8-2012

An AUV Simulator for Incorporating Physical Feedback

James Brawn Jr.

AN AUV SIMULATOR FOR INCORPORATING PHYSICAL FEEDBACK

By

James Brawn, Jr.

B.S., University of Maine, 2009

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Science)

The Graduate School

The University of Maine

August 2012

Advisory Committee:

Roy Turner, Associate Professor of Computer Science, Advisor

Phillip Dickens, Associate Professor of Computer Science

James Fastook, Professor of Computer Science

Larry Latour, Associate Professor of Computer Science

THESIS

ACCEPTANCE STATEMENT

On behalf of the Graduate Committee for James Brawn, Jr., I affirm that this manuscript is the final and accepted thesis. Signatures of all committee members are on file with the Graduate School at the University of Maine, 42 Stodder Hall, Orono, Maine.

Submitted for graduation in August 2012

Roy Turner, Associate Professor of Computer Science (Date)

LIBRARY RIGHTS STATEMENT

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at The University of Maine, I agree that the Library shall make it freely available for inspection. I further agree that permission for "fair use" copying of this thesis for scholarly purposes may be granted by the Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

James Brawn, Jr. (Date)

AN AUV SIMULATOR FOR INCORPORATING PHYSICAL FEEDBACK

By James Brawn, Jr.

Thesis Advisor: Dr. Roy Turner

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Science)
August 2012

Traditional physics simulators use mathematical models to represent a realistic environment. Natural processes, however, are difficult to mimic accurately. We present a simulator that has the capability to alter its model based on actual physical measurements.

The simulator runs as a server to which remote clients can connect and assume control of entities within the virtual environment. The simulator then sends position updates to clients according to its model. Clients have the option of then correcting the data in these updates, sending feedback to the server. The server adjusts its model to accord with the corrections, allowing for a more realistic model.

Our simulator is general, allowing the user a wide range of customization. With its flexible system of virtual object representation, users can create their own arbitrarily rich virtual environments that include rigid bodies, magnetic fields, and radio waves. The system is designed to be extensible, which also allows the user to customize how the simulator processes its model. Default algorithms for calculating

a time step and detecting collisions are provided, but are can easily be replaced by a user's own implementation.

DEDICATION

To my parents, without whom I never would have taken this path.

ACKNOWLEDGEMENTS

This thesis never would have been completed were it not for the help, advice, and support from mentors, colleagues, friends and family.

I give tremendous thanks to my advisor, Dr. Roy Turner, whose constant help, support and guidance made every part of this thesis possible. He was never too busy to assist with any critical issues I encountered, always available to discuss designs and goals, and always able to put me back on track whenever I felt disorganized.

I would like to thank Dr. James Fastook for taking the time for several impromptu discussions about simulating rigid bodies, which inspired design choices throughout the project.

I would like to thank Michael Brady Butler for his work with his version of the visualization tool and for his suggestions.

Finally, I would like to thank Rahzell and Zahra, for reminding me that it's okay to enjoy the simpler things in life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
INTRODUCTION

In physics simulation, mathematical models are used to represent a virtual environment. Any future state of the environment is completely determined from applying mathematical transformations to some previous state. Often, simulations are used to predict what would happen in real-life scenarios. In the physical world, the future states of environments cannot always be cleanly predicted, as various types of noise affect simple predictions. For this reason, it is essential to create a noise model in order to produce realistic simulations.

Our simulator brings the physical environment more directly into the virtual environment's state. Rather than using mathematics to model noise, our simulator allows physical agents to connect to it and provide actual data from measurements taken in the physical realm. As an example, consider a virtual environment with a virtual agent V. This agent has a physical counterpart P that has connected to the simulator and can use sensors to locate itself within the physical environment. The simulator will use its model to predict the next future state. Assume that it decides that V should move forward 1 m. It will alter its virtual model accordingly and send a message to P informing it of this change. Let us assume that P then attempts to move forward 1 m, but encounters some uneven terrain which causes its movement to be slightly irregular. Instead of moving ahead 1 m, its sensors tell it that it moved ahead 0.97 m. P sends this information back to the simulator, and the simulator then "corrects" itself by replaying the state such that V will end up having moved forward 0.97 m as well.

Though the original intent for the project was to be used as a simulator for autonomous underwater vehicles (AUVs), it grew into a general physics simulator

over the course of the implementation. It simulates a 3D environment in which objects can interact. Objects within the world are called *entities*, and their definition is very general, so as to admit a variety of possible object representations. Entities can model such diverse things as rigid bodies, radio waves, AUVs, and magnetic fields. The standard type of entity that is typically most applicable is called the *world object*. World objects are implemented as rigid bodies, and so contain algorithms for modeling rigid-body dynamics. Each world object has a position, orientation, volume, mass, linear momentum and angular momentum, and a number of forces that act on it. As rigid bodies, each world object can also collide with and bounce off of other world objects.

Once a virtual environment has been created using these components, the user has several options to interact with the simulator. It can be run as a standalone program, where the user supplies each object's behavior ahead of time and lets the simulation play out. It can also be run in a distributed manner with the simulator acting as a server to which remote hosts, or clients, can connect. The network interface is general, not programming language-specific, and its communication medium is human-readable, which yields an easy-to-understand and flexible means of communication. This is achieved by encoding all messages between server and client in AUVSML, an XML-based language that we have developed for the simulator. This architecture is what allows physical agents to easily communicate with the server.

A flexible and language-agnostic network interface is one area in which one of our major goals, extensibility, is apparent.[1] We sought to create a tool that allows the user a wide range of customization. Rather than producing a simulator suited for a specific task, we have attempted to produce a simulator that can be configured, or for more advanced use cases, extended, for whatever task the user needs. We

[1]See Chapter 3 for a discussion of extensibility.

chose to implement the simulator in the Java programming language, which runs on the Java Virtual Machine (JVM). Because it runs on the JVM, the user also has the option of seamlessly extending the simulator with other JVM languages such as Jython or Clojure.[2]

This theme of extensibility permeates the simulator's architecture. The critical algorithms that determine how the simulator calculates the next state of the world are written to general interfaces, the underlying implementations of which can be substituted for any alternative implementation that the user desires. The representation of entities is similarly defined using interfaces, allowing the creation of new and complex types that can either provide alternate implementations, new implementations, or combinations of existing implementations as definitions. New AUVSML messages can be easily introduced into the system, giving the user full discretion as to how much or how little control a remote host has over the virtual world. This also permits any new functionality that the user introduces to be easily used remotely by defining new AUVSML messages.

Another place where extensibility is evident is in the implementation of the *Corrective Feedback System* (CFS). The CFS is that part of the software that accepts data from physical sensors, which is called *feedback*, and decides how to process them using *feedback processors*. There are many different ways to use CFS, and so the system's design allows for this by permitting alternative implementations. One implementation that is discussed in this thesis gives clients an interval of time during which to send feedback messages regarding movement.[3] After it gathers these messages, it compares them with its virtual models, rewinds time in the virtual world to before the state change, and alters the forces acting on each world object such that

[2]This depends on the language, but for most popular ones, extending Java code is typically seamless.

[3]See Section 7.2.

when time is moved forward again, each world object will end up in the same position as its physical counterpart. Many other possibilities exist, as CFS allows users to define their own physical feedback data types and their own feedback processors.

Given the nature of extensibility, this project is ripe for future work. Custom clients, simulation rules, entities, feedback processors, and communication protocols can all be easily created given the architecture of the software. Given its distributed nature, it is easy to build complementary software. For instance, visualization tools, which provide representations of the simulated world that are easier for humans to perceive and understand, can easily interoperate with the simulator. This is achieved by having the tool connect to the simulator as a client. It sends a command to the server indicating that it wants to receive updates on every entity within the world. All data necessary to construct a visual model, including entity positions and orientations, is then sent to the visualization tool.

The project has evolved greatly over the course of its life cycle. Several parts of the system have undergone reimplementations both small- and large-scale. The approaches that are currently used as well as some that have been scrapped are documented here. In addition, there are several areas in which the software could be improved or expanded upon. Ultimately, we have arrived at a system that fulfills the project's goals and is ripe for both use and extension.

Chapter 2
RELATED WORK

This project was created to address a need for extensible simulation software that integrated well with robots for the Maine Software Agents and Artificial Intelligence Laboratory (MaineSAIL).[1] It ties together many known solutions to various problems of simulation, such as time step advancement and collision detection, within an extensible architecture. As a result, existing work was adapted to fit within the architecture that provided a useful solution to MaineSAIL's needs. This chapter gives a summary of related work that has contributed the most to the development of our simulator.

2.1 AUV Simulation

An outstanding example of autonomous underwater vehicle (AUV) simulation is the work of Donald Brutzman (Brutzman, 1994). Brutzman's work tackles many of the real-world difficulties associated with AUV deployment.

> The primary difficulty facing AUV developers is a challenging physical environment: an operating AUV is inaccessible, remote, and unattended. It is subjected to extremes of pressure, temperature, corrosion. Communications are intermittent or nonexistent. Sonar sensing is physically slower and very much different from vision. Vehicle deployment, operation and recovery are time-consuming and expensive. Vehicle physical dynamic control is very challenging. There are six spatial degrees of freedom (three dimensions each for position and rotation), not all physical control issues are solved, and there may be an unpredictable influence by

[1] http://mainesail.umcs.maine.edu/

> ocean currents. Propulsion is costly, slow and limited. A typical vehicle only has a few hours endurance. (Brutzman, 1994)

While Brutzman's work seeks to solve these issues in regards to simulation, our model abstracts many of these issues away, yielding a more general platform.

