from the simulation, and the simulation time is updated. This core

sequence of events is executed at every time the world is stepped and ensures that the ODE and OSG components of the simulation are kept in strict synchronization.

5.3 Timestamp and Synchronization in Ubersim Network API

Ubersim's network synchronization protocol essentially uses a frame-locked architecture to synchronize clients with the server. At every simulation step, each client is sent a sequence of messages, beginning with a timestamp which indicates the current frame being simulated. Immediately afterwards, zero or more sensory messages are sent depending on the number of sensor components on the client's robot that have fired during the current simulation frame. Finally, a frame end message is sent to each client to signal that all of the sensory data for the current frame has been sent.

After sending this data to all connected clients, the simulation enters a wait phase where it continually reads actuation messages from each client. It does so until the client sends a SimEnd message which marks that the client has sent all actuation messages for the current frame. Note that there may be zero or more actuation messages each frame. If a client does not send the SimEnd message, the simulation will continue to wait without a timeout. This dependency on each client is an obstacle barring the deployment of a public Ubersim server. Without a timeout or additional modification, a single improperly written client or maliciously written client can cause the simulation to halt.

Although the synchronization algorithm is simple, the frame-locking algorithm is guaranteed to produce complete synchronization and a deterministic result. Currently, we assume that Ubersim will be used in an internal environment where both the server and clients are under the control of a single owning entity.

6 Ubersim Test Client Architecture

The current Ubersim test client is written in C# and is contained in the TestControl solution file. This project contains code that is suitable for testing purposes only. Essentially, the test client code connects to a specified server, sends the self-instantiation XML, and enters a message processing loop where messages the server are parsed and handled. The test program simulates a Segway RMP robot using a simple PD-controller to maintain balance. The complete self-instantiation XML used to describe the virtual Segway RMP is given in Appendix A. Essentially, the virtual Segway is composed of several solid components which approximate the weight distribution of the real Segway RMP. In addition, the virtual Segway has a single camera, mounted underneath the top plate as is present on the CMU modified Segway RMP. The inclinometer primitive is also attached to the base of the virtual segway, instead of the embedded inclinometer which exists in the real Segway RMP.

At runtime, messages from the server are handled in the following manner. Messages from the camera sensor are simply dumped to a GDI+ surface and are displayed in an area on the application. Inclinometer messages are parsed, and angular orientation and angular velocity where a simple first-order approximation is used to determine velocity as the derivative of orientation. Once the server has sent the sensory data complete message for a particular frame, the client immediately responds with actuation messages that set the desired angular velocity for the Segway's wheels. This first involves computing the desired angular velocity for the wheels using the PD controller. The PD controller takes as input the angular tilt, tilt velocity, linear position, and linear velocity of the robot. The resulting desired wheel velocity is packed into two actuation NetMessages for the two wheel motors on the simulated Segway RMP. These two messages are then sent to the server and are followed by the client frame end message which indicates that all actuation messages have been set. This message processing loop runs until the client program is terminated.

7 Experimentation

This section outlines how experiments were carried out as well as how other data was captured from the simulation at runtime.

7.1 Video Capture

One of the most desired functions in a simulator is the ability to capture visual output for use in debugging or for demonstration purposes. To this end, an Nvidia library called MovieMaker was explored and an initial implementation was tested. This library essentially allows for extremely fast capture of screen-rendered OpenGL to any compressed file format supported by windows. Using such a library is often more desirable than using an external third-party solution such as FRAPS as video capture can be triggered from within code and additional control is gained over when the library captures images from the simulation. The downside of the Nvidia library, however, is that the fast capture algorithm implemented makes use of the screen device context as the source of a bit block transfer operation, which means that any windows that block the Ubersim simulator window will be captured into the video.

For the videos that have been captured during Ubersim development, the FRAPS tool was used manually to capture output from the OpenGL window.

8 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.

9 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.

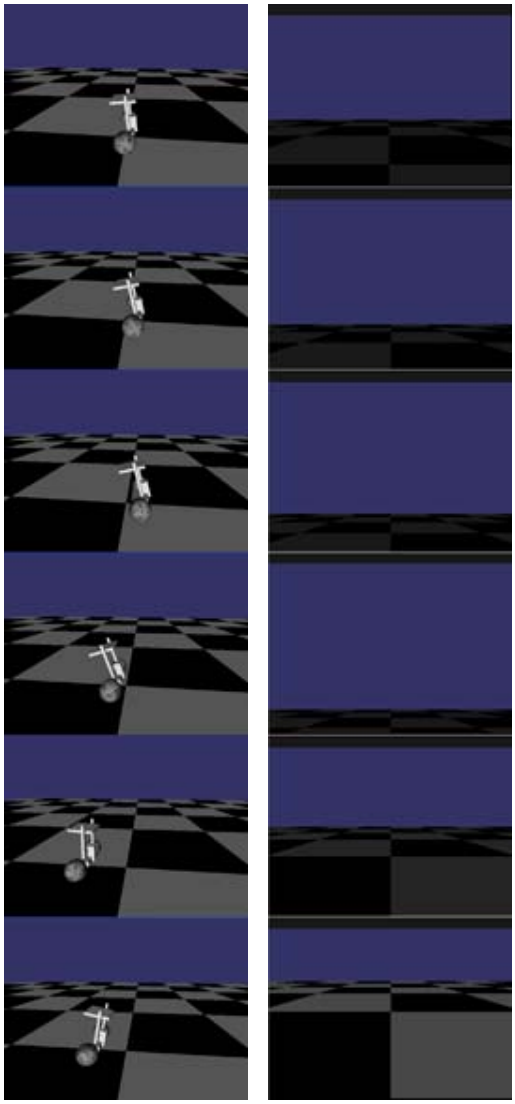

Figure 5. Simulation view of Segway RMP (left) and client-side onboard camera view (right) during balancing.

