SIMULATION FOR AUTONOMOUS SYSTEMS

By

MILIND SHASTRI

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2008

To my parents, who have always encouraged my interests and given unconditional love. To my brother, who has been a constant support in all my endeavors.

ACKNOWLEDGMENTS

I thank my committee members, Dr. Carl Crane, Dr. Scott Banks and Dr. Douglas Dankel, for providing me with invaluable guidance in my project. I thank the PhD and MS students at the Center for Intelligent Machines and Robotics, Steve Velat, Jean-Francois Kamath, Nicholas Johnson, Shannon Ridgeway, Jaesang Lee, Ji Hyun Yoon, Mandar Harshe, Vishesh Vikas and Eric Thorn and students at the Machine Intelligence Laboratory for being a part of all the technical discussions that helped give direction to my research. Robotics forums on the MSDN website were quick to respond to queries and helpful to solve all doubts posted. I thank my friends in being supportive of my research work.

TABLE OF CONTENTS

page

LIST OF FIGURES

LIST OF OBJECTS

LIST OF ABBREVIATIONS

CIMAR	Center for intelligent machines and robotics
CRR	Concurrency and coordination runtime
DARPA	Defense advanced research projects agency
DSS	Decentralized software services
GPOS	Global positioning component
GPS	Global positioning system
HLP	High level planner
KVM	Keyboard, video, mouse
LADAR	Laser detection and ranging
JAUS	Joint architecture for unmanned systems
Jscript	Java script
MRDS	Microsoft robotics developers studio
MSDN	Microsoft developer network
NFL	National football league
NIST	National institute of standards and technology
OOP	Object oriented programming
USAR	Urban search and rescue
VPL	Visual programming language
VSE	Visual simulation environment
XNA	XNA's not acronymed

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

SIMULATION FOR AUTONOMOUS SYSTEMS

By

Milind Shastri

December 2008

Chair: Carl D. Crane III
Major: Mechanical Engineering

An autonomous vehicle in a typical urban environment must maneuver in a city environment, merge into moving traffic, navigate traffic circles, negotiate busy intersections, and avoid obstacles. To test each of these behaviors, it is beneficial if the autonomous vehicle sensing and control algorithms can be tested in a simulated environment similar to the actual urban environment. In such a scenario, computer simulations help reduce testing costs, increase the ease and frequency of tests, put test subjects through extreme conditions without fear of harming other objects and enable quick changes in the system.

We developed simulation models of a physical urban vehicle with LADAR and camera sensors and a typical urban environment, which is built on the Microsoft Robotics Developer Studio framework. The models were developed as a simulation of the actual autonomous vehicle of team Gator Nation and a part of the environment for the DARPA Urban Challenge 2007.

CHAPTER 1
REAL SYSTEM VERSUS SIMULATION

Introduction

In DARPA's vision, "The Urban Challenge features autonomous ground vehicles maneuvering in a mock city environment, executing simulated military supply missions while merging into moving traffic, navigating traffic circles, negotiating busy intersections, and avoiding obstacles." Moving the challenge into an urban setting adds structure and complexity to the Grand Challenge problem. Previous success relied on a single mode of operation, without interaction with the environment beyond simple traversal. Success in the Urban Challenge required numerous modes of operation and complex interaction with the environment [2]. It was expected that the urban environment would also hamper the use of GPS for localization, further complicating the challenge. The specific problem to be solved is detailed in the Urban Challenge Technical Evaluation Criteria document [3]. The problem is organized into four categories: Basic Navigation, Basic Traffic, Advanced Navigation, and Advanced Traffic; each more complex than the previous.

Problems with Testing in a Real Environment

The autonomous vehicle is essentially a robot, as in it has no human operator to drive it [2]. The vehicle is a Toyota Highlander SUV. The vehicle has a wheel base of 2.715 m, a width of 1.825m and a height of 1.735 m. Vehicle weight as delivered is 1850 kg (4070 lbs). Toyota equips the vehicle with electronic power steering. The nature of a full hybrid requires that the throttle and brake be drive-by-wire. Automation of these controls is effected via an interface with the brake and throttle sensors to allow computer input of control signals to the vehicle control architecture. A servo-motor has been installed on the steering column to replace the human input to allow drive-by-wire behavior of the steering subsystem. The robot also consists of six SICK

LMS-291 LADARs, two SICK LD-LRS1000 long range LADARs, six Matrix Vision BlueFox high-speed USB2.0 color cameras, and an additional BlueFox camera configured to see in the near IR band. The sensors provide the robot with an awareness of the nearby and far off environment and aid the robot's on board intelligent systems to make a decision on how to drive the robot around.

To run the robot all its constituent systems have to be tested to work together. Testing the system physically means putting all equipment along with all participants at risk in the event of a malfunction or any improper development of any system.

Why the Simulation Solves the Problems

The autonomous vehicle for the challenge can be imagined as a system comprising physical elements (like the vehicle itself, its sensors and etcetera) and the software that interacts with all the physical elements to produce an intelligent behavior. The intention of the simulation is to provide a simulated platform to the intelligent system running on the Navigator with a realistic physical model of the robot and its environment so that the correctness of the system can be quickly tested without putting the actual robot in danger of getting damaged when tested through extreme conditions. Virtualization of the sensory and actuator components are done in Microsoft Robotics Developers Studio (MRDS) and the control system wherein autonomous algorithms are implemented are unaltered (Figure 1-1).

In essence the complete workings of the car can be reduced to the control of a model car in a game like environment and the testing of "extreme" algorithms without the risk of damaging the real car or the environment. Once finalized, the algorithm can very easily be attached to the real car instead of the simulated car as both systems have similar physical features and take the same inputs to produce similar outputs.

The simulation provided a 3D model of the robot car with similarly physical features of

- Weight
- Size
- Essential mechanical dimensions
- Ackerman steering
- Four wheel independent suspension and damper
- Lateral and longitudinal friction on wheels
- Positions and orientations of sensors

The model is developed in a Microsoft Robotics Studio environment and is essentially programmed in C#. The model is programmed by following standard OOP conventions and MRDS set rules. The architecture is also developed in a manner to be able to interface with the JAUS software running on the Navigator.

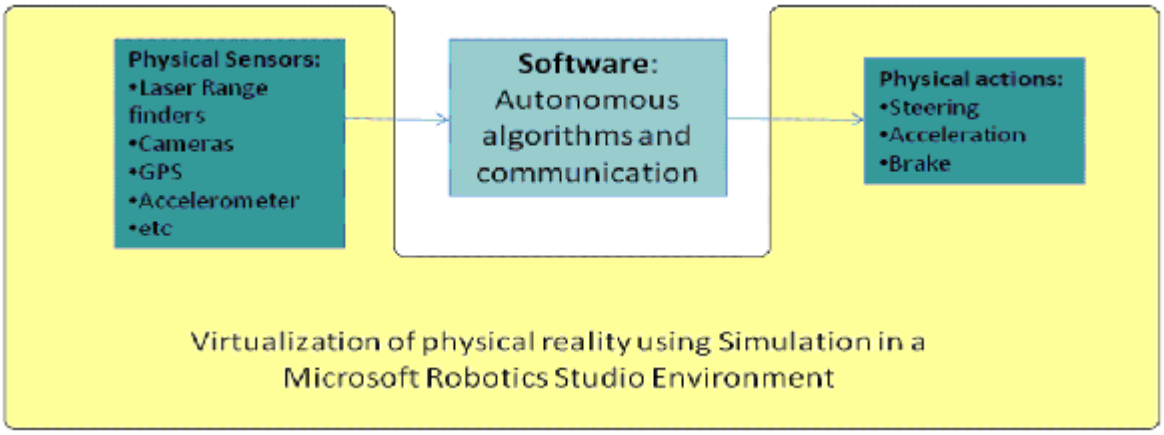

Figure 1-1. Components of project being simulated

CHAPTER 2
LITERATURE SURVEY

Simulation is the imitation of some real thing, state of affairs, or process. The act of simulating something generally entails representing certain key characteristics or behaviors of a selected physical or abstract system [25]. The behaviours can be modeled using mathematical descriptions of actions, events, processes which constitute behaviour.

Simulation is used in many contexts, including the modeling of natural systems or man made systems in order to gain insight into their functioning. Other contexts include simulation of technology for performance optimization, safety engineering, testing, training and education. For example, A simulation has been developed at the Center for Simulation, Safety, Advanced Learning and Technology, University of Florida, (Figure 2-1) that represents what happens when dry cooled air is forced in channels placed between the body and the football pad depending on adjustable variables like air temperature, forced air flow, relative humidity and sweat production [23].

The important issues to consider in a simulation are a valid description of the system being modeled, the essential behaviours which need to be modeled, a mathematical/physical representation or approximation of the behaviour and finally a method to verify the resemblance of the simulation to the actual system.

Historically, simulations used in different fields developed largely independently, but the spreading use of computers across all those fields have led to some unification and a more systematic view of the concept. Simulations have found a great deal of useful tools developed initially by game developers to give gamers a realistic experience of the real world. For example games like Need for Speed Pro by EA Sports, Microsoft's Flight Simulator, Nintendo Wii sports' Bowling and etcetera simulate the real conditions to very high degrees to provide a rich level or

realistic experience to the gamer. They use physics engines to represent the laws of physics for realistic kinematic and dynamic behaviour with interactions of modeled objects (for example, NVIDIA® PhysX™ engine used by Microsoft Corporation for game development), the use of rendering tools for realistic visual appearances (for example, the Microsoft XNA framework used for graphics rendering in Microsoft games) and other input/output libraries and tools for performing communication tasks with input/output devices.

Computer simulation has become a useful part of modeling many natural systems in physics, chemistry and biology [15], and human systems in economics and social science (the computational sociology) as well as in engineering to gain insight into the operation of those systems [25]. They can be used by architects to depict the in and out flow of people in a cinema theater, or by mass production factories to develop the most efficient method to assemble their products.

Traditionally, the formal modeling of systems has been via a mathematical model, which attempts to find analytical solutions enabling the prediction of the behaviour of the system from a set of parameters and initial conditions.

An interesting application of computer simulation is to simulate computers using computers or microcontrollers [15]. In computer architecture, a type of simulator, typically called an emulator, is often used to execute a program that has to run on some inconvenient type of computer, or in a tightly controlled testing environment. For example, PICsimulator by Oshonsoft software solutions (Figure 2-2) is used to simulate programs written for PIC microcontrollers by Microchip. It aids in program debugging, input output visualization and time optimization of the executed code.

As a training tool, simulation is used to train civilian and military personnel where the cost of real equipment is very high or the risks involved with untrained operation of equipment are substantial. The personnel can spend substantial time training in the simulated environment which provides tools to understand and analyze level of skill and provide feedback for improvements.

One such welding simulator has been developed by the author for a two wheeler manufacturing giant of India called TVS motor company [22]. The simulator provides the welding trainee with a typical welding like environment (Figure 2-3). The environment includes a welding booth, welding torch, smoke generated during simulated welding, vibrations on the welding torch while performing simulated welding and real time welding sounds depending on the welding technique. The simulator then provides the operator with graphical plots of various parameters being analyzed, a final predicted appearance of the weld bead and a score representing the skill level of the welding operation.