One area in which our implementation is an improvement over Brutzman's work is the representation of rigid body orientation. Brutzman's model uses six spatial degrees of freedom to represent a rigid body's posture: x, y, and z encode position along the x-, y-, and z-axes, respectively, and ϕ, θ, and ψ encode orientation about the x-, y-, and z-axes, respectively, which are referred to as Euler angles (Brutzman, 1994).

Quaternions, which are an extension of complex numbers, can be used instead to represent orientation. They have several advantages over Euler angles; for instance, over time, updating the orientation will produce numerical error that would cause a graphical skewing effect (Witkin & Baraff, 1997). Quaternions overcome this problem, as discussed in Section 5.2.

Brutzman's work was published in 1994, and so utilized the technology of that time. Though our project has many of the same features, we had the opportunity to implement them with the modern and more widespread technologies available in 2012. This stands out the most in regard to networking.

Two kinds of messages are used in Brutzman's architecture: telemetry vectors, which encode all vehicle state variables, and free-format commands, which begin with a keyword identifying the type of command and continue with type-specific command data. Both are represented as strings and can be passed between various parts of the system. The use of strings ensures that all communication is readable by humans, making debugging easier (Brutzman, 1994).

Our network communications system is very similar in design to Brutzman's approach. We use strings to pass data between the server and any remote clients, and each message has a type that defines type-specific data, not unlike the free-format commands in Brutzman's system. Our strings are formatted in AUVSML, which is based on XML, a now-ubiquitous standard for representing structured and semi-structured data that was originally proposed in 1996, two years after Brutzman's publication (Bray & Sperberg-McQueen, 1996).

2.2 Rigid-body Dynamics

The physics algorithms which model all aspects of rigid-body dynamics in our simulator were adapted from Witkin and Baraff's lecture notes (1997). The notes detail an approach to modeling rigid-body dynamics in a simulation. The first set of notes cover how objects are represented. Position, orientation, linear momentum, and angular momentum are used to describe an object. Ordinary differential equations (ODEs) are used to advance from one state to the next. The notes provide complete mathematical descriptions of these methods, and even provide snippets of a pseudo-implementation in C. The challenge in utilizing these materials was finding a way to adapt the structure of rigid-body computations to the structure of the simulator's software. For instance, where the notes simply show how to express time step advancement as an ODE, our task was to model the ODE as a Java object that interacted both with our existing design for world objects and with a third-party library (described in Section 2.3) that provided ODE solvers.

The second set of notes covers collision detection and response. The treatment remains general by discussing how to detect collisions between polygons with any number of vertices. Since our system approximates world objects as spheres, most

of this did not apply.[2] One concept from these notes that was the most useful to us was the equation that calculated the impulse caused by a collision. The equation provides the basis for the behavior of our rigid bodies that allows them to bounce off of other rigid bodies, and takes linear and rotational changes in movement into account.

2.3 Open Source Tools

Though most of the rigid-body dynamics calculations were straightforward to implement, some parts required special algorithms. The Open Source Physics (OSP) library[3] provides many such tools, including ODE solvers, which are utilized by our simulator during time step advancement. Using the accompanying manual (Christian, 2007), we learned not only how to best use the library, but also about some core physics concepts that its software models. Initially, the library was used for more core tasks than it was at the end of the project; some of the functionality it provided has since been replaced by our own implementations. For example, we wrote our own class to represent quaternions for rotation. Currently, the only part of the core system that is leveraged with OSP is the fourth-order Runge-Kutta ODE solver, though future work could see this reimplemented so as to remove the dependency on OSP.

Our default visualization tool, however, benefits heavily from OSP. OSP provides an intuitive 3D graphical display to which 3D objects, such as spheres and boxes, can be added. The user can interact with the window with the mouse cursor to rotate the graph and zoom in or out. The user can also choose between perspective and orthogonal views using the provided menus. This made it much easier to get

[2]Note that although we approximate world objects as spheres, the system is written generally such that many differently-shaped volumes may be used. The default implementation, however, provides only spheres and basic boxes.

[3]http://www.opensourcephysics.org/

a visualization up and running without having to spend time and effort developing 3D graphics software.

Another open source library that the simulator uses is the Apache Commons Math library.[4] Though we represent orientation as quaternions, it is useful to convert the quaternions into Euler angle rotation matrices when rotating vectors. For matrix representation and matrix operations, our simulator defers to the Apache Commons matrix implementation.

2.4 Good Software Development Practices

As mentioned in Section 2.2, one of the challenges we faced was ensuring that the rigid-body dynamics code worked in a manner consistent with our architecture. Designing the architecture itself was also a great challenge, and underwent revisions both major and minor several times over the course of the project. This was necessary to cleanly support new features of the simulator.

Informing these designs were good software engineering practices that we have developed over time. With regard to how these designs best worked with the Java programming language, the book *Effective Java* was used often (Bloch, 2008). *Effective Java* details many techniques and patterns that have been discovered over time to be the best practices in most situations. For example, the builder pattern (Bloch, 2008, p. 11), which provides an intuitive way for users of a class to build an object with many parameters, was used in our SIMULATOR and WORLD classes, which represent the simulator and the virtual world, respectively.

[4] http://commons.apache.org/math/

Chapter 3
PROBLEMS AND PROJECT GOALS

Building a simulator gives rise to many interesting problems. When designing solutions to these problems, having a set of general guidelines helped tremendously. We present several issues on which we focused, including advancing an adaptive time step, dealing with object collisions, and how to incorporate feedback data from remote clients. We also discuss our primary design goal of extensibility, which will allow users of the software to craft simulations to their specifications.

3.1 Advancing the Time Step

Computer simulation requires representing continuous time as discrete chunks referred to as *time steps*. We consider a number of *snapshots* that capture the instantaneous state of the simulated environment; a time step represents the interval between successive snapshots. For example, simulating all events from $t = 0$ s to $t = 1$ s might involve only simulating the state at $t = 0$ s, jumping ahead to $t = .2$ s, and then to $t = .4$ s, $t = .6$ s, $t = .8$ s, and $t = 1$ s. The length of the time step between each snapshot is called the *time step delta* (Δt). To run a simulation through an interval of n seconds with a fixed Δt, the simulator processes $\frac{n}{\Delta t}$ time steps.

There are a number of issues that arise with this approach, since we are approximating what is happening between each snapshot. If Δt is large, we may end up not simulating an event that would have occurred. For our purposes, we typically consider an event to be a collision between two objects. For example, say that the current time is t, an event will occur at $t + t_\epsilon$, and $\Delta t > t_\epsilon$. Since we are only examining the state at times t and $t + \Delta t$, we will miss the moment at which the

event occurs. Since the event does not occur, any future events that depended on it will now be different. If Δt is small, however, we may end up spending a lot of processing time on unnecessary computation, since we may be processing many time steps during which no events occur. Compounding this is the fact that despite even a very small time step delta, we may still miss the precise moment in which an event occurs. A compromise must be struck between the two. A large time step will be sufficient in some situations, whereas in others, smaller time steps are necessary.

Using *adaptive time steps* allows the simulation to predict how big Δt should be based on the current state of the simulation. We would like to use a fixed Δt as a default, but decrease it when we suspect that an event will occur at some time t_E such that $t < t_E < t + \Delta t$. Here, we would set $\Delta t = t_E - t$ so that time is advanced to t_E and we can process the event. How to choose the adaptive time step will require making predictions as to when the next event will occur. We must also be accurate in choosing the next time step delta so that we can properly model the effects of a collision.

3.2 Collision Detection and Response

In a simulation, virtual objects are, at some level, represented by where they are located and how much space they take up in the world. These are just numbers; there is initially no notion of what happens if two objects overlap. To simulate collisions between objects as in physical reality requires algorithms that perform *collision detection* and *collision response*.

A collision detection algorithm will analyze two objects and determine whether or not they overlap. Because a time step advancement from time t to time $t + \Delta t$ could miss an overlap that occurs at time $t + t_\epsilon$ where $t_\epsilon < \Delta t$, we also need a means of finding t_ϵ.

A collision response algorithm operates on objects that have collided and determines how to change their state. A typical collision response behavior is to simply make each object bounce off of the other, as in an elastic collision. We should be able to define any response behavior. For instance, we might want to simulate a mine object that explodes whenever another object collides with it. Collision responses should operate per-object. In our example, the unwitting object would simply perform its normal bounce-off response calculations, even though it hit a mine. The mine would be solely responsible for producing an explosion and a shockwave. The shockwave, being an object itself, would then have *its* collision response damage the world object.

3.3 Processing Physical Feedback

Traditionally, simulators simulate an environment and are in complete control of that environment. How objects move, how interactions between objects and the environment occur, and how the environment changes are all determined by the simulator's calculations. Although computers can create strikingly realistic worlds and compute incredibly useful data that have real-world applications in this way, there are some drawbacks to this approach.