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.

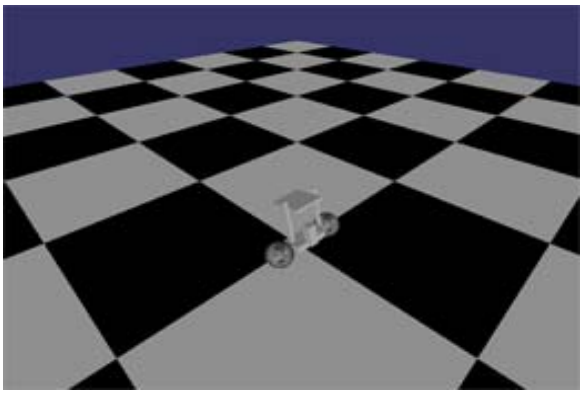

Figure 6. Another view of the simulated Segway RMP.

10 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™ 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.

11 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*

Appendix A. Self-Instantiation XML used for Segway test client.

```
<Client>

<CSettings>
 <Name>CMU</Name>
 <LatencyMS>20.0</LatencyMS>

 <Orientation>
       <angle>-0.5</angle>
       <axis>
                 <x>0.0</x>
                 <y>0.0</y>
                 <z>1.0</z>
       </axis>
 </Orientation>

 <Offset>
       <x>0.0</x>
       <y>0.0</y>
       <z>0.0</z>
 </Offset>

</CSettings>

<Entity>

 <RigidBodies>

       <Body name="MainBody">

                 <Box name="Platform">
                           <Mass>35.0</Mass>
                           <XSize>0.46</XSize>
                           <YSize>0.387</YSize>
                           <ZSize>0.095</ZSize>
                           <Position>
                                     <x>0.0</x>
                                     <y>0.0</y>
                                     <z>0.0</z>
                           </Position>
                           <Material>Plastic</Material>
                 </Box>

                 <Box name="Handle">
                           <Mass>7.5</Mass>
                           <XSize>0.019</XSize>
                           <YSize>0.210</YSize>
                           <ZSize>0.55</ZSize>
                           <Position>
                                     <x>0.220</x>
                                     <y>0.00</y>
                                     <z>0.32</z>
                           </Position>
                           <Material>Plastic</Material>
                 </Box>


                 <Box name="Handle2">
                           <Mass>7.5</Mass>
                           <XSize>0.019</XSize>
                           <YSize>0.210</YSize>
                           <ZSize>0.55</ZSize>
                           <Position>
                                     <x>-0.220</x>
                                     <y>0.00</y>
                                     <z>0.32</z>
                           </Position>
                           <Material>Plastic</Material>
                 </Box>

                 <Box name="Table">
                           <Mass>22.7</Mass>
                           <XSize>0.610</XSize>
                           <YSize>0.508</YSize>
                           <ZSize>0.016</ZSize>
                           <Position>
                                     <x>0.0</x>
                                     <y>0.1</y>
                                     <z>0.597</z>
                           </Position>
                           <Material>Plastic</Material>
                 </Box>

                 <Inclinometer name="Incmeter">
                 </Inclinometer>

                 <CameraSensor name="Cam">
                           <Fov>60.0</Fov>
                           <Position>
                                     <x>0.0</x>
                                     <y>0.16</y>
                                     <z>0.47</z>
                           </Position>
                           <Orientation>
                                     <angle>-0.436</angle>
                                     <axis>
                                               <x>1.0</x>
                                               <y>0.0</y>
```

```
                                        <z>0.0</z>
                                </axis>
                        </Orientation>
                </CameraSensor>

        </Body>


        <Body name="LWheel">
                <Cylinder name="B1">
                        <Mass>5</Mass>
                        <Radius>0.236</Radius>
                        <Length>0.089</Length>
                        <Position>
                                <x>-0.285</x>
                                <y>0.0</y>
                                <z>0.074</z>
                        </Position>
                        <Orientation>
                                <angle>1.5708f</angle>
                                <axis>
                                        <x>0.0</x>
                                        <y>0.0</y>
                                        <z>1.0</z>
                                </axis>
                        </Orientation>
                        <Material>Rubber</Material>
                </Cylinder>
        </Body>

        <Body name="RWheel">
                <Cylinder name="B2">
                        <Mass>5</Mass>
                        <Radius>0.236</Radius>
                        <Length>0.089</Length>
                        <Position>
                                <x>0.285</x>
                                <y>0.0</y>
                                <z>0.074</z>
                        </Position>
                        <Orientation>
                                <angle>1.5708f</angle>
                                <axis>
                                        <x>0.0</x>
                                        <y>0.0</y>
                                        <z>1.0</z>
                                </axis>
                        </Orientation>
                        <Material>Rubber</Material>
                </Cylinder>
        </Body>


</RigidBodies>


<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>0.074</z>
                </AnchorPos>
        </JointHinge2>

        <JointBallSocket name="LConstraint">
                <Body1>MainBody</Body1>
                <Body2>LWheel</Body2>
                <AnchorPos>
                        <x>0.0</x>
                        <y>0.0</y>
                        <z>0.074</z>
                </AnchorPos>
        </JointBallSocket>

        <JointHinge2 name="RMotor">
                <Body1>MainBody</Body1>
                <Body2>RWheel</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>0.074</z>
                </AnchorPos>
        </JointHinge2>

        <JointBallSocket name="RConstraint">
                <Body1>MainBody</Body1>
                <Body2>RWheel</Body2>
                <AnchorPos>
                        <x>0.0</x>
                        <y>0.0</y>
                        <z>0.074</z>
                </AnchorPos>
        </JointBallSocket>
 </Joints>


</Entity>

</Client>
```