Management games (or business simulations) have been finding favor in business education in recent years. Business simulations that incorporate a dynamic model enable experimentation with business strategies in a risk free environment and provide a useful extension to case study discussions. In finance, computer simulations are often used for financial scenario planning. Risk-adjusted net present value, for example, is computed from well-defined but not always known (or fixed) inputs. By imitating the performance of the project under evaluation, simulation can provide an analysis and estimation of the project's financial parameters over an extended range of time [14].

Medical simulators are increasingly being developed and deployed to teach therapeutic and diagnostic procedures as well as medical concepts and decision making to personnel in the health

professions [31]. Simulators have been developed for training procedures ranging from the basics such as blood draw, to laparoscopic surgery and trauma care. They are also important to help on prototyping new devices for biomedical engineering problems. Currently, simulators are applied to research and development of tools for new therapies, treatments and early diagnosis in medicine.

Many medical simulators involve a computer connected to a plastic simulation of the relevant anatomy [17], [7]. Sophisticated simulators of this type employ a life size mannequin that responds to injected drugs and can be programmed to create simulations of life-threatening emergencies. In other simulations, visual components of the procedure are reproduced by computer graphics techniques, while touch-based components are reproduced by haptic feedback devices combined with physical simulation routines computed in response to the user's actions. Some simulators model the musculoskeletal interactions in the human body [15] and are meant for analysis, educational and training purposes [5]. In conjunction with robotic devices, usually manipulator arms with surgical tools, simulations are also used in surgeries which require high precision. They can be performed without the physical presence of a surgeon in the immediate vicinity of the surgery.

Simulation is an important feature in engineering systems or any system that involves many processes. For example the way atoms interact with each other in different molecules, how dry cooled air is forced in channels placed between the body and the football pad in football jerseys [23], the forces and stress on public transport vehicles like trains after going through a collision like scenario [27]. Software like Processing use Java applets to develop visual programs which can interact with the user.

Processing is an open source programming language and environment for people who want to program images, animation, and interactions. It is used by students, artists, designers, researchers, and hobbyists for learning, prototyping, and production. It is created to teach fundamentals of computer programming within a visual context and to serve as a software sketchbook and professional production tool. Programs developed in Processing are a powerful and easy to develop option to demonstrate concepts via means of simulations and graphical interactions. Processing can be used to simulate a variety of scenarios. For example, simulating interactions of lions and its prey in a wild Serengeti environment (Figure 2-4).

Flight simulators were initially developed for defense organizations to train their pilots for combat situations without having to be in the actual aircraft. The pilots train for extremely hazardous environments and learn without the risk of crashing or casing damage to life and property. Using high-fidelity display adapters and quick to respond hydraulic systems, the simulators are able to provide pilots with an experience close to actually flying an aircraft. On similar lines, marine simulators are also developed to give personnel on a ship the experience of interacting on a ship's environment.

A robotics simulator is used to create embedded applications for a specific (or not) robot without being dependent on the 'real' robot. In some cases, these applications can be transferred to the real robot (or rebuilt) without modifications. Robotics simulators allow reproducing situations that cannot be 'created' in the real world because of cost, time, or the 'uniqueness' of a resource. A simulator also allows fast robot prototyping. Robot simulators feature physics engines to simulate a robot's dynamics.

Robot simulators have accompanied robotics from a long time ago. Recently, they have become essential tools in the design and programming of industrial robots and tasks. They also

had an important role in the research field, where have been critical for the development and demonstration of many algorithms and techniques (i.e., path planning algorithms, grasp planning, and mobile robot navigation and etcetera). The main reason for its success is that they are a cheap and fast alternative to test prototypes and tune algorithms.

Our study delved into the development of a robot simulator which aids in research and development of an autonomous vehicle. The autonomous vehicle is being developed at the University of Florida at the Center of Intelligent Machines and Robotics and is intended for being driven in an urban environment [2]. Listed below are a few projects and software which have used simulations and which have similarities with the simulation developed in the thesis.

Subjugator

The Association of Unmanned Vehicle Systems International (AUVSI) and the Office of Naval Research (ONR) sponsored the 11th annual International Autonomous Underwater Vehicle Competition, held in San Diego, California at the SPAWAR facility July 29th through August 3rd, 2008. A student team at the University of Florida's Machine Intelligence Lab (MIL) developed an autonomous underwater vehicle (AUV) (Figure 2-5) for the 2008 contest. Subjugator is designed to operate underwater at depths up to 100 feet. Two 3.5" Intel Core 2 Duo computers running Microsoft Windows Server 2003 provide the processing power for monitoring and controlling all systems [15]. The mission behavior of Subjugator is controlled with the Microsoft Robotics Studio framework that communicates with a network of intelligent sensors. The sensor systems include cameras, hydrophones, Doppler Velocity Log, imaging sonar, digital compass, depth sensor, altimeter, and internal environment monitoring sensors. The submarine also makes use of custom-designed motor controllers with current sensing, actuated external devices, and other peripherals necessary for completing the mission.

3DRAD

3DRad provides a 3D environment to drive around a simulated vehicle in a simulated terrain (Figure 2-6). It has the ability to be able to program the simulator to drive autonomously and follow waypoints and avoid obstacles [1]. However, this is a commercial program with the sole purpose of simulating vehicles in a game like environment without the need to be able to interface the simulation with other programs. It cannot interact with the JAUS system on external machines.

Simulink Environment

Mathworks Simlink® is a powerful programming environment based on logic blocks connected by joints representing data flow. Simulink® has handy toolkits like SimMechanica®, Communications toolkit and Data Acquisition toolkit which help in programmers to quickly develop simulations of control diagrams and programs. The Virtual Reality toolkit, or VR Toolkit® gives the functionality of displaying three dimensional objects in three dimensional space (Figure 2-7). Some DARPA Grand challenge teams have used Mathworks products for certain components of their autonomous systems [27].

The software package, however, is not freely available and has a considerable cost associated for each of the additionally required toolkits. It is also not specifically designed to control robotic simulation applications.

Maze Simulator

A simple example of a wandering behavior for the simulated Pioneer robot using the simulated laser range finder sensors moving around in a MRDS environment developed by Trevor Taylor. The maze simulator demonstrates 3D SLAM(Simultaneous Localization And Mapping) used to create a map of the environment and navigate autonomously through it (Figure 2-8).

National Institute of Standards and Technology (NIST) Urban Search and Rescue Arenas

Interactive simulations of the NIST Reference Test Facility for Autonomous Mobile Robots Urban Search and Rescue (USAR) have been developed. The NIST USAR Test Facility is a standardized disaster environment consisting of three scenarios of progressive difficulty: Yellow, Orange, and Red arenas. The USAR task focuses on robot behaviors, and physical interaction with standardized but disorderly rubble filled environments.

The simulation is used to test and evaluate designs for teleoperation interfaces and robot sensing and cooperation that are subsequently incorporated into experimental robots (Figure 2-9). A novel simulation approach using an inexpensive game engine to rapidly construct a visually and dynamically accurate simulation for both individual robots and robot teams has been used. The simulations are based on the Unreal Engine released by Epic Games with Unreal Tournament 2003 [6].

Princeton University Entry for Defense Advanced Research Projects Agency (DARPA) Urban Challenge

Princeton University's entry into the DARPA Urban Challenge had a modified 2005 Ford Escape Hybrid SUV. It utilized Simulation, Perception, Cognition, Actuation, Substrate and Environment cycle implemented in a multi-threaded environment of Microsoft Robotics Studio and the Microsoft Windows Server 2003 operating system.

The simulation engine (Figure 2-10) has taken advantage of the Microsoft Robotics Studio framework by emulating Sensor Fusion and Actuation output while retaining their production interfaces. As a result, transitioning code from simulation to the autonomous vehicle does not require recompilation. The simulation engine has been modeled with vehicle motion using experimentally gathered vehicle dynamics and control system responses, allowing system

constants to remain largely unchanged between real-world and simulated trial runs. Stationary and moving obstacles have been added to the environment.

This chapter discussed some of the most popular applications of simulation in industry, education, health care, training and military. Some examples of projects and simulations which have a reasonably relevance to the thesis topic have been examined. Chapter 3 provides an introduction to the use of MRDS and various tools used in the development of a simulation of a physical vehicle and its sensors.