Movement within the physical world is never exact and, especially with imprecise tools, is carried out with a moderate degree of noise. For example, if a robot is instructed to move 10 m forward, it may actually end up moving 9.98 m or 10.002 m forward. This can occur for a number of reasons. In an outside environment, factors such as terrain and weather must be taken into account. For a roving robot, small pebbles could disrupt the precision of its movement. In an inside, controlled environment, many of these factors are eliminated. Noise can still creep in due to, for instance, how well the motors are working and how precise the robot can execute

a turn. All of these factors should be accounted for in a realistic simulation by its choice of noise model.

Modeling environmental noise mathematically, though it may be reasonably accurate in certain situations, cannot always take into account all sources of noise in the real world. Rather than simulating it, the simulator should take actual data from the physical world and incorporate it into its virtual world. The simulator will no longer be completely responsible for determining the state of the virtual world. The user should also be in control of how the physical data alters the virtual world, as this capability would allow numerous useful possibilities. For instance, one might use the data to simply "correct" the virtual objects' positions, or one could create an algorithm that would build a dynamic mathematical model as a function of actual data from the environment.

3.4 Server/Client Communication

The simulator should act as a server to which clients may connect. Each client will be able to send commands to manipulate objects within the simulation as well as gather data from the virtual world. Some clients will simply control one of the virtual agents within the simulation; others may register to receive notifications of all in-world events, which can be used to create a visualization; still others may have the ability to alter important attributes of the simulator itself while it is running.

To allow for clients to send physical data back to the server, some messages will be classified as *feedback*. Feedback messages will have the ability to alter the simulation.

The simulator's network interface should be easy to access and manipulate from any programming environment. It should be easy to write a client in any language.

3.5 Extensible Software

Many challenges must be dealt with when creating a physics simulator. How to choose a time step, how to detect collisions, and how to handle physical feedback are a few examples. Each has several practical solutions, and no one solution is strictly better than another in all situations; trade-offs typically exist. Choosing the proper solution for a given situation depends on many factors. For this reason, the simulator should not lock the user into any particular solutions. It should provide default solutions, but also provide the framework for allowing the user to create and easily integrate his/her own solutions.

Though the code may be designed to be easily extended, this does not necessarily mean that it is easy to understand. If a user cannot understand how a system works, then its extensibility is hampered, despite how well it may be designed in a technical sense. This can be alleviated with good documentation. Unfortunately, there are a number of interesting and useful academic software tools in existence that have a paucity of good documentation. The AUV simulator project should be well-documented for future use by third parties.

Chapter 4
SYSTEM DESIGN

The system was designed from the top down as three major components, each with its own responsibilities. These components are the *virtual world* (sometimes referred to as the *environment*), the *network server*, and the *simulator*. Figure 4.1 shows how these components interact.

The world maintains the current state of the simulated environment, which includes the current world time and the state of all entities in the world. Since most of the interesting topics revolve around world objects, which are rigid bodies, we will often talk mostly about world objects rather than entities. The world is also charged with handling collision detection. The network server is the interface between the system and the outside world. Remote clients connect to and communicate with the server. A client could be a visualization tool, a robot with sensors and a program that dictates how it should communicate, or even a human sitting at a terminal entering commands and responding to server messages. The server is responsible for parsing messages received from clients and passing commands to other parts of the system for execution. It is also responsible for sending updates to the clients regarding the state of the world, such as when world objects move. The simulator ties the world and the server together and is charged with handling the behavior of how the Corrective Feedback System (CFS) is integrated with the world.

4.1 The Virtual World

The virtual world maintains the state of the simulated environment. It has two major concerns: maintaining the state of each world object and deciding how these states will change when the time step is advanced.

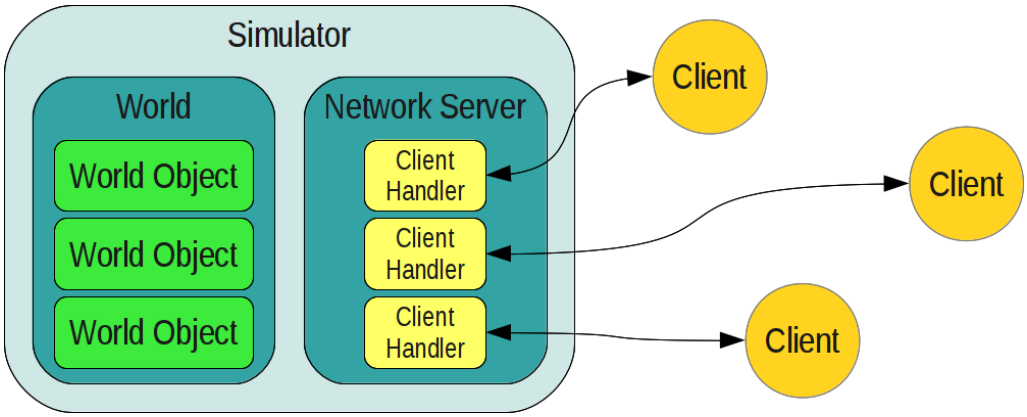

Figure 4.1. A diagram of how the three major parts of the system interact. Depicted is a world with three world objects and a network server with three connected clients.

Each world object's state is compromised of its mass, its volume, a set of all of the forces that act upon it, and its *movement state*, which encapsulates the object's position, orientation, linear momentum, and angular momentum.[1] Forces can operate on any point within the volume of the object. Depending on this point, the force could induce a change in linear momentum (which will alter its position) and/or a change in angular momentum (which will alter its orientation).

Forces are represented as vectors and are classified as either *world-relative* or *body-relative*. World-relative vectors are given in coordinates relative to the world. This allows easy modeling of forces such as gravity, which push on the object irrespective of its orientation. Body-relative vectors are given in coordinates relative to the current orientation of the object. This allows easy modeling of forces such as the thrust of engines, whose world-relative direction changes along with the orientation of the object.

[1]We chose to store momenta rather than velocities for reasons given in Section 5.2.

4.2 Network Server

The network server is in charge of accepting and maintaining connections with remote clients. Messages of different types will be sent to and received from the clients. Clients typically send *command messages*, or *commands*, which are routed to a particular part of system for interpretation and execution. The type of a message is distinguished by which part of the system is meant to receive it. The following is a list of message types.

- Client commands involve the server/client connection, such as disconnecting.
- World object commands alter the state of a world object.
- Simulator commands alter the state of the simulator (such as pausing and resuming the simulation).
- Feedback messages contain physical feedback data that the simulator can use to alter the world.

The server can send messages back to the clients. These are typically updates regarding the state of the world. Clients can register for particular updates; for instance, a client can register to only receive updates about a certain world object.

The server is multithreaded to easily allow multiple concurrent connections. It shields the rest of the system from concurrency concerns by abstracting away all the details of multithreading.

All of the server's incoming and outgoing data is formatted in a language we created for this project, the *AUV Simulator Markup Language (AUVSML)*. AUVSML is an XML-based language that encodes messages in a human-readable format. We describe AUVSML in more detail in Section 7.1. The server is responsible for performing all encoding and decoding for AUVSML data.

The following chapters provide more details about the world and the network server. Chapter 5 discusses the intricacies of the physics engine, Chapter 6 explains how objects are represented in the world, and Chapter 7 covers client/server communication and the Corrective Feedback System (CFS), which allows physical feedback to be incorporated into the simulation.

Chapter 5
PHYSICS ENGINE

The world is responsible for keeping track of world objects and maintaining the state of the environment. As such, it contains all necessary logic for simulating physics and representing objects within the world. This chapter discusses the physics engine, which implements a simulation of rigid-body dynamics, and the algorithms used to project the world's state forward.

5.1 Time Step Advancement and Collision Detection

As discussed in Chapter 3, choosing an adaptive time step depends heavily on collision detection in our model. The algorithms we propose in this section can be used together to create a conservative model of the world. That is, it is designed never to miss any collisions, which can sometimes put it at odds with other concerns, such as efficiency.[1] Our process is composed of four steps.

1. **Choose a preliminary time step.** Choose a new time step delta, DT, based on the current state of the world that is small enough such that it shouldn't, on average, miss any collisions. This will be a preliminary value that will most likely be refined in the following steps.

2. **Partition the world into collision sets.** Partition the world into *collision sets*, or sets that contain objects that are close enough to one another, relative to all other objects, that they may collide between the current time t and $t + DT$. This partitioning is done with the intent of reducing the amount of collision checks that we must perform.

[1] We discuss these concerns further in Section 5.1.4 and Chapter 8.

3. **Find the earliest collision.** For each collision set $s \in S$, where S is the set of all collision sets, find the time t_s of the earliest collision between members of the set. Choose the smallest of these times $t_S = \min_{s \in S}\{t_s\}$. If no collisions occur at all, let $t_S = DT$.

4. **Advance the time step.** Let $\Delta t = t_S$ and advance the world time by Δt.