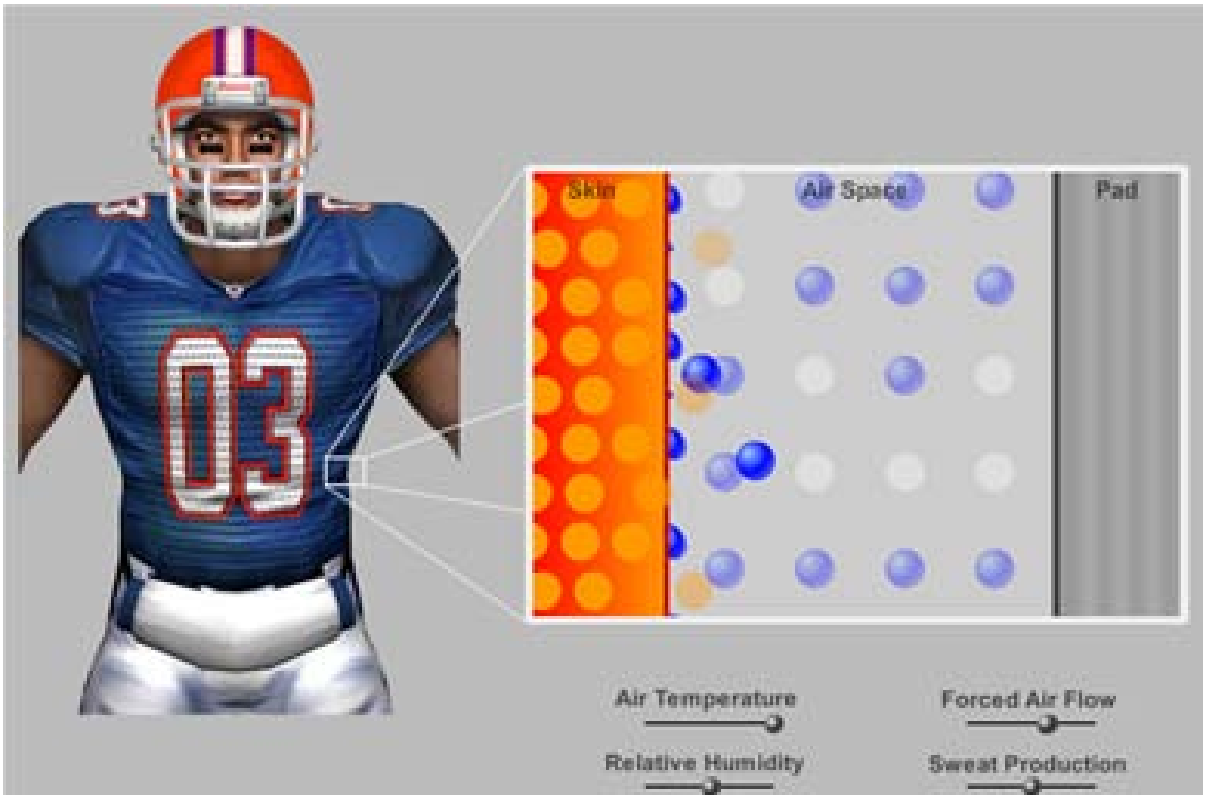

Figure 2-1. Air cooling of football pads simulated at University of Florida

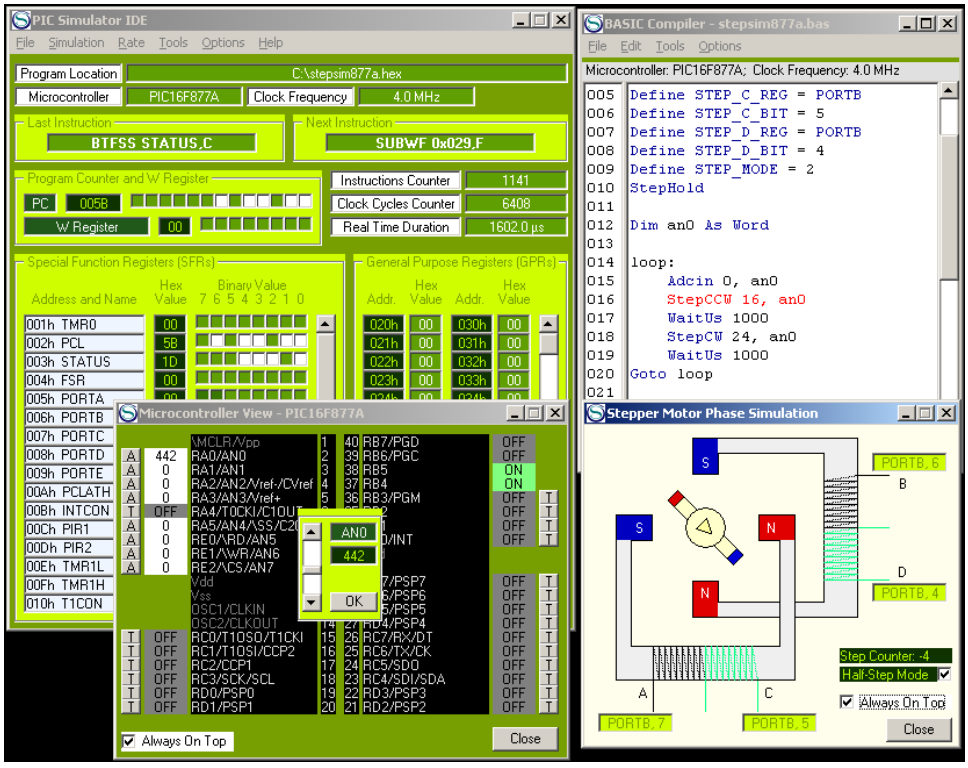

Figure 2-2. PICsimulator by Oshonsoft software solutions simulates a program to control a stepper motor controlled by a PIC16F877A microcontroller

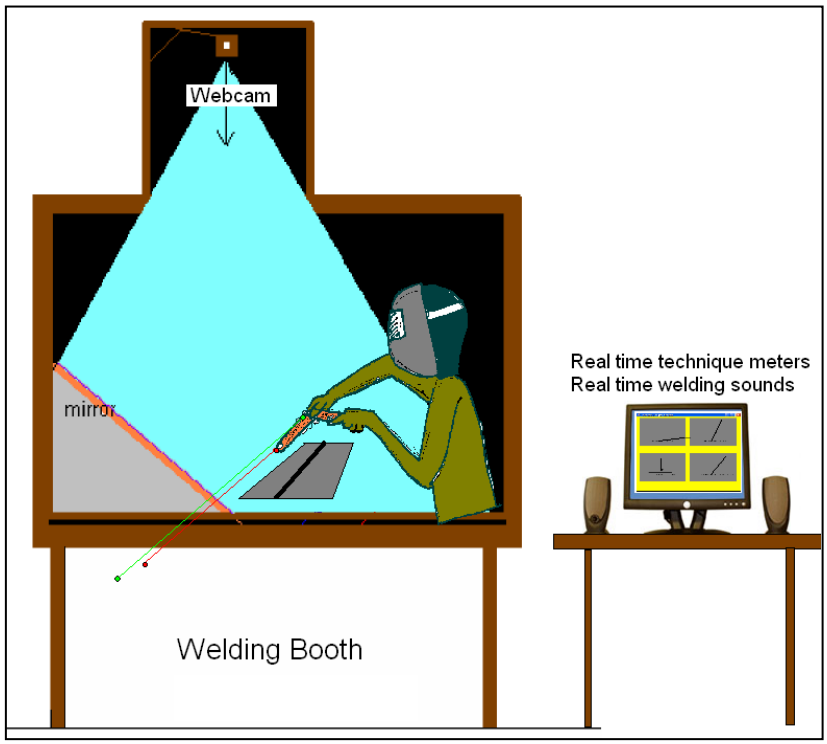

Figure 2-3. Welding simulator developed for TVS motor company, India

Figure 2-4. Simulation of predator prey interactions in an African Serengeti developed in Processing by Yonas Kolb

Figure 2-5. University of Florida's autonomous submarine for the 2008 AUVSI contest

Figure 2-6. Snapshot of 3D RAD gaming software

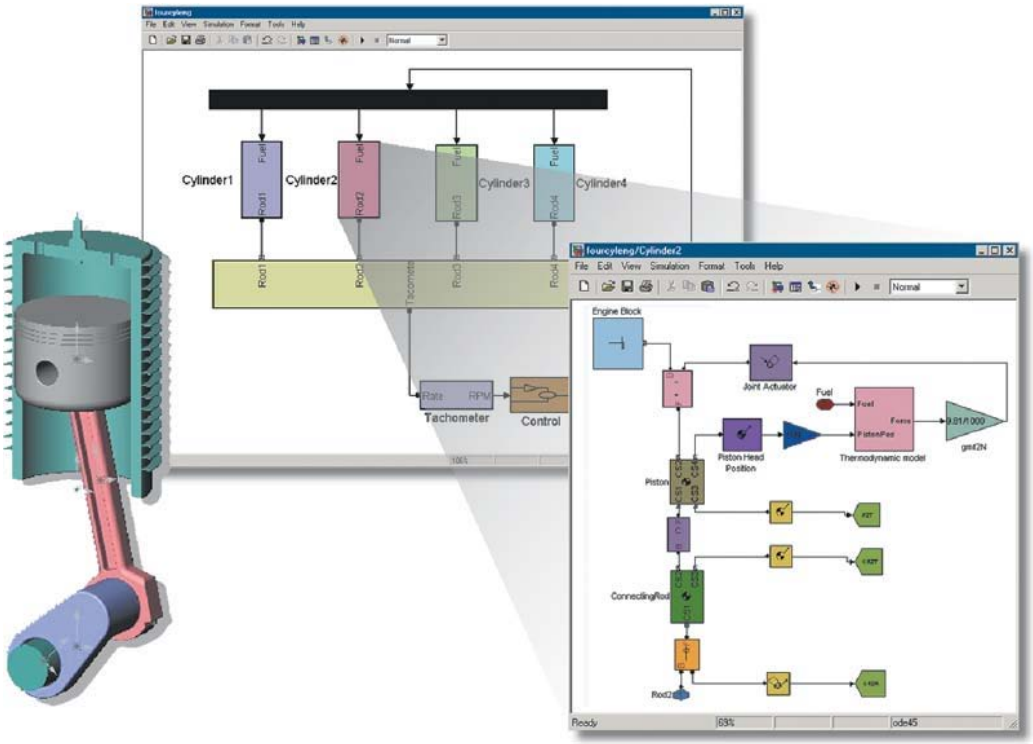

Figure 2-7. Simulink® used to simulate a piston of an internal combustion engine

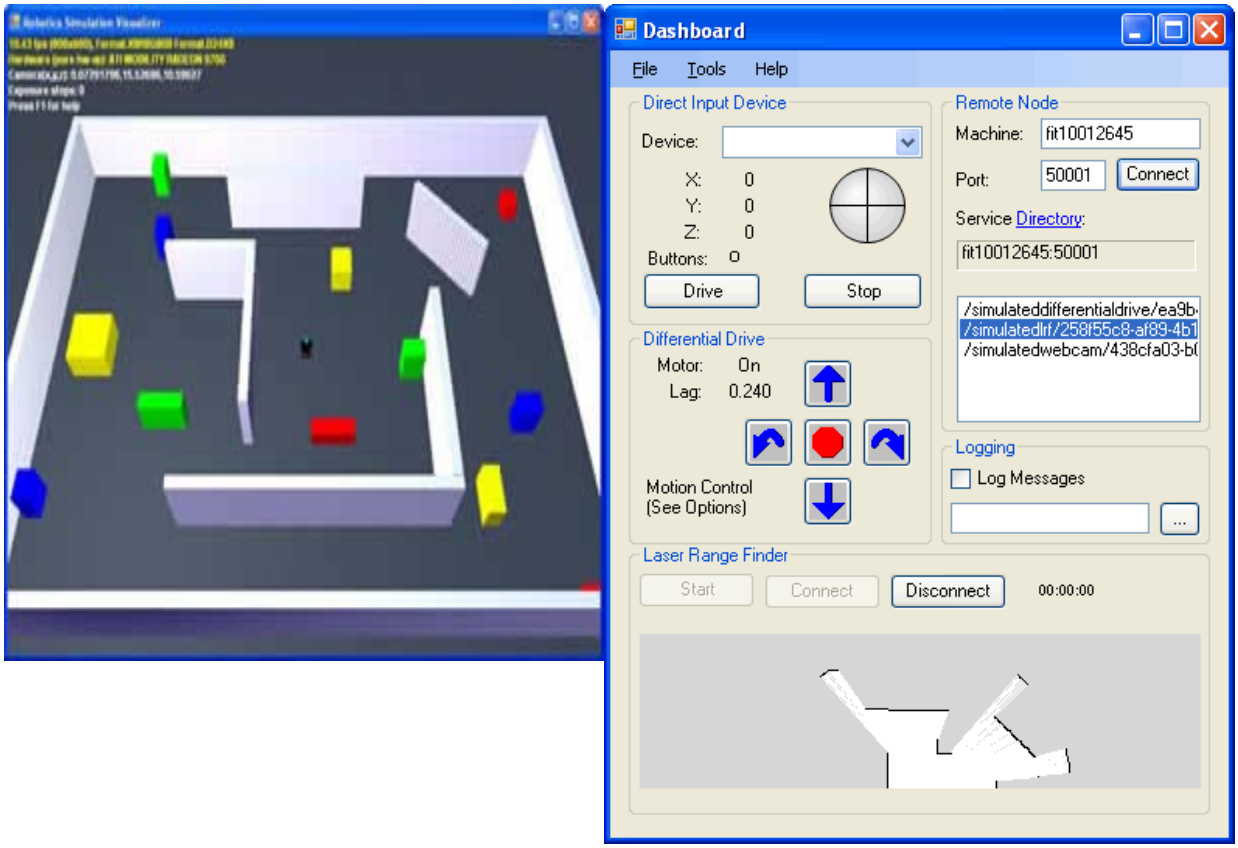

Figure 2-8. Maze simulator snapshot. A) Simulation environment. B) Dashboard with SLAM map visualization.

Figure 2-9. National Institute of Standards and Technology's Urban Search and Rescue Arena simulation using Unreal Tournament 2003 game engine

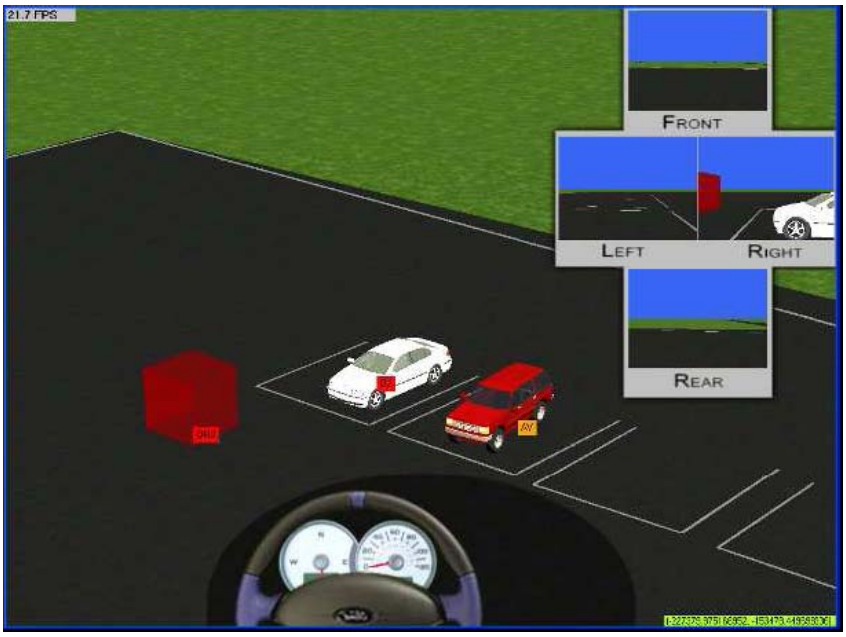

Figure 2-10. Simulator of Princeton University's DARPA Urban Challenge

CHAPTER 3
SIMULATION

Introduction to Microsoft Robotics Developers Studio (MRDS)

MRDS is a Windows-based environment for robot control and simulation. It is aimed at academic, hobby, and commercial developers and handles a wide variety of robot hardware.

Features include Visual Programming Language (VPL) (Figure 3-1) for creating and debugging robot applications, web-based and windows-based interfaces, 3D simulation (including hardware acceleration), a lightweight services-oriented runtime, easy access to a robot's sensors and actuators via a .NET-based concurrent library implementation, and support for a number of languages including C# and Visual Basic .NET, JScript, and IronPython.

MRDS comes packaged with a number of samples of working sensors, moving objects, and communicating services. The code provided is open-source and intended to be used in custom programs and services written by users. The code is written in such a way that it can be easily copied and pasted into custom programs and start working just by adding the correct references. A sample Kuka robotic manipulator arm simulation (Figure 3-2) is packaged with MRDS CTP July. It is intended to teach how to create a serial six-degree of freedom manipulator comprised of one degree of freedom joints and compose them with simple capsule shapes to build articulated, motor driven arms.

The MRDS Features Used in the Project

Microsoft robotics studio has been chosen amongst many other programming platforms for having some very useful practical capabilities which can be utilized for efficient and powerful code. These features are discussed as follows:

Partnership With a Physics Engine

MRDS has partnered with NVIDIA's AGEIA Technologies™ PhysX™ engine. It is a graphics engine to produce realistic physics behavior in simulated objects. The behaviors include particle kinematics and dynamics, solid object kinematics and dynamics, particle and object collisions, friction, gravity and etcetera (Figure 3-3).

The car has been modeled using some very essential capabilities of the provided physics engine. It has been designed to have mass, friction, coefficient of restitution, spring and damping properties for the suspension and visual features like colors, texture and lighting (Figure 3-4).

Ability to Interface With Input Devices –Xinput

Input devices, like joysticks can be incorporated quickly by a drag drop operation once a joystick service is created. The service is written keeping in mind the structure of a typical MRDS service template [15]. Code has been written for initializing the joystick, accessing its state and updating the output of the service as the states subscribed to by other services.

The car has been programmed to be controlled by any external partnered service [5], in this case the Xbox controller joystick service (Figure 3-5). These values are mapped onto acceleration/braking and steering angle values of the simulated car. In place of the Xbox service the car can easily partner with a JAUS service.

Multi-Threading Environment

The simulation is written as an event driven architecture. I.e. serially running code is avoided where ever possible. All routines run simultaneously and interact with each other at the same time depending on levels of hierarchy assigned to them. Methods are created and destroyed throughout the code and memory associated with the methods are also assigned and unassigned respectively. This aids proper memory management.

Usage of Code and Concepts from Sample Programs

MRDS comes equipped with sample programs explaining how to use each feature of the package. The supporting explanation for the examples are however slightly inadequate to explain detailed functionalities. The MRDS forums on the Microsoft website are an excellent place to find answers to common questions and problems pertinent to individual applications [17]. Some books written by MRDS developers provide an insight about the software package's capabilities and illustrate concepts with examples [8].

Simulation

PC and Console gaming paved the way of affordable, widely usable, robotics simulation. Games rely on photo-realistic visualization with advanced physics simulation running within real time constraints. This is a perfect starting point for robot simulations. The Visual Simulation Environment (VSE) is designed to be used in a variety of advanced scenarios with high demands for fidelity, visualization and scaling. The integration of the AGEIA™ PhysX™Technologies physics simulation product and a rendering engine based on Microsoft XNA (XNA's Not Acronymed) Framework has been used to produce realistic physical interaction of objects and high quality visual effects.

The VSE is implemented as a simulation service. All objects which appear in the simulation are partnered with simulation service. They are then one by one inserted in the simulation by the call of a method of the simulation service.

C# environment: Easy Interfacing with Joint Architecture for Unmanned Systems (JAUS)

Most of the algorithms and code running on the Navigator's systems are based on the JAUS architecture. JAUS is developed to be compatible in Microsoft Visual Studio's C# environment. This aids easy incorporation of JAUS components in the MRDS code. The JAUS and MRDS components are all made a part of the same project and easily exchange data.

This chapter introduces the MRDS environment and the various tools that are useful in developing the simulation of the autonomous vehicle and its environment. The next chapter will discuss in detail the implementation of each of the components of the simulated vehicle and its environment. Snippets of C# code would be provided to show parts the actual software.

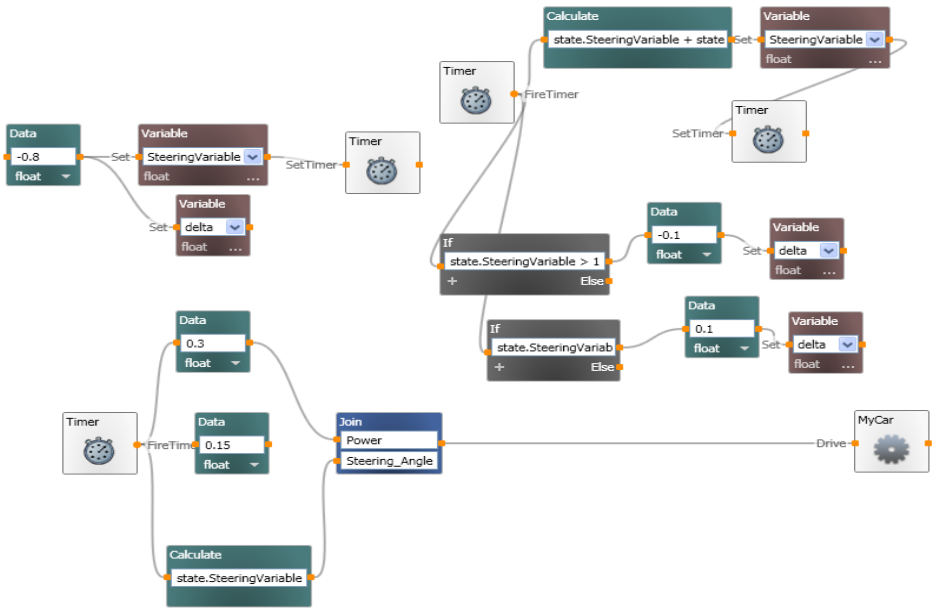

Figure 3-1. Sample VPL program showing how data is being transferred between various element

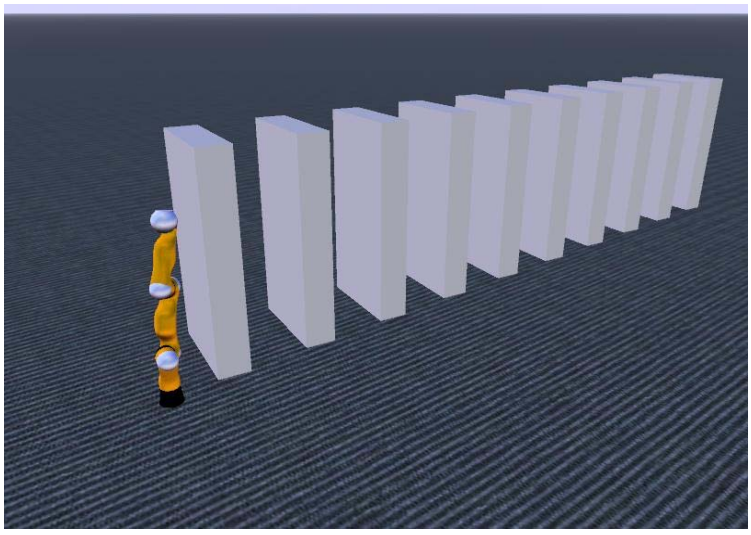

Figure 3-2. Kuka robot simulation environment packaged with MRDS community technical preview,July 2008

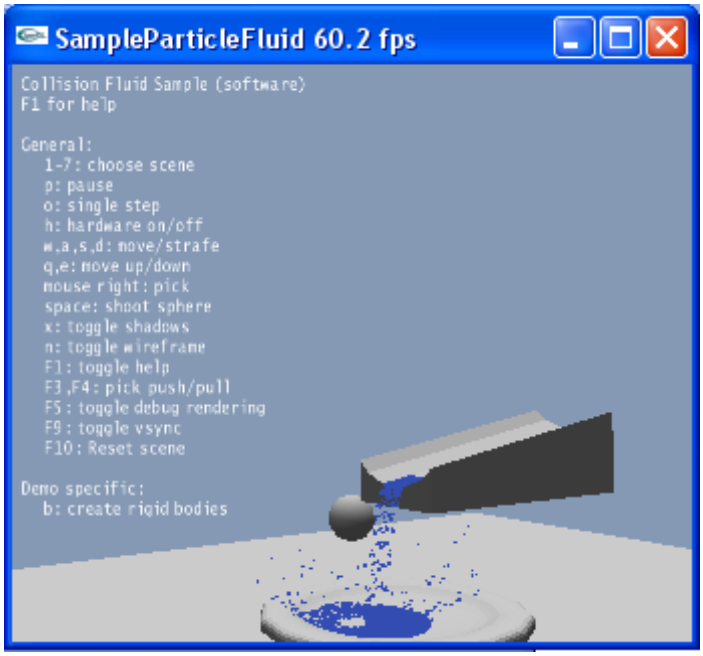

Figure 3-3. AGEIA Technologies™ PhysX™ Particle demo

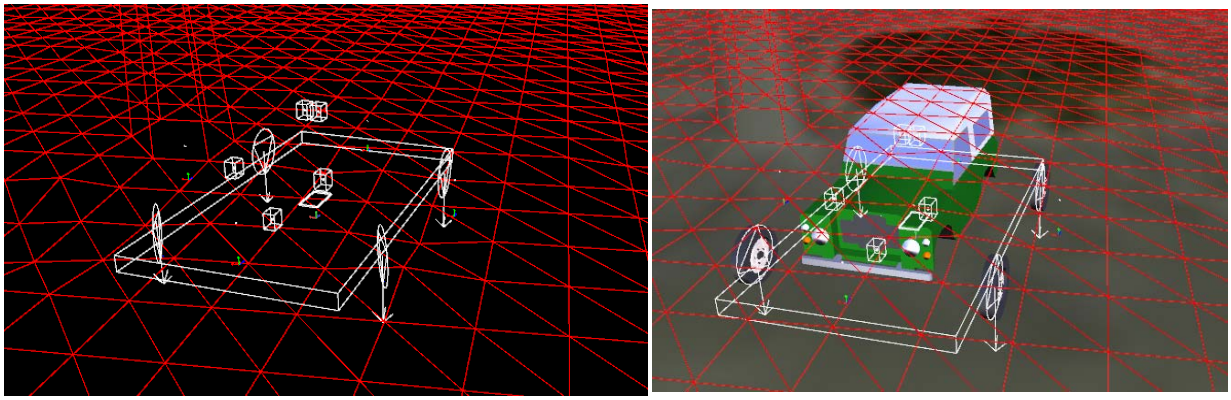

Figure 3-4. Simulation snapshots. A) Physics representation of the model. B) Physics & visual representation of the model.