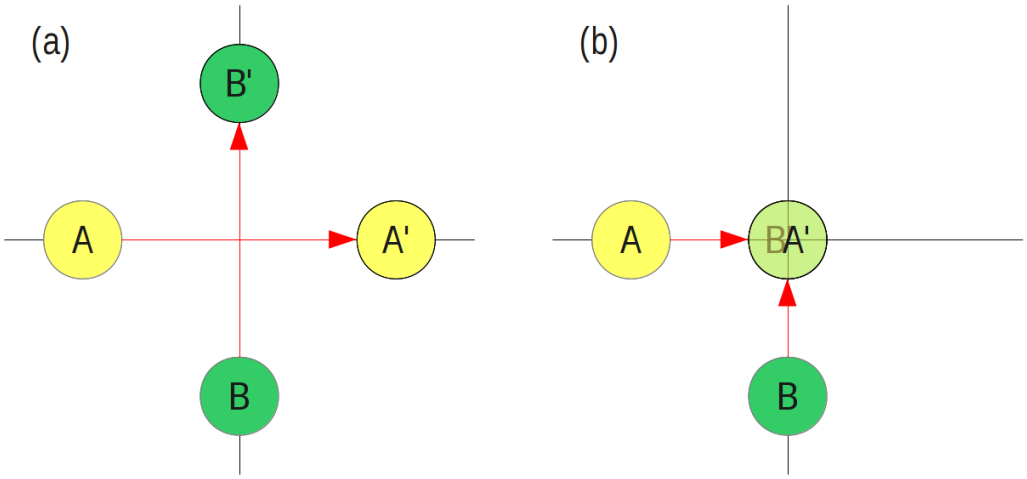

Figure 5.1. (a) An example of missing a collision; $\Delta t = 1$. (b) By reducing the time step to $\Delta t = 0.5$, we see that the objects actually collided at one point.

Time must be advanced by a Δt that is chosen adaptively at each snapshot to ensure that collisions between objects are not missed. Consider the following example involving two objects A and B, which is illustrated in Figure 5.1. Let $\mathbf{S_A}(t)$ and $\mathbf{S_B}(t)$ give the positions of A and B at time t, respectively, and let $\mathbf{v_A}$ and $\mathbf{v_B}$ give the velocity vectors, relative to the world, of A and B, respectively (their velocities are constant). Assume that $\mathbf{S_A}(t_0) = \langle -1, 0\rangle$, $\mathbf{S_B}(t_0) = \langle 0, -1\rangle$, and that $\mathbf{v_A} = \langle 2, 0\rangle$ and $\mathbf{v_B} = \langle 0, 2\rangle$. If we are to assume an advancement by unit time, that is, $\Delta t = 1$, then we have $\mathbf{S_A}(t + \Delta t) = \langle 1, 0\rangle$ and $\mathbf{S_B}(t + \Delta t) = \langle 0, 1\rangle$. From these results, we can find no collision. If $\Delta t = .5$, however, then we have $\mathbf{S_A}(t + \Delta t) = \langle 0, 0\rangle$ and $\mathbf{S_B}(t + \Delta t) = \langle 0, 0\rangle$, which is clearly a collision.

Our basic strategy for time step advancement is as follows. Let DT be a fixed time step delta. To calculate Δt given the state of the world at time t_W, we use the following formula.

$$\Delta t(t_W) = \begin{cases} t_C - t_W, & \text{If a collision will occur at time } t_C, \\ DT, & \text{otherwise} \end{cases}$$

This will then rely on finding t_C given a state of the world at time t_W.

Finding t_C accurately is not a trivial problem. The problem can be defined as follows. Given two objects A and B, their position functions $\mathbf{S_A}(t)$ and $\mathbf{S_B}(t)$, and their radii R_A and R_B, find the earliest time t_C such that $|\mathbf{S_A}(t_C) - \mathbf{S_B}(t_C)| = R_A + R_B$. Because our model includes the orientation of objects and multiple body force vectors that change every time the orientation changes, an analytical solution is impractical. We therefore proceed by approximating where the collision will occur.

5.1.1 Choosing an Adaptive Time Step

In the example above, objects A and B's paths crossed one another, but the time step was too big to capture any moments in which they overlapped. One way to address this problem is to alter the time step such that each object cannot move a distance larger than its own radius.[2]

The idea for this preliminary step is to ensure that objects do not go far enough in the given time step so as to miss any collisions. If the time step is reduced such that each object will not move a distance greater than its radius, then it is far less likely that we will miss collisions. (Note that for a world in which the difference in size between objects is very large, we may still miss some collisions.)

The problem is defined formally as follows. Assume the current time is t. Given a set U of n objects, find a time $t + t_R$ such that $(\forall obj \in U)(|\mathbf{S_{obj}}(t + t_R) - \mathbf{S_{obj}}(t)| \leq$

[2]The idea for this approach came from a personal communication with Professor James Fastook.

R_{obj}), where $\mathbf{S_{obj}}(t)$ gives the position of object obj at time t and R_{obj} gives the radius of obj.

We will approximate $t + t_R$ with the following algorithm, which we will refer to as a *radius-distance restricter*, as it restricts the amount of distance objects can travel based on their radii.

```
1: function RadiusDistanceRestricter(WorldObjects, DT)
2:     t_R ← DT
3:     for all object ∈ WorldObjects do
4:         repeat
5:             oldPos ← object.getPosition()
6:             newPos ← object.projectNewPosition(t_R)
7:             distance ← |newPos − oldPos|
8:             if distance > object.radius then
9:                 t_R ← t_R/2
10:            end if
11:        until distance ≤ object.radius
12:    end for
13:    return t_R
14: end function
```

We proceed with a theoretical runtime analysis. The outer loop iterates over each world object, contributing $\Theta(n)$ to the runtime, where n is the number of world objects. The inner loop then executes, but for convenience, we will first examine the runtime of its contents. The call to getPosition() runs in constant time, as this information is already stored as part of object's state. The call to projectNewPosition() depends on how the physics engine implements rigid-body movement. As we discuss in Section 5.2, an object's new position is ultimately

determined by the fourth-order Runge-Kutta ODE solver, which is a constant time calculation. The distance calculation in Line 7 is a constant time operation as well. Next, we test to see whether the distance traveled by the object is greater than its radius. If not, the inner loop terminates, leaving us with a $\Theta(1)$ runtime for this iteration of the outer loop.

In the best case, this would hold for all world objects. Thus, the algorithm has a lower bound of $\Omega(n)$. If the distance traveled by the object is greater than its radius, however, then the inner loop is repeated. How many times this loop is repeated depends upon both the distance traveled by the object within the time step (which is dependent on how fast it is moving) and its radius. Note, however, that the time step is adjusted for all future iterations as well. That means that if two objects were moving distances greater than their radii, the iteration for the first object may reduce the time step enough so that the second object no longer moves a distance greater than its radius when its iteration is processed.

For this reason, the worst case would be hard to achieve. In the worst case, we can imagine the first object moving far enough to trigger the time step reduction several times. The second object would then have to be moving much farther away such that this time step reduction would still not be sufficient. The third object would have to be even farther, and so forth. This worst case could be described as contributing a $O(n \cdot m)$ runtime complexity, where m is the maximum number of time step reductions required for an iteration.

5.1.2 Partitioning the World into Collision Sets

Naïve collision detection is a $\Theta(n^2)$ algorithm: given n objects, each object would have to be checked against at most $n - 1$ other objects. We can prune unnecessary computations, however, by only checking for collisions between objects that are relatively close to one another. This is done by partitioning the world

into *collision sets*. We define a collision set as a set of objects from the world that are reasonably close to one another. For example, if there are 5 objects in a 2D world with coordinates $(2, 1)$, $(2, 3)$, $(47, -14)$, $(48, -12)$, and $(49, -11)$, a good partitioning algorithm might place the first two into a collision set and the last three into a collision set. Given an object A that belongs to a collision set S, we can run collision checks between A and $\forall s \in S$ rather than between A and all other objects in the world. Note that it is possible for an object to belong to more than one collision set.

The area considered to be taken up by each world object when partitioning occurs is a concern. Let us first define the validity of a partitioning of collision sets. Let S be a *full set* of collision sets, or the set of all collision sets after a partitioning operation. That is, for all $obj \in WO$, where WO is the set of all world objects, obj exists in some set that is a member of S. Formally, we have that $(\forall obj \in WO)(\exists C \in S)(\exists member \in C)(obj = member)$. A full set of collision sets S is valid if and only if for each collision c that occurs within the interval $(t, t + \Delta t)$, the two objects colliding—call them obj_1 and obj_2—exist in the same collision set $C \in S$. Formally, we have that

$$\begin{aligned} &valid(S) \Leftrightarrow \\ &(\forall obj_1, obj_2 \in WO)(obj_1 \neq obj_2 \wedge collides(obj_1, obj_2, t, t + \Delta t) \Rightarrow \\ &(\exists C \in S)(obj_1, obj_2 \in C)), \end{aligned} \tag{5.1}$$

where $collides(o_1, o_2, t_1, t_2)$ is a predicate that returns true when the areas taken up by objects o_1 and o_2 overlap at some point in the interval (t_1, t_2).

If we were to consider only the area taken up by world objects at any given time t, the collision sets are only valid for time t and not necessarily valid at any time $t' \neq t$. Because we are interested in detecting collisions over the interval $(t, t + \Delta t)$ rather than at an instantaneous moment, this will not suffice. We must then partition

based on the area taken up by world objects over the interval $(t, t + \Delta t)$. We use a simple method to achieve this.

Given an interval $(t, t + \Delta t)$, we construct a bounding sphere $B^i_{(t,t+\Delta t)}$ for object i that encompasses the area taken up by the object's bounding sphere at times t and $t + \Delta t$. For small Δt, B^i encompasses all space taken up by the object at all points during the interval $(t, t + \Delta t)$, as well as some extra space that the object never occupies. The larger Δt is, the more empty space B^i will encompass. An example of a situation in which a large Δt would yield an inaccurate B^i is shown in Figure 5.2. Since our simulation will be using small Δt, and the radius-distance restricter may make Δt smaller still, this approximation of the occupied area will suffice.