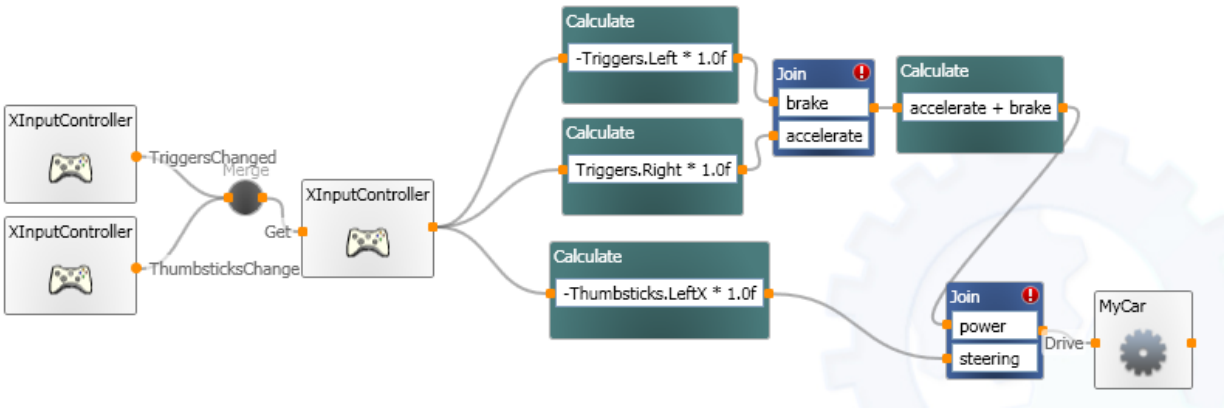

Figure 3-5. VPL diagram using Xbox joystick to control transfer data to Car service

CHAPTER 4
IMPLEMENTATION

The vehicle has been implemented as an elegant integration of physics objects interconnected with joints, visual entities using the Microsoft XNA frameworks renderings features, custom objects developed with capabilities to send and receive messages and partner with other services and features of MRDS' DSS (Decentralized Software Services) and MRDS' CRR (Concurrency and Coordination Runtime) inherent in the development of constituent services. Each component has been developed in a manner they physically exist.

Sensors

The Simulated Laser Range Finder is modeled by the Microsoft Robotics studio team due to the growing popularity of the Sick Laser Ranger Finders. The way it is done is that rays are sent from the origin of the sensor at a set angular resolution and within a fixed angular range. The distance of the impact points are noted and stored in an array in a manner similar to that done in the actual Sick sensors.

For future improvements, the simulation engine also provides the developer with material properties and using that scattering, reflections and noise can be modeled. An added feature is that the impact points are made Red in color and hence providing a visualization of them.

The angular ranges and laser distance ranges on the simulated LRFs were modified to match a range of 80m and running angular range of 90 degrees as being used in the Navigator.

```
void CreateDefaultState()
{
    _state.Units = sicklrf.Units.Millimeters;
    _state.AngularRange = 90;
    _state.AngularResolution = 0.5f;
     }
```

The simulated laser range finder is attached to the simulated car by inserting it as a child of the car entity as shown below

```
            LaserRangeFinderEntity vehicleLRF1 = CreateLRF(new Vector3(0.6f,
1.0f, -1.0f), "LRF1");
            LaserRangeFinderEntity vehicleLRF2 = CreateLRF(new Vector3(-0.6f,
1.0f, -1.0f), "LRF2");
            LaserRangeFinderEntity vehicleLRF3 = CreateLRF(new Vector3(0.0f,
0.5f, -1.2f), "LRF3");
            LaserRangeFinderEntity vehicleLRF4 = CreateLRF(new Vector3(0.1f,
1.5f, 0.0f), "LRF4");
            LaserRangeFinderEntity vehicleLRF5 = CreateLRF(new Vector3(-0.1f,
1.5f, 0.0f), "LRF5");

//            vehicleLRF1.RaycastProperties.Range = 45;
//            vehicleLRF1.RaycastProperties.EndAngle = 180;
//            vehicleLRF1.RaycastProperties.StartAngle = 90;
            CarBaseEntity.InsertEntity(vehicleLRF1);
            CarBaseEntity.InsertEntity(vehicleLRF2);
            CarBaseEntity.InsertEntity(vehicleLRF3);
            CarBaseEntity.InsertEntity(vehicleLRF4);
            CarBaseEntity.InsertEntity(vehicleLRF5);
```

The obstacle distance data from each of the sensors can be accessed and send over a port by creating a noifications port and then creating a listerner which is activated whenever the sensor is ready throw a new set of readings. Shown below is code for accessing and broadcasting readings from one of the sensors

```
            Port<RaycastResult> notificationTarget = new
Port<RaycastResult>();
            vehicleLRF1.Register(notificationTarget);

            Activate(Arbiter.Receive(true /* true means receiver is
persistent */, notificationTarget,
            raycastResults =>
            {
                LogInfo("impact point type" + raycastResults.GetType());
                RaycastImpactPoint min=new RaycastImpactPoint();
                min.Position.W=20000;
                foreach (RaycastImpactPoint pt in
raycastResults.ImpactPoints)
                {
                    if (pt.Position.W < min.Position.W) min.Position.W =
pt.Position.W;
                }
                LogInfo("closest point on L1 is : {0}mm away" +
min.Position.W);
// do something with raycastResults here
            }));
```

The JAUS Interface

On the Navigator, the system architecture is a natural extension of the JAUS Reference Architecture, Version 3.2, which defines a set of reusable components and their interfaces. The actual core software to support the JAUS messaging system was developed and extensively tested for the previous Grand Challenge and supports the current effort.

The JAUS libraries are referenced in the main project of the car and environment simulation. The class defining the car and its behavior is made into a partial class so that a part of the functionality of the class can be coded in a separate file, but as the remaining part of the partial car class. A separate file called JAUS component.cs is created and the JAUS components are initialized as shown in the code below.

```
    partial class MyCarService
    {
        JausComponent _component = new
JausComponent(EServiceType.GLOBAL_POSE_SENSOR);
        static int _jausPort = 4700;
        static int _intPort = 4800;

        void StartComponent()
        {
            // services
            List<MessageInfo> inputMsgs = new List<MessageInfo>();
            List<MessageInfo> outputMsgs = new List<MessageInfo>();
            outputMsgs.Add(new
MessageInfo((int)EJausMessageType.REPORT_GLOBAL_POSE, uint.MaxValue, 30));

            _component.ServiceConnectionManager.RegisterSupportedServices(new
ServiceInfo(EServiceType.GLOBAL_POSE_SENSOR, inputMsgs, outputMsgs));

            // operating rate
            _component.ServiceConnectionManager.OperatingRate = 30;
            _component.ServiceConnectionManager.OutboundMessage = new
JausMessage(new JausHeader() { Destination = new JausAddress(),

Source = new JausAddress(),

CommandCode=(int)EJausMessageType.REPORT_GLOBAL_POSE }, new
ReportGlobalPose());

            // state change event
            _component.OnStateChange += new
JAUS.NMI.NMI.StateChangedCallback(_component_OnStateChange);
```

```
            _component.OnConnectionStatusUpdate += new
JAUS.NMI.NMI.ConnectionStatusUpdateCallback(_component_OnConnectionStatusUpda
te);

            // attach message processor
            _component.JMH.AddMessageProcessor(this);

            // initialization
            _component.InitNMI(_jausPort, _intPort);

            // state transition
            _component.SetState(EComponentState.STARTUP);
        }

        void _component_OnConnectionStatusUpdate(bool connected)
        {
            if (connected)
                _component.SetState(EComponentState.READY);
        }

        void _component_OnStateChange(EComponentState state)
        {
            switch(state)
            {
                case EComponentState.STARTUP:
                    _component.CheckIn((int)EServiceType.GLOBAL_POSE_SENSOR,
_jausPort);
                    Console.WriteLine("Checking in.");
                    break;
                case EComponentState.READY:
                    _updateTimer.Start();
                    Console.WriteLine("Component is Ready.");
                    break;
                case EComponentState.SHUTDOWN:
                    _component.CheckOut();
                    Console.WriteLine("Checking out.");
                    break;
                default:
                    break;
            }
        }

        void UpdateGPose(Pose pose)
        {
            ReportGlobalPose gPose = new ReportGlobalPose();

            gPose.PresenceVector.Vector = uint.MaxValue;
            gPose.Latitude = pose.Position.X;
            gPose.Longitude = pose.Position.Z;
            gPose.Elevation = pose.Position.Y;

            JausMessage msg =
_component.ServiceConnectionManager.OutboundMessage;
            lock (msg)
            {
                msg.Contents = gPose;
            }
```

```
        }

        [MessageProcessor((int)EJausMessageType.CREATE_SERVICE_CONNECTION)]
        void TestInput(JausMessage msg)
        {
            CreateServiceConnection svc = msg.Contents as
CreateServiceConnection;
            Console.WriteLine("Service connection request: {0}",
svc.CommandCode);
        }
    }
```

The component was referenced to the MRDS car implementation so that the MRDS and JAUS components could intercommunicate.

```
JausComponent _component = new JausComponent
(EServiceType.GLOBAL_POSE_SENSOR);
static int _jausPort = 4700;
static int _intPort = 4800;
```

An instance of the JAUS component was then created in the MRDS car class and data send to it in a typical JAUS message protocol. A JAUS node was then initialized on the computer where the project was run and then the project was ready to transmit and receive JAUS messages to and from any other JAUS node on the network and transmit and receive data to the MRDS project.

Terrain Modeling

A satellite image is first acquired of the track and basic tweaking of brightness, contrast and sharpness is done (Figure 4-1). The image is cropped and a region of interest is selected. Roads are demarcated with a black color and are given a leveled surface. Road's surroundings are colored with a slightly lighter color to get elevated pavements (Figure 4-2).The terrain height map image is used to create a terrain using the average grayscale value of each pixel on the image to map on the height of the terrain (Figure 4-3).Finally a texture image is overlaid onto the three dimensional terrain to give a realistic environment appearance (Figure 4-4).