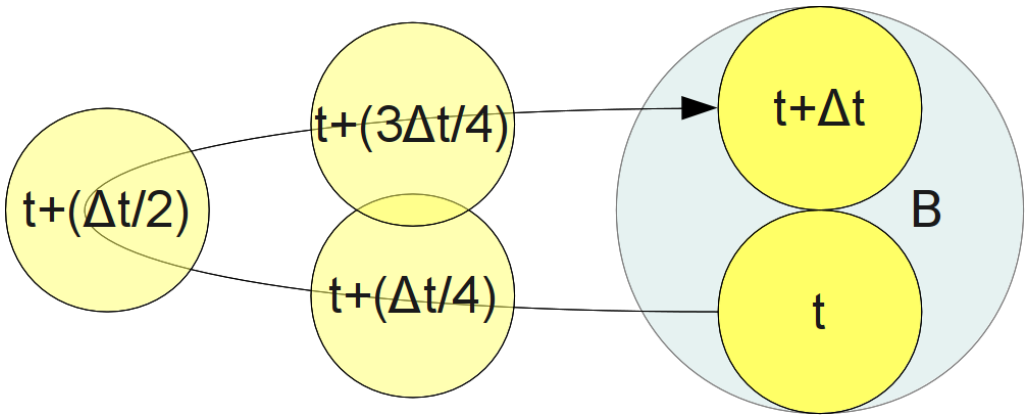

Figure 5.2. If the object travels along a path depicted by the arrow from t to $t+\Delta t$, the bounding sphere B will not encompass all space taken up by the object at all points during the interval. For small Δt and objects that don't move incredibly fast, this situation will most likely not occur.

We use the octree algorithm to partition the world into collision sets (Ganovelli et al., 2000). An octree works as follows. Each node of the tree will encompass a cube of space and will contain whatever objects happen to be within that cube over the given interval. Figure 5.3 illustrates this with a quadtree, a 2D equivalent of an octree that is easier to depict. Assume that the world is contained within a

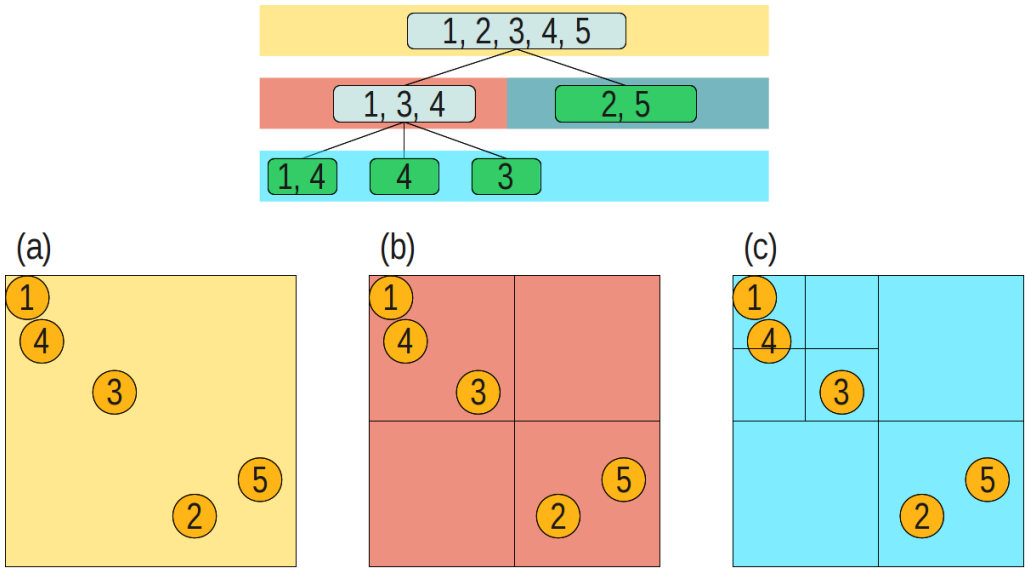

Figure 5.3. A diagram of a quadtree, the 2D equivalent of an octree, with $N = 2$. The tree is shown above, and the world is shown below (a) before any splitting, (b) after the first split, and (c) after the second split. The color of each world representation matches the color in the tree. (a) The algorithm begins with all five objects in the same collision set. (b) The set is split once, yielding a set with $1, 3, 4$ and one with $2, 5$. (c) The $2, 5$ set requires no further splitting, and so is retained through the second split. $1, 4$ are placed in a set, but 4 is also contained in its own set. 3 also gets placed in its own set.

cube volume. Let D be the maximum depth of the tree and let N be the maximum number of objects per node (these will be chosen by the user depending on what values are best for his/her simulation). The root node encompasses the entire world, and so contains all objects. At the beginning of the algorithm, the current node C is the root node. The algorithm proceeds as follows.

1: $leaves \leftarrow [rootNode]$ ▷ To start, $rootNode$ is the only leaf
2: $d \leftarrow 0$
3: **while** $d \leq D$ **do**
4: $\quad d \leftarrow d + 1$
5: $\quad newLeaves \leftarrow []$
6: $\quad$ **for all** $leaf \in leaves$ **do**

```
7:          if leaf.numOfObjects > N then
8:              newLeaves ← newLeaves + Split(leaf)
9:          else
10:             newLeaves ← newLeaves + leaf
11:         end if
12:     end for
13:     leaves ← newLeaves
14: end while
```

Let n be the number of objects in this node. If $n \leq N$, then no further work is done. If $n > N$, then the current node is split. This is done by dividing the cube into 8 cubes of equal size. For each of these cubes, a child node of the current node is created.

SPLIT() is defined as follows.

```
1: function SPLIT(leaf)
2:     newLeaves ← Divide leaf's area into 8 cubic areas of equal size
3:     for all newLeaf ∈ newLeaves do
4:         for all object ∈ leaf.objects() do
5:             if newLeaf's area overlaps with object's then
6:                 newLeaf.addObject(object)
7:             end if
8:         end for
9:     end for
10:    return newLeaves
11: end function
```

It is important to point out that an object can exist in multiple collision sets. There are several cases in which this might occur, but perhaps the simplest to

imagine is the case in which an object is located at the center point of the world. This center point will become a corner for each of the eight subcubes that are created by SPLIT(), and so an object in the center will thus exist partly in each of those subcubes. To ensure that it misses no collisions between objects in any of those cubes (imagine the case in which there are eight objects, each completely contained in one of the subcubes, all heading for the center), that object will be a member of all eight collision sets.

Since there might be multiple objects at the center, this could become a problem for how many times SPLIT() is run. Imagine the case of $N + 1$ objects located at the center. Since each of the eight subcubes contains $N + 1 > N$ objects, SPLIT() is called on each. Since the objects are contained in the center, the split does nothing but "shrink" the subcubes about the center, and we are still left with eight collision sets each with $N + 1$ objects. To avoid an infinite loop, the depth of the tree, restricted by D, is also used as a stopping condition. Thus, the algorithm would continue to perform these splits until the tree had a depth of D, at which point it would stop. Though we are left with an unhelpful result, we must consider that this is a rare case.

We will analyze the complexity of our octree implementation beginning with SPLIT(). SPLIT() begins by dividing the area of LEAF into 8 cubic areas of equal size. This amounts to a constant number of arithmetic operations which runs in $\Theta(1)$ time. Line 3 begins a loop over this 8-element array, multiplying the cost of the loop's contents by $\Theta(1)$. Line 4 iterates over all objects contained within LEAF, which will contribute at most a factor of $\Theta(n)$ to the loop's contents, where n is the number of all objects in the world. The IF statement's test and possible call to ADDOBJECT() each run in constant time. Therefore, the overall time complexity of SPLIT() is $\Theta(n)$.

In the octree algorithm proper, we will begin our analysis with the inner loop. This loop iterates over all leaves in the tree, of which there can be at most n, where n is the number of objects, since we may have a node for each object. This contributes $O(n)$ to the runtime. The test within will execute if the number of objects in the current node exceeds N. By conservatively assuming that it does, SPLIT() will be called, contributing $O(n)$. This yields a total of $O(n^2)$ for the inner loop. Proceeding to the outer loop, we see that it iterates D times. Since D is a constant, the outer loop contributes $O(1)$. Overall, the worst case yields $O(n^2)$.

In the best case, objects would be positioned such that at most one object is contained in each node. For instance, in keeping with the quadtree example shown in Figure 5.3, we would have four objects positioned so that after the first split, each node contains only one object, or sixteen objects such that the same case occurs after two split iterations (five splits total). Extending this to octrees, eight objects would require only one split, sixty-four would require only two split iterations (nine splits total), and so forth. This runtime can be summarized as $O(n \cdot 2(\log_8 n))$, where the n factor is the runtime of SPLIT() and the $2(\log_8 n)$ factor is a loose upper bound on the number of splits required. We can drop the constants, yielding $O(n \cdot \log n)$.

An average case analysis here would be quite complex, since many complex factors such as object positions and velocities are involved. We will defer instead to the empirical analysis in Chapter 8.

5.1.3 Finding the Earliest Collision

With the set of all world objects divided into smaller sets, collision detection's normal $O(n^2)$ complexity is mitigated by a decrease in the constant factor. Each collision set is scanned to see if any collisions actually occur given the time step delta. This process proceeds as follows.

1. Using the bounding spheres from above, check whether any of the bounding spheres in any of the collision sets overlap.

2. For each overlap of objects i and j, use bisection to approximate the earliest time t_{ij} at which the two objects collide.

3. Select the earliest collision time $t = \min_{i,j \in WO} t_{ij}$, where WO is the set of all world objects, and where $t_{ij} = \infty$ if i and j do not collide.

Bisection is a method of numerically approximating the inputs of a function that yield a particular output. If two bounding spheres overlap, we proceed with bisection. The process begins by dividing the interval $(t, t + \Delta t)$ into $n + 1$ time indices. That is, we consider the instantaneous times $t, t + (\frac{1}{n})\Delta t, t + (\frac{2}{n})\Delta t, ..., t + (\frac{n-1}{n})\Delta t, t + \Delta t$. In order from earliest to latest, we check each index t_k to see if $overlap(obj_i, obj_j, t_k)$ is true. If none return true, then we assume that no collision will occur over the interval, and that the bounding sphere overlap simply happened over extra space within the spheres. If the objects overlap at a time t_m, then we set $t_{min} \leftarrow t_{m-1}$ and $t_{max} \leftarrow t_m$, where t_{m-1} is the time of the previous snapshot.

```
1: function BISECTION( obj_1, obj_2, t_min, t_max )
2:     if Bisection has iterated past the maximum number of iterations then
3:         return t_max
4:     end if
5:     t_mid ← (t_min + t_max)/2
6:     if overlap(obj_1, obj_2, t_mid) = true then
7:         return Bisection(obj_1, obj_2, t_min, t_mid)
8:     else
9:         return Bisection(obj_1, obj_2, t_mid, t_max)
10:    end if
```

11: **end function**

Since the only stopping condition for BISECTION() is surpassing a fixed number of iterations, and since all operations run in constant time, BISECTION() runs in $\Theta(1)$ time. Note, however, that BISECTION() must be executed for each pair of objects that collide and possibly some pairs that almost collide. The overall runtime, then, would be $O(p)$, where p is the number of pairs considered.

In the project's earlier stages, an analytical approach was used to determine whether two objects collided. It operated as follows. Let obj_1 and obj_2 be two objects within the same collision set that might collide, and let $\mathbf{S_i}(t)$ give the position of obj_i at time t as a vector. Then $d(t) = |\mathbf{S_1}(t) - \mathbf{S_2}(t)|$ gives the distance between the objects at time t. We find the minimums of $d(t)$ using its first and second derivatives, and check to see whether any minimum $d(t_m)$ is such that $d(t_m) \leq (r_1 + r_2)$, where r_i is the radius of obj_i. If so, a collision between the two objects will occur during the given interval.

Though the math was complex, the solution worked. In order to account for orientation and rotational effects, which were added later, the position equation would have to model the effect of body-relative forces as the orientation changed over the interval of consideration. Since this would add more complexity to an analytical approach that was already complex, it was scrapped in favor of the numerical approach.

5.1.4 Advancing the Time Step

At this point, either a collision occurs at time t_c where $t < t_c < t + \Delta t$, or no collision occurs during the interval $(t, t + \Delta t)$. The new time step is then $\delta t = \min\{t_c, t + \Delta t\} - t$. Given δt, the world can advance the simulation. It does this by informing all world objects of the change in time. If collisions occur, it will first tell all world objects that are involved in collisions that a collision has occurred and

will pass each colliding world object some description of the world object it collided with.

Note that these algorithms were chosen to create a conservative simulation. The radius-distance restricter will stop objects from moving too far, and the collision finder will reduce the time step delta whenever a collision is imminent. An advantage of this approach is that it is far less likely to miss any collisions or end up with objects that overlap. A disadvantage is that it can result in real-time performance costs, as more snapshots must be processed whenever collisions are imminent. These effects are analyzed in Chapter 8.

Though this may work well when precision is key to a simulation, it may not be ideal for all circumstances. A design goal of the system was to allow critical pieces of processing to be swapped out for alternate implementations. Because of this, all of the above algorithms can be swapped out for user-written implementations. Thus, if a user wishes to do collision detection that allows overlap, they might remove the radius-distance restricter and alter the earliest collision finder. If the user knows that he/she will never have more than a small number of objects in the simulation, he/she may choose to remove the octree partitioner and just use a naïve $\Theta(n^2)$ collision detection algorithm.

Overall, the algorithms roughly contribute $O(n)$, $O(n \cdot \log n)$, and $O(p)$ for runtimes. Taken together, if we assume that the number of collision pairs cannot exceed $k \cdot n$ where k is some constant factor, then we have a total upper bound of $O(n \cdot \log n)$. In Chapter 8, we examine the actual runtime of these algorithms and how they compare with our theoretical estimates.

5.2 Rigid-body Dynamics

Each world object contains logic necessary to simulate a three-dimensional rigid body that can interact with other world objects. Most of the concepts used in implementing the rigid-body dynamics were given in Witkin and Baraff's SIGGRAPH lecture notes (Witkin & Baraff, 1997). A summary follows.

Each world object's physical state is defined by its position, its orientation, its linear momentum, its angular momentum, its mass, and its collection of force vectors. Position is described by three points $\langle x, y, z \rangle$. Orientation is described using quaternions, rather than Euler angles, which will be justified below. Quaternions are members of $\mathbb{R}^4$ and can be thought of as an extension of complex numbers. Where a complex number is described by $a+bi$, a quaternion has three imaginary components: $a + bi + cj + dk$. We can alternatively remove the i, j, k for convenience and think of the quaternion as a vector of four values $\langle a, b, c, d \rangle$. Linear momentum, p, is given by $p = mv$, where m is mass and v is velocity. p is described by components as $\langle p_x, p_y, p_z \rangle$. Angular momentum, L, is given by $L = I\omega$, where I is the *inertia tensor* and ω is a vector representing the object's axis of rotation. The inertia tensor is a matrix that represents the scaling factor between angular momentum and angular velocity (Witkin & Baraff, 1997). Each type of volume (such as sphere and box) has its own inertia tensor, and can be computed ahead of time and stored as a constant for each object. L is described by components as $\langle L_x, L_y, L_z \rangle$.

Our algorithms work with linear momentum and angular momentum, which are derived quantities, rather than the simpler linear velocity and angular velocity, for reasons argued in Witkin and Baraff's notes:

> The only reason that one even bothers with the angular momentum of a rigid body is that it lets you write simpler equations than you would get if you stuck with angular velocity. ... Angular momentum ends up

> simplifying equations because it is conserved in nature, while angular velocity is not: if you have a body floating through space with no torque acting on it, the body's angular momentum is constant. This is not true for a body's angular velocity though: even if the angular momentum of a body is constant, the body's angular velocity may not be! Consequently, a body's angular velocity can vary even when no force acts on the body. Because of this, it ends up being simpler to choose angular momentum as a state variable over angular velocity. (Witkin & Baraff, 1997)

Linear momentum is then chosen over linear velocity to maintain consistency with the choice of angular momentum over angular velocity (Witkin & Baraff, 1997).

To represent orientation, quaternions provide several advantages over Euler angles. Because quaternion calculations do not involve trigonometric functions, they are very efficient (Gould et al., 2007). Interpolation between one orientation and another is much easier to do using quaternions (Shoemake, 1985). Most importantly for our purposes, quaternions are much less susceptible to the numerical drift encountered when performing calculations using Euler angle matrices (Witkin & Baraff, 1997).

The orientation of a body can be represented by an axis of rotation ω and an angle of rotation θ about the axis. To represent this with a quaternion, let ω be given by $\langle v_1, v_2, v_3 \rangle$. We construct the quaternion as $\langle \cos\frac{\theta}{2}, v_1 \sin\frac{\theta}{2}, v_2 \sin\frac{\theta}{2}, v_3 \sin\frac{\theta}{2} \rangle$ (Gould et al., 2007). To combine two orientations, we need only multiply the two quaternion representations. Given two quaternions that represent orientation Q_1 and Q_2, $Q_2 Q_1$ represents the composite rotation of Q_1 followed by Q_2 (Witkin & Baraff, 1997).

A world object will hold two collections of force vectors that act upon it: a set of world-relative vectors $\mathbb{F}_W$ and a set of body-relative vectors $\mathbb{F}_B$. These are

Position	$\frac{dS(t)}{dt} = P(t)/m$
Orientation	$\frac{dq(t)}{dt} = \frac{1}{2}[0, \omega(t)]q(t)$
Linear Momentum	$\frac{dP(t)}{dt} = \mathbf{F}(t)$
Angular Momentum	$\frac{dL(t)}{dt} = \tau(t)$

Table 5.1. The ODE for rigid body movement (Witkin & Baraff, 1997).

summed to find the world-relative resultant force vector $\mathbf{F_W} = \sum_{\mathbf{f} \in \mathbb{F}_W} \mathbf{f}$ and the body-relative resultant force vector $\mathbf{F_B} = \sum_{\mathbf{f} \in \mathbb{F}_B} \mathbf{f}$ at each snapshot. $\mathbf{F}$ is the resultant force vector, which is given by $\mathbf{F} = \mathbf{F_W} + Q\mathbf{F_B}$, where Q is the matrix representation of the orientation quaternion. What this means is that $\mathbf{F_B}$ is transformed to world-relative coordinates by the world object's orientation and then summed with $\mathbf{F_W}$.