The implementation of the sky and terrain has been shown in the code snippet below.

```
#region Environment Entities

void AddSky()
{
    // Add a sky using a static texture. We will use the sky texture
    // to do per pixel lighting on each simulation visual entity
    SkyEntity sky = new SkyEntity("sky.dds", "sky_diff.dds");
    SimulationEngine.GlobalInstancePort.Insert(sky);

    // Add a directional light to simulate the sun.
    LightSourceEntity sun = new LightSourceEntity();
    sun.State.Name = "Sun";
    sun.Type = LightSourceEntityType.Directional;
    sun.Color = new Vector4(0.8f, 0.8f, 0.8f, 1);
    sun.Direction = new Vector3(-1.0f, -1.0f, 0.5f);
    SimulationEngine.GlobalInstancePort.Insert(sun);
}

private void AddTerrain()
{
    TerrainEntity ground = new TerrainEntity(
             "victorville5_30percent.bmp",//"terrain.bmp",
             //"jumpTerrain.bmp",
             //"wood_cherry.bmp",
             //"woodfloor.jpg",//"terrain_tex.jpg",
             "victorville31.jpg",

             new MaterialProperties("ground",
                 0.2f, // restitution
                 0.9f, // dynamic friction
                 0.9f) // static friction
             );
   //ground.MeshScale=new Vector3(1.0f, 0.005f, 1.0f);
    //ground.
    SimulationEngine.GlobalInstancePort.Insert(ground);

}
     #endregion
```

Car Modeling

Ackerman Steering

Ackerman steering has been incorporated in the steering mechanism of the model [2]. Although most modern vehicles do not follow an exact Ackerman model, since it does not take into account the dynamics of the vehicle, at low speeds it is a fairly good mechanism for no slip turning movement of a vehicle.

The Navigator has a servo motor installed onto the steering wheel. The JAUS system gives wrench commands between -1 and 1 which correspond to the steering angle. Similarly the

simulation model receives desired steering angles between -1 to 1. The steering effort is then mapped onto a value between the maximum steering extreme angles of the inner wheel. The outer wheel angle is calculated to be Ackerman compliant with the inner wheel angle. The code implementing the Ackerman compliant angle is shown below.

```
    #region Ackerman Steering convert
                    double d = CarBaseEntity.WheelBase;
                    double b = CarBaseEntity.DistanceBetweenWheels;
                    double MAXANGLE = 70;
                    double A2 = 0;
                    if (m.Body._steeringAngle > 0)
                    {
                        A2 = d / (d / Math.Tan(m.Body._steeringAngle *
MAXANGLE * (float)Math.PI / 180.0f) + b);
                        A2 = Math.Atan(A2);
                        A2 /= MAXANGLE * (float)Math.PI / 180.0f;
                        leftAngle = m.Body._steeringAngle;
                        rightAngle = A2;
                    }
                    else if (m.Body._steeringAngle < 0)
                    {
                        A2 = d / (d / Math.Tan(-m.Body._steeringAngle *
MAXANGLE * (float)Math.PI / 180.0f) + b);
                        A2 = -Math.Atan(A2);
                        A2 /= MAXANGLE * (float)Math.PI / 180.0f;
                        leftAngle = A2;
                        rightAngle = m.Body._steeringAngle;
                    }
    #endregion
```

Wheel Entity

Wheels are the means of motion for most robotic vehicle applications. It is for this reason that the MRDS team has provided developers with a preprogrammed model of a simple wheel. The basic features that are provided are wheel rotation and wheel radius. The wheels have a point mass and are simple two dimensional discs in their physics representation.

For the application of the car, instances of the wheel entities are attached to the chassis of the vehicle as children elements of the chassis. The front wheels are written with functions that provide access to the relative orientation of the wheels and these are private properties of the

class. The wheels are modeled as thin discs which can rotate (roll) and twist (steer) and have a point contact on the ground (Figure 4-3).

Friction

Tires interact with the road by allowing some minimal amount of slippage due to lateral and longitudinal friction [6]. The ranges for these values were taken from a popular race car simulator game developer community (Figure 4-4), www.unity3d.com [8].

It has been implemented in the code as shown below.

```
            float LoAsSlip = 0.04f; float LoExSlip = 0.01f;
            float LoExValue = 0.01f; float LoAsValue = 0.06f;
            float LoStiffness = 50000f;
            float LaStiffness = 10000f;
// front left wheel
            WheelShapeProperties w = new WheelShapeProperties("front left
wheel", FrontWheelMass, FrontWheelRadius);
            w.TireLateralForceFunction = new TireForceFunctionDescription();
            w.TireLateralForceFunction.AsymptoteSlip = LoAsSlip;
w.TireLateralForceFunction.AsymptoteValue = LoAsValue;
w.TireLateralForceFunction.ExtremumSlip = LoExSlip;
w.TireLateralForceFunction.ExtremumValue = LoExValue;
            w.TireLateralForceFunction.StiffnessFactor = LaStiffness;
```

Finally all components are integrated together to develop a Car entity as shown below

```
        #region CAR_PA Robot entity
        void AddVehicle(Vector3 CarPosition)
        {
            CarBaseEntity = new SimpleFourByFour(CarPosition);
            CarBaseEntity.State.Name = "Navigator";
            CarBaseEntity.FrontWheelMesh = "4x4wheel.obj";
            CarBaseEntity.RearWheelMesh = "4x4wheel.obj";
            LogInfo("Car created!!!");
            CarBaseEntity.State.Velocity = new Vector3(1, 1, 5);
            CarBaseEntity.State.AngularVelocity = new Vector3(0.0f, 0.0f,
0.0f);
            CarBaseEntity.SuspensionTravel = 0.127f;
            CarBaseEntity.SuspensionTravel = 0.5f; //off road
            CarBaseEntity.State.Assets.Mesh = "4x4Body.obj";
            CarBaseEntity.MotorTorqueScaling = 60.0f;
            CarBaseEntity.DistanceBetweenWheels = 3.5f;
            CarBaseEntity.WheelBase = 4.0f;
//The Laser range finders are inserted as children of the car vehicle
             LaserRangeFinderEntity vehicleLRF1 = CreateLRF(new Vector3(0.6f,
1.0f, -1.0f), "LRF1");
            LaserRangeFinderEntity vehicleLRF2 = CreateLRF(new Vector3(-0.6f,
1.0f, -1.0f), "LRF2");
```

```
            LaserRangeFinderEntity vehicleLRF3 = CreateLRF(new Vector3(0.0f,
0.5f, -1.2f), "LRF3");
            LaserRangeFinderEntity vehicleLRF4 = CreateLRF(new Vector3(0.1f,
1.5f, 0.0f), "LRF4");
            LaserRangeFinderEntity vehicleLRF5 = CreateLRF(new Vector3(-0.1f,
1.5f, 0.0f), "LRF5");

//            vehicleLRF1.RaycastProperties.Range = 45;
//            vehicleLRF1.RaycastProperties.EndAngle = 180;
//            vehicleLRF1.RaycastProperties.StartAngle = 90;

//some points are added around the car to provide a non-uniform weight
distribution similar to the actual car and also provide physical support

            SingleShapeEntity weight1=AddWeight("bigWeight",new
Vector3(0,0f,0),2);
            SingleShapeEntity wheelieBar = AddWeight("Wheelie Bar",new
Vector3(0f, 0.3f, 2.0f), 1);
            SingleShapeEntity noseWheelieBar = AddWeight("Nose Wheelier Bar",
new Vector3(0f, 0.3f, -2.0f), 1);
            SingleShapeEntity leftBar = AddWeight("Left topple bar", new
Vector3(-2.0f, 0.3f, 0.0f), 1);
            SingleShapeEntity rightBar = AddWeight("Right topple bar", new
Vector3(2.0f, 0.3f, 0.0f), 1);

//each componenet of the vehicle is added as childeren of the car entity

            CarBaseEntity.InsertEntity(weight1);
            CarBaseEntity.InsertEntity(wheelieBar);
            CarBaseEntity.InsertEntity(noseWheelieBar);
            CarBaseEntity.InsertEntity(leftBar);
            CarBaseEntity.InsertEntity(rightBar);

            CarBaseEntity.InsertEntity(vehicleLRF1);
            CarBaseEntity.InsertEntity(vehicleLRF2);
            CarBaseEntity.InsertEntity(vehicleLRF3);
            CarBaseEntity.InsertEntity(vehicleLRF4);
            CarBaseEntity.InsertEntity(vehicleLRF5);
            //            CarBaseEntity

            Port<RaycastResult> notificationTarget = new
Port<RaycastResult>();

            vehicleLRF1.Register(notificationTarget);
            Activate(Arbiter.Receive(true /* true means receiver is
persistent */, notificationTarget,
            raycastResults =>
            {
                LogInfo("impact point type" + raycastResults.GetType());
                RaycastImpactPoint min=new RaycastImpactPoint();
                min.Position.W=20000;
                foreach (RaycastImpactPoint pt in
raycastResults.ImpactPoints)
                {
                    if (pt.Position.W < min.Position.W) min.Position.W =
pt.Position.W;
                }
```

```
                LogInfo("closest point on L1 is : {0}mm away" +
min.Position.W);
            // do something with raycastResults here
            }));

            // create Camera Entity ans start SimulatedWebcam service
            CameraEntity camera = CreateCamera();
            // insert as child of motor base
            CarBaseEntity.InsertEntity(camera);
            CameraSprite camSprite = new CameraSprite(8.0f, 6.0f,
SpritePivotType.Center, 0, new Vector3(0, 5, 0));
            camSprite.State.Name = "NaviGator camSprite";
            camSprite.Flags |= VisualEntityProperties.DisableBackfaceCulling;
            camera.InsertEntity(camSprite);
            SimulationEngine.GlobalInstancePort.Insert(CarBaseEntity);

            // Start simulated arcos motor service

            //CarBaseEntity.SetWheelAngles(0.45f, 0.45f, 0, 0);

            Activate(Arbiter.Receive<Drive>(true, _mainPort, m =>
                {CarBaseEntity.SetMotorTorque(m.Body._power, m.Body._power);
                    double leftAngle = m.Body._steeringAngle, rightAngle =
m.Body._steeringAngle;

                    #region Ackerman Steering convert
                    double d = CarBaseEntity.WheelBase;
                    double b = CarBaseEntity.DistanceBetweenWheels;
                    double MAXANGLE = 70;
                    double A2 = 0;
                    if (m.Body._steeringAngle > 0)
                    {
                        A2 = d / (d / Math.Tan(m.Body._steeringAngle *
MAXANGLE * (float)Math.PI / 180.0f) + b);
                        A2 = Math.Atan(A2);
                        A2 /= MAXANGLE * (float)Math.PI / 180.0f;
                        leftAngle = m.Body._steeringAngle;
                        rightAngle = A2;
                    }
                    else if (m.Body._steeringAngle < 0)
                    {
                        A2 = d / (d / Math.Tan(-m.Body._steeringAngle *
MAXANGLE * (float)Math.PI / 180.0f) + b);
                        A2 = -Math.Atan(A2);
                        A2 /= MAXANGLE * (float)Math.PI / 180.0f;
                        leftAngle = A2;
                        rightAngle = m.Body._steeringAngle;
                    }
                    #endregion
                    CarBaseEntity.SetWheelAngles((float)leftAngle,
(float)rightAngle, 0, 0);
                }));

/*
            #region Simulated LRF distance accessing
            Port<RaycastResult> notificationTarget1 = new
Port<RaycastResult>();
```

```
            vehicleLRF1.Register(notificationTarget1);

            Activate(Arbiter.Receive(true , notificationTarget1,
            raycastResults =>
            {
                // do something with raycastResults here
                //raycastResults.ImpactPoints.ToString();
                System.Console.WriteLine("LaserData: {0}",
raycastResults.ImpactPoints.ToString());
            }));

            #endregion
*/
      }
```

This chapter discussed the implemention of the individual components of the simulation with relevant C# code from the project. Chapter 5 consolidates the results obtained and discusses future work.

Figure 4-1. Satellite view of a typical urban environment. Courtesy www.tatuGIS.com

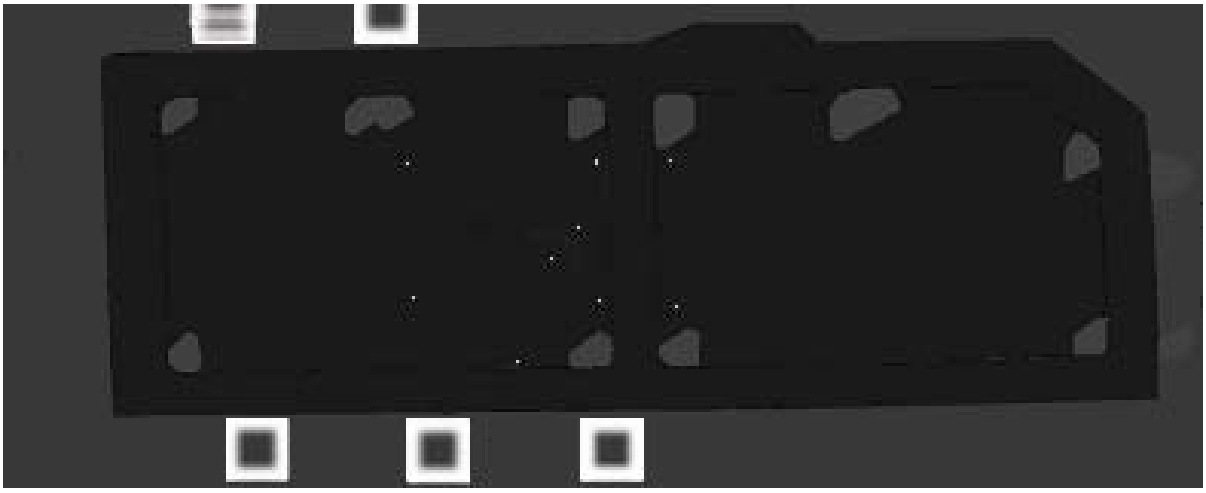

Figure 4-2. Image created for the terrain height map

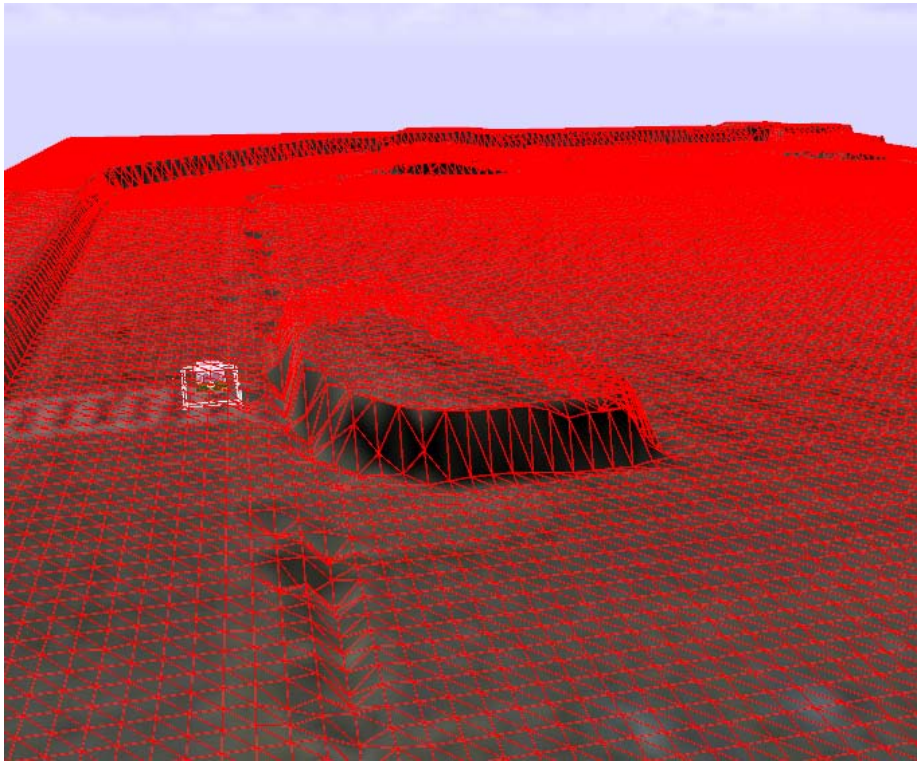

Figure 4-3. Terrain created from height map image

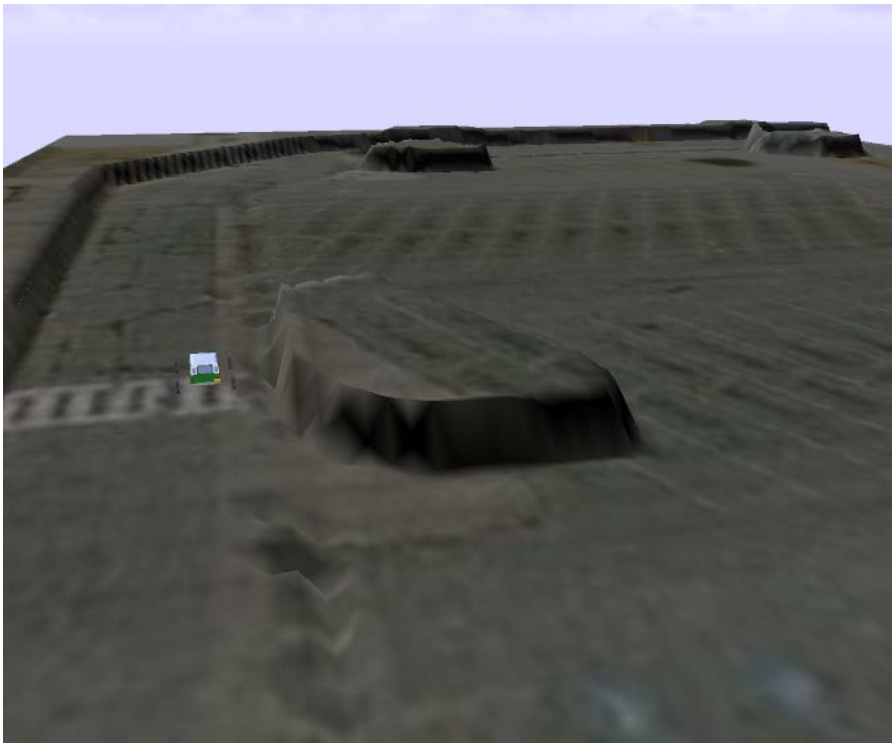

Figure 4-4. Final terrain with image texture overlaid

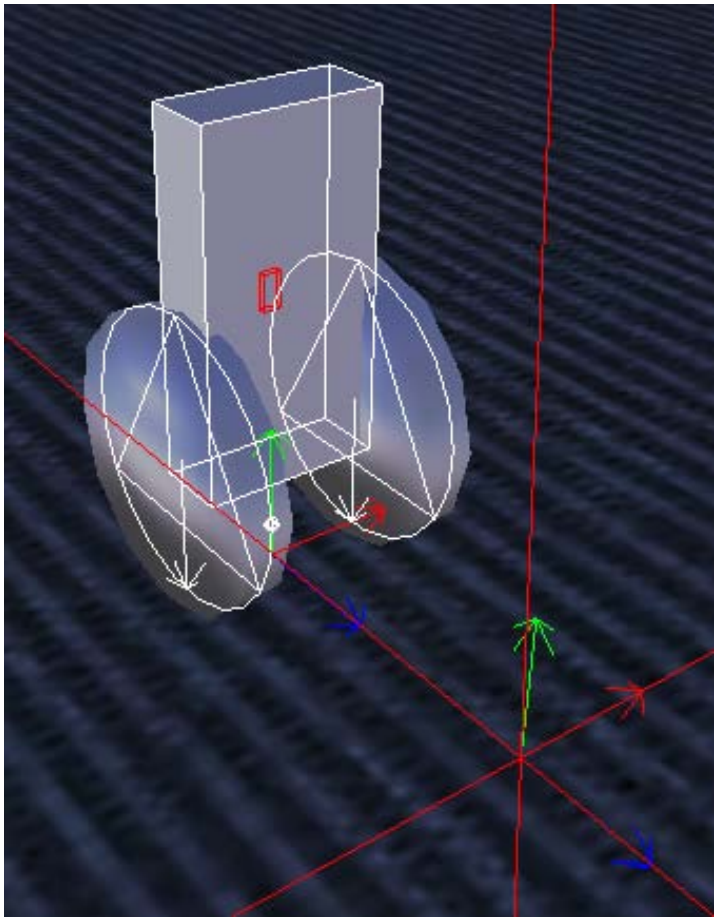

Figure 4-3. Physics representation of a simple robot with two wheel entities

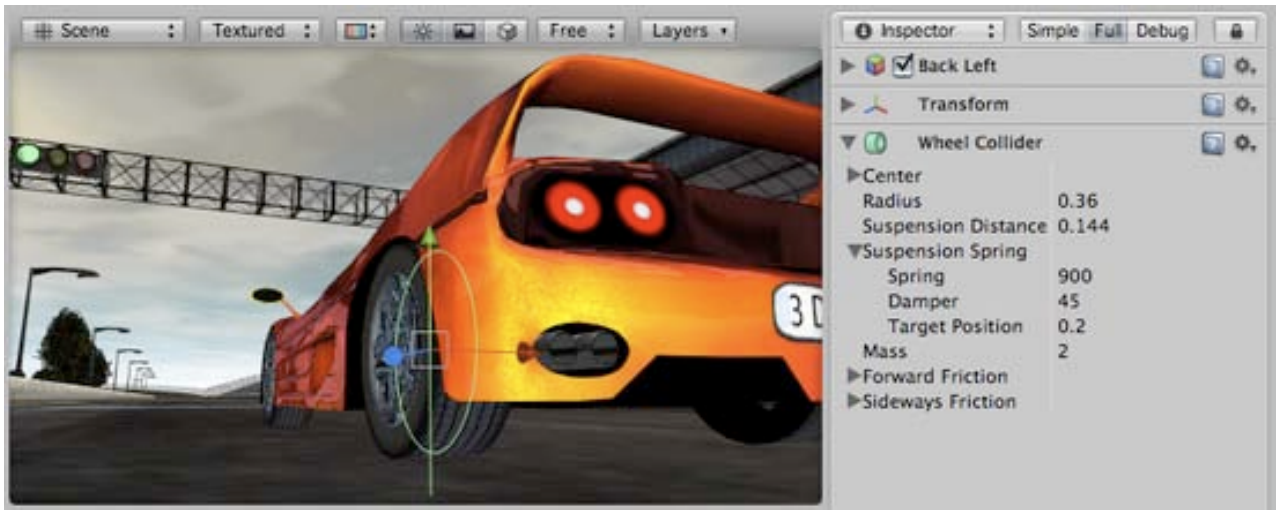

Figure 4-4. Wheel collider component. car model courtesy of ATI Technologies Inc.