The world object updates its state by representing its physical state as an *ordinary differential equation* (ODE) and using an ODE solver to calculate its new state. Only the position, orientation, linear momentum, and angular momentum are updated with the ODE solver. The mass m remains constant, and the force vectors are updated by external events (either collisions with other surfaces or intervention from a client's command). The ODE is shown in Table 5.1, where $\tau(t)$ is the resultant torque vector at time t.

The orientation portion of the ODE deserves some explanation. The change in orientation, $\frac{dq(t)}{dt}$, involves multiplying the current orientation $q(t)$ by $[0, \omega(t)]$. The latter expression is shorthand for the quaternion $\langle 0, \omega_x(t), \omega_y(t), \omega_z(t) \rangle$ (Witkin & Baraff, 1997).

One of the most popular ODE solvers, fourth-order Runge-Kutta, or RK4, is used to solve the ODE, which is known to be very accurate and well-behaved for a

wide range of problems (Neumann, 2004). The algorithm works by evaluating the ODE at several points along the interval $(t, t + \Delta t)$ and taking a weighted average of these values. Contrast this method with the more basic Euler solver. Consider that, disregarding rotational effects for the moment, position can be defined as $S(t + \Delta t) = S(t) + vt + \frac{1}{2}at^2$. The Euler solver would solve this equation, using the velocity at time t as a constant for the vt term. Since acceleration will change the velocity over the interval, however, it misses the effect that the non-constant velocity contributes. RK4 is able to account for this change in velocity, yielding a much more accurate result.

Chapter 6
WORLD OBJECT REPRESENTATION

The system and tools by which objects within the world can be created are very important in simulation design. If the system is not versatile enough, only simple world objects can be created. If the system is too free-form and not well-defined, it may be difficult for the user to build objects correctly. Because this is a difficult trade-off, we went through three major versions of the world object representation system. Our first attempt left the user with very few options for creating objects that weren't rigid bodies. Our second attempt was incredibly versatile, but lacked structure and added a lot of overhead in object creation and processing. Our final attempt settles on a middle ground. By combining structure with versatility, we arrived at an architecture that promotes both ease-of-use and customization.

6.1 Rigid-body Architecture

Our early prototype code based the system around the WorldObject type, which, at the time, was our rigid body representation. Over time, as more and more of the system was built, each part began to rely more heavily on this as the basic type for all entities contained within the world. Unfortunately, this meant that non-rigid body entities, such as magnetic fields and radio waves, would have difficulty being represented.

There were two options to consider in regard to incorporating these types of entities. The first would be to try and represent them through the WorldObject type, perhaps by overriding the default rigid body behavior and providing each entity's own mechanisms. We considered this poor design. If an entity isn't a rigid body, why should it have anything to do with a rigid body type?

The other option would be to create types more general than WorldObject, usurping WorldObject's place as the basic entity type within the system. Unfortunately, this would mean having to change a lot of code that depended on the WorldObject type. Ultimately, however, it would be the preferable solution, as a lot of the code that depended on WorldObject didn't use the rigid body-specific mechanisms.

We began by creating a small hierarchy[1] with the Entity type as its root. Entity was incredibly general, and defined only that an object had some unique identification number. All other properties and behaviors would be given in subtypes. The Collidable subtype, for instance, could represent objects that could collide with other objects in the world, and the more specific RigidBody type represented objects that could not only collide with other objects, but bounce off of them.

When we were making this major shift in the simulator's architecture, we wondered if there weren't an even better, more flexible solution to representing entities. This lead us to implement the next major shift in the object representational system.

6.2 Entity/Component Architecture

In keeping with our design goal of extensibility, we decided to make the object representational system as flexible as possible to give the user the ability to create arbitrarily-rich virtual worlds. This second version of our world object architecture was heavily influenced by the *entity/component system*, an approach to object representation used often in video game engines (Kirmse, 2004). It is similar to the *Strategy* design pattern (Gamma et al., 1994), which allows Java objects to dynamically acquire behaviors.[2]

[1] Of Java interfaces.
[2] Much like languages with first-class functions can do trivially.

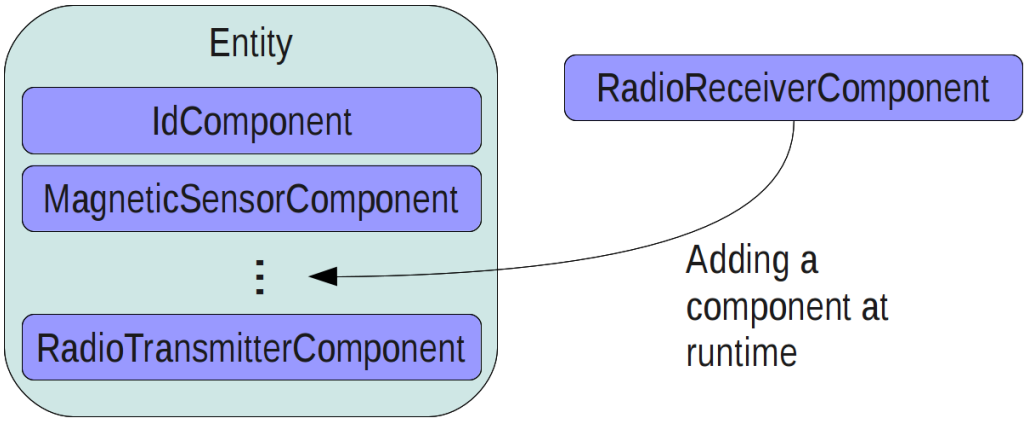

Figure 6.1. A depiction of an entity with several components. The RadioReceiverComponent is being added at runtime.

The system is built out of *components*, which contain some set of related data and behaviors, and *entities*, which contain an arbitrary number of components. (For the remainder of this section, the term "entity" will not refer to the top-level Entity interface that we have already discussed.) Each component type has a name, and an entity can contain at most one instance of any given component type (see Figure 6.1). Some components are simple, such as the IdComponent, which just stores a unique identifier, or the PositionComponent, which stores a position within the world. Others are more complex and incorporate a variety of data and behaviors, such as the RigidBodyComponent, which stores position and orientation state, a collection of forces, and behaviors that determine collision detection and response, among other things (see Section 5.2).

By combining components, entities could acquire new and varied behaviors, even at runtime. To add cohesion between components, components were given the ability to depend upon other components. For instance, instead of having RigidBodyComponent contain position data, it could depend upon the entity also having a PositionComponent to which it would refer when it needed to read or write position state data. By separating the position state from RigidBodyCompo-

NENT, other components that depended upon the entity's position could also access it. Since at most one component of any given type would be allowed per entity, components could refer to dependency components by name without ambiguity.

To add to user-friendliness, we made components themselves entities; that is, components could contain components, allowing tree-like structures to be built. The rationale for this design was as follows. Given the argument above, it would be beneficial to split off as many pieces of state and behavior from complex components as possible. That is, RIGIDBODYCOMPONENT should depend upon a POSITIONCOMPONENT, an ORIENTATIONCOMPONENT, a COLLISIONRESPONSECOMPONENT, and so forth, rather than define them itself, so as to allow other components to access them. Unfortunately, this would introduce a large amount of overhead when creating objects. Instead of simply creating an entity with a RIGIDBODYCOMPONENT, a user would first have to add all of the building-block components listed above.

By allowing components to contain components, we were able to achieve both the building-block structure of complex components as well as user-friendliness. As an example, RIGIDBODYCOMPONENT would contain the smaller components listed above. If a component added to the entity later needed to reference the POSITIONCOMPONENT, the access request would first go to RIGIDBODYCOMPONENT, which would then forward it to its contained POSITIONCOMPONENT. The logical effect would be the same as if the smaller components had been added manually.

The entity/component architecture as a whole allowed for a flexible system of describing objects that exist within the world. For instance, a user could create an object with rigid-body dynamics by simply adding RIGIDBODYCOMPONENT. A user could create an object with position and the ability to collide with other objects, but without rigid-body dynamics that allow the object to move, by adding only POSITIONCOMPONENT and COLLIDERCOMPONENT. This would define a volume

that collides with objects, but does not perform any collision response. Such an object could be used to create an unmovable wall or ground floor within the world. A user could create a radio wave object that is comprised of PositionComponent, ColliderComponent, and CollisionResponseComponent. The radio wave would begin at some position and expand outward, carrying a message with it. When it collided with another entity, the collision response component would attempt to retrieve that entity's radio receiver component, if it had one. If so, it could then invoke a method on the receiver, sending it the contents of its message. Thus, an object has been created that requires collision detection and response, but does not act like a rigid body.

6.3 Type-hierarchy Architecture

Although the benefits in regard to representation were numerous, the entity/-component architecture actually proved to be a hassle to work with. Because Java is strongly- and statically-typed, the mechanisms to retrieve arbitrary component types had to circumvent Java's typing safeguards. In practice, this ends up creating a lot of code, even on simple calls. We decided to pull back and confine the entity/component architecture to a smaller part of the system. Our new scheme restored the original generalized type hierarchy described at the end of Section 6.1, but added the flexibility in those parts of object representation in which it made more sense.

In particular, types that represented AUVs would need to provide a facility that allowed various capabilities to be added. For instance, one AUV might need a magnetic sensor, a thermometer, and a radio receiver, while another might need a radio receiver and a radio transmitter. Each of these capabilities would have its own

data and behaviors, and the set of capabilities would vary from one AUV instance to the next. This was a perfect fit for the entity/component system.

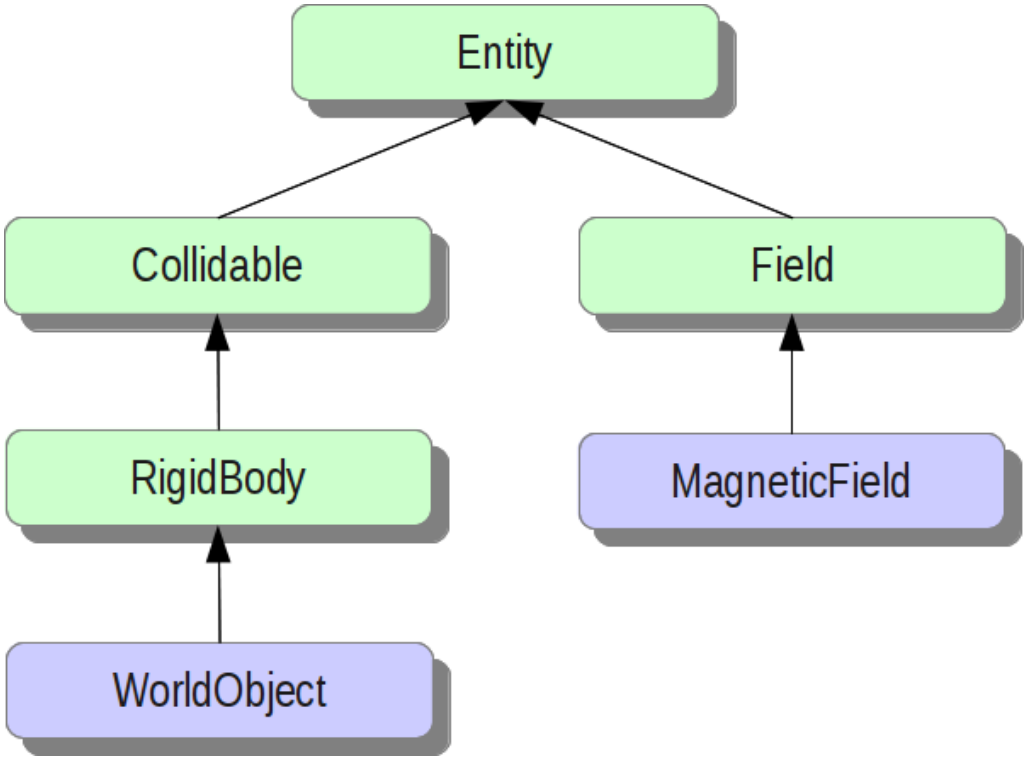

Figure 6.2. The current object representational system. Green types are interfaces and purple types are concrete. Arrows point from more specific types to more general types.

Our current architecture can be summarized as follows, and is illustrated in Figure 6.2. The top-level type is ENTITY, which is as defined in Section 6.1 (again, not to be confused with the "entity" in "entity/component system"). COLLIDABLE extends ENTITY and represents any entity which might collide with something. Collision detection algorithms work with COLLIDABLEs, but COLLIDABLEs are not necessarily associated with a collision response. FIELD also extends ENTITY and represents anything that permeates the entire world and can be evaluated at any point. This would be used to model magnetic fields or water currents. The RIGIDBODY type represents rigid bodies as described in Chapter 5.

The WorldObject type is then redefined simply to be a RigidBody that has a container for components. Thus, anything from simple objects such as pebbles to complex objects such as AUVs with arrays of sensors can be modeled using WorldObject and optionally adding any necessary components.

Chapter 7
NETWORKING AND THE CORRECTIVE FEEDBACK SYSTEM

The network server is responsible for communication with remote clients. This chapter discusses its more interesting functions. AUVSML is used as the medium through which communication is done, providing a flexible and easily-extensible language. We also discuss the Corrective Feedback System, which allows remote clients to send feedback data to the simulator such that it may correct its own representation of the world.

7.1 Client/Server Communication

All communication between client and server is encoded in AUVSML, a language we created. Mechanisms were written to easily transform Java objects into AUVSML (serialization) and to transform AUVSML into Java objects (deserialization). Any object that represents a piece of data that will either be sent to or received from a client implements the interface MESSAGE, which requires that the implementing class provide the TOAUVSML() serialization method and the FROMAUVSML() deserialization method.

7.1.1 AUVSML

AUVSML is a simple XML-based language that describes data sent back and forth between server and client. An example is shown below, where >>> indicates what the client sends. For a summary of AUVSML messages, please refer to Appendix A.

Listing 7.1. Client/server conversation example

```
>>> <auvsml><get-world-object-list-command /></auvsml>


<auvsml>
   <world-object-list-response>
      <world-object id="1" type="WorldObject" name="Bob" />
      <world-object id="2" type="WorldObject" name="Carl" />
      <world-object id="3" type="WorldObject" name="Tank" />
      <world-object id="4" type="Agent" name="myAgent" />
   </world-object-list-response>
</auvsml>


>>> <auvsml><get-propulsor-list-request agent-id="4" /></auvsml>


<auvsml>
   <propulsor-list-response agent-id="4">
      <propulsor name="Aft Thruster" />
      <propulsor name="Forward Thruster" />
      <propulsor name="Ventral Thruster" />
   </propulsor-list-response>
</auvsml>
```

The JDOM library[1] was used to aid in the manipulation of XML. JDOM transforms XML text into a tree of Java objects that represents its logical structure. It also provides functionality to easily transform such a tree into its textual representation.

AUVSML is very flexible in what can be represented. Although the simulator defines several useful commands itself, the user can easily add his/her own commands.

[1]http://www.jdom.org/

To do this, the user must define a MESSAGE type (discussed in Section 7.1.2). This will hold all of the data that is to be communicated.

As an example, say the user wanted to create a new message type that instructed an AUV to shut down some of its propulsors. The user could define something like the following.

Listing 7.2. Propulsor shutdown example

```
<auvsml>
   <shut-down-propulsors id="4">
      <propulsor name="Aft Thruster" />
      <propulsor name="Forward Thruster" />
   </shut-down-propulsors>
</auvsml>
```

The user might define the semantics of this message to mean "Shut down the aft and forward thrusters on the AUV with an ID of 4." So long as the user defines the MESSAGE type and provides methods to translate between this data object and the AUVSML string representation, it can be used just like the built-in AUVSML message types.

AUVSML messages contain data, but can also be associated with code. Messages that do this implement one of the COMMAND interfaces, which provide EXECUTE() methods that define the semantics of the command. The AUVSML data is loaded into a new instance of a concrete COMMAND class and can be passed around the system until it reaches the section for which it is intended. SIMULATORCOMMANDS, for instance, should be executed by the simulator, and typically involve things like pausing and resuming the simulation. WORLDCOMMANDS, on the other hand, deal with the world, and may be used for querying how many world objects there are. Each concrete COMMAND class overrides EXECUTE() to provide its own behavior.

EXECUTE() is passed necessary data, such as the SIMULATOR object or the WORLD object, which it can then manipulate.

7.1.2 Serialization and Deserialization

(Note: Although this section touches on some implementation details, it is important to discuss these to understand some underlying issues.)

Serialization describes the process of transforming a piece of data accessible within the programming language (in this case, a Java object) into some persistent form external to the language, such as plain text. Deserialization is the converse; it is the process of transforming a persistent piece of data into a language-accessible form. In the context of the system, all language-accessible data that must be serialized is represented as an object of type MESSAGE. All messages can be serialized into their AUVSML representations by invoking the TOAUVSML() method, and can be deserialized from an AUVSML string by invoking the FROMAUVSML() method.

Though the logic behind TOAUVSML() is straightforward, processing FROMAUVSML() is somewhat complex. This is due to the fact that we have a string that needs to be transformed into an object of some concrete implementation of MESSAGE, the type of which is unknown to us. We solve this issue by providing a mechanism to inspect the name of the AUVSML element, such as SHUT-DOWN-PROPULSORS above, and using Java's reflection facilities[2] to retrieve the appropriate concrete implementation.

The process of transforming an AUVSML string S into its corresponding instance MSG proceeds as follows.

1. Using JDOM, transform S into an object tree TREE.

2. Inspect the root tag of TREE. Let AUVSMLNAME be the name of this tag.

[2]Discussed in Chapter 9.

3. Find the concrete class C corresponding to AUVSMLNAME using a lookup table mapping names to classes.

4. Reflectively invoke C.FROMAUVSML(S) to generate a new instance of C with the data stored in TREE.

Most of the complexity involved in this process is due to the choice of Java as the implementation language. This is discussed in more detail in Chapter 9.

7.2 Corrective Feedback System

The *Corrective Feedback System* (CFS) allows clients to send back physical data at each snapshot to be incorporated into the simulation. CFS is customizable; a user running the simulator could choose to ignore all feedback, wait for feedback only from certain clients, wait for feedback only for a given amount of time, or the user could write his/her own behavior. All behaviors operate within