CHAPTER 5
RESULTS AND FUTURE WORK

Results

An environment representing a typical urban scene was created. 3D objects with physics characteristics defined were created and inserted into 3D scene. Objects include a

- Sky dome
- A distant omni directional light source as a representation of the sun
- Terrain fixed to the world coordinate system with material properties of coefficient of friction, texture and a 3D shape profile of a typical urban environment
- Telephone poles, traffic light poles, electricity pole representation as high elevated points on the terrain
- Buildings, houses, walls as terrain features with height over 1 meter
- Roads were represented as smooth, ground level terrains with pavements around them with 20cm elevation

The simulated car was controlled by messages consisting of data in a structure similar to that given to the Navigator. Wrench commands on the Navigator comprise mainly of 3 data elements

- Steering angle mapped onto values between -1 to 1
- Acceleration effort mapped onto values between 0 to 1
- Braking effort mapped onto values between 0 to 1

Object 5-1. Video file of car being driven manually with joystick in simulated environment (.avi file 4.82 MB)

A JAUS component was integrated into the MRDS program demonstrating the communication of data to devices on separate nodes on a network. Data from the MRDS program was sent to a JAUS node on the same computer. Then the data was immediately sent to a JAUS node on a different computer on the network.

Realistic properties of vehicle like dynamics were demonstrated (Figure 5-1). The dynamics can be inferred from

- Skidding of vehicle during excessive turning
- Nose dipping and nose rise during braking and acceleration respectively
- No slippage Ackerman steering
- Four point contact of vehicle on a non-planar terrain due to spring and damper on wheels

A community developed environment called KIA Urban Challenge was also explored. The car simulation was included into the KIA urban challenge environment. The community developed environment is a high quality representation of an urban environment developed for MRDS simulation (Figure 5-2). The simulated car was inserted into the simulation of the KIA Urban Challenge and made to interact with all objects present in the environment.

Summary of Results

A simulated car with vehicle like dynamics was created. Simulated Laser and camera sensors were attached to the car. The simulated car was programmed to respond to commands in a manner similar to the commands given to the actual car being simulated. The simulated car was inserted in an urban environment and driven using joystick commands. Data transfer from the MRDS environment to another computer on the network via a JAUS connection was established and demonstrated.

Future Work

MRDS-JAUS communication has already been established. The next step is to control the simulated car using the wrench command send to the simulation via JAUS wrench commands. Eventually, the simulated car can be driven in the simulated environment using the already existing autonomous JAUS system used to drive the real car in the real environment.

Simulated Laser sensors of a single type have been attached to the simulated car. Variations of the sensors need to be made to replicate the variations the various characteristics

(like angular resolution, angular range and refresh rate) of the different models of laser sensors used. This can be achieved by simply creating a new instance of the simulated laser sensor and modifying the state properties of that particular instance. The positions and orientations of the sensors relative to the simulated car have been decided by visual intuitive approximation. The positions and orientations should be exactly measured from the real car and replicated in the simulation as a linear displacement and a quaternion rotation.

Braking and acceleration behaviors need to be tweaked for the car to behave with higher quality of realism. This can be achieved by comparing the motion of the simulated car with motion of the actual car given the same input.

Figure 5-1. Close-up of simulated vehicle

Figure 5-2. Simulated car inserted into environment developed by KIA urban challenge team

LIST OF REFERENCES

[1] 3D Rad development team, “3D Rad v6”, ©3D Rad Corporation, www.3drad.com [Accessed on 21st October 2008], Orlando, FL , 2008

[2] Crane, C., Armstrong, D., Arroyo, A., Baker, A., Dankel, D., Garcia, G., Johnson, N., Lee, J., Ridgeway, S., Schwartz, E., Thorn, E., Velat, S., Yoon, J., and Washburn, J., "Team Gator Nation's Autonomous Vehicle Development for the 2007 DARPA Urban Challenge", Journal of Aerospace Computing, Information, and Communication, Vol. 4, Dec. 2007, pp. 1059-1085.

[3] DARPA Urban Challenge Program Manager, “DARPA Urban Challenge Technical Evaluation Criteria Document”, Defense Advanced Research Projects Agency, http://www.darpa.mil/GRANDCHALLENGE/docs/Technical_Evaluation_Criteria_031607.pdf [Accessed 28th October 2008], 16 Mar. 2006.

[4] D. Burnette, T. Feeney, Z. Fuchs, G. Cieslewski, M. Hunter Longshore,J. Morales, L. Vega, P. Walters, S. Cheam, O. Allen, C. Adam, B. Hood, E. Schwartz, A. Arroyo, “University of Florida’s 2008 ASUVSI entry” Annual International Autonomous Underwater Vehicle Competition, 11th Jul. 2008.

[5] D. Murphy, B. Challacombe, T. Nedas, O. Elhage, K. Althoefer, L. Seneviratne, P. Dasgupta, "Equipment and technology in robotics" Arch. Esp. Urol. 60 (4): 349–55. PMID 17626526. Retrieved on 2008-08-26.

[6] Epic Games, Inc, “Unreal Tournament 3©”, © Midway Inc., www.unrealtournament.com [Accessed 2nd November 2008], Chicago, Il, 2007.

[7] J. Leung, E. Foster"How do we ensure that trainees learn to perform biliary sphincterotomy safely, appropriately, and effectively?". Curr Gastroenterol Rep 10 (2): 163–8. doi:10.1007/s11894-008-0038-3. PMID 18462603, April 2006. Retrieved on 2008-08-26.

[8] J.Taylor, M.Andrews, “Wheel collider physics”, ATI Technologies Inc., http://unity3d.com/support/documentation/Components/class-WheelCollider.html#Friction [Accessed 29th October 2008], Tokyo, Japan, 2004.

[9] J.Wang, M.Lewis, J.Gennari, “A Game Engine Based Simulation Of The NIST Urban Search And Rescue Arenas”, Proceedings of the Winter Simulation Conference, 2003.

[10] K. Johns, T. Taylor , “Professional Microsoft Robotics Developer Studio”, Ch.2, pp. 40-63, May 2008.

[11] K. U. Dörr, J. Schiefele, I. Kubbat “Virtual Cockpit Simulation for Pilot Training”, Institute for Flight Mechanics and Control Technical University Darmstad, 2002.

[12] K. Johns, T. Taylor , “Professional Microsoft Robotics Developer Studio”, Ch.2, pp. 70-83, May 2008.

[13] K. Nice, “How Car Steering Works”, http://auto.howstuffworks.com/steering.htm#, HowStuffWorks, Inc. ©, Discovery Communications, Raleigh, NC,2008.

[14] L. Bressman, J. Browne ,C. Nanninga ,B. Weintrob ,”Financial simulation model: Assessing project risk at the Port Authority of New York and New Jersey”, Proc. 10th conference on Winter simulation, Volume 2. Miami Publisher IEEE Computer Society Press Los Alamitos, CA, USA , 2001.

[15] L. Chen, Y. Liu, C. Chen, C. Kao, I. Huang, “Parameterized Embedded In-circuit Emulator and Its Retargetable Debugging Software for Microprocessor/Microcontroller/DSP Processor”, Department of Computer Science and Engineering, National Sun Yat-Sen University Kaohsiung, 80424, Taiwan, R.O.C.

[16] L. Li, F.Wang, Q. Zhou, “Integrated Longitudinal and Lateral Tire/Road Friction Modeling and Monitoring for Vehicle Motion Control”, Intelligent Transportation Systems, IEEE Transactions, Vol. 7, Iss. 1, pp. 1-19, Mar. 2006.

[17] L. Hugues, N. Bredeche,“Simbad: An Autonomous Robot Simulation Package for Education and Research”, Equipe Inf´erence et Apprentissage, TAO / INRIA uturs Laboratoire de Recherche en Informatique, Bat.490, Universit´e de Paris-Sud, 91405 Orsay Cedex, France Paris, France, 2001.
http://www.lri.fr/ bredeche

[18] Mondorobot, “Robochamps – KIA Urban Challenge”, ©Microsoft Corporation, www.robochamps.com [Accessed on 3rd November 2008], Seattle, WA, 2008.

[19] Microsoft Robotics Developers Studio team, “Service Tutorial 1 (C#)-Creating a Service”, Microsoft Robotics Developer Studio 2008 July Community Technical Preview (CTP) software documentation, Seattle, WA, 2008.

[20] Microsoft Robotics Developers Studio team, “Interop and Device Samples Overview-XInput Controller Sample”, Microsoft Robotics Developer Studio 2008 July Community Technical Preview (CTP) software documentation, 2008.

[21] MRDS developer and support team, “Microsoft Robotics – Simulation”, http://forums.microsoft.com/msdn/default.aspx?ForumGroupID=383&SiteID=1 [Accessed 5th November 2008], ©Microsoft Corporation, Seattle, WA, 2008.

[22] M.Shastri and A.Aggarwal, “A Welding Simulator Based on a Three Dimensional Coordinates and Orientation Measuring System”;ICRA, Florida, 2007.

[23] N. Gravenstein ,"Temperature BioPhysics and Football Pads", NFL Equipment Managers conference, Orlando, FL, March 7, 2005.

[24] P. Fishwick “CAP 5805: Computer Simulation Concepts”, Department of Computer and Information Sciences, University of Florida, 2008. ch 3.

[25] P. Fishwick, P. Luker ," Simulation Model Design and Execution: Building Digital Worlds", Springer-Verlag, 1991, pp 34-36

[26] P. Vlaovic, E. Sargent, J. Boker, "Immediate impact of an intensive one-week laparoscopy training program on laparoscopic skills among postgraduate urologists". JSLS 12 (1): 1–8. PMID 18402731, May 2006. Retrieved on 2008-08-26.

[27] R. MacNeill, S. Kirkpatrick, "Vehicle Postmortem and Data Analysis of a Passenger Rail Car Collision Test," Proc. JRC2002, The 2002 ASME/IEEE Joint Rail Conference, Washington D.C., April 23-25, 2002.

[28] R. Satava, "Medical virtual reality: The current status of the future," in Proc. 4th Conf. medicine Meets Virtual Reality (MMVR IV), San Diego, CA, 1996, pp. 100–106.

[29] S. Delp and J. Loan, "A graphics-based software system to develop and analyze models of muskuloskeletal structures," J. Comput. Biol. Med., vol. 25, no. 1, pp. 21–34, 1995.

[30] University of Pennsylvania, Lehigh University, Lockheed Martin Advanced Technology Laboratories, "The Ben Franklin Racing Team 2006 DARPA Grand Challenge Entry," Journal of Field Robotics, Vol. 23, No. 8,Aug. 2006.

[31] V. Karthik, V. John, M. Mary-Ann, "Designing optical waveguides for clinical diagnostics in layered tissues : Monte Carlo simulations", IEEE Lasers and Electro-Optics Society 2004 Annual Meeting, Rio Grande, Puerto Rico, 7-11 November 2004.

BIOGRAPHICAL SKETCH

Milind Shastri is a graduate student at the Department of Mechanical Engineering at the University of Florida, Gainesville Florida. His research focus is in the field of robotics, simulation and design. The author received a bachelor's degree in mechanical engineering from the Indian Institute of Technology, Delhi, in 2006. He started and worked as the Director, Robotics Engineer of a robotics startup company called ErockIT Technologies Private Limited in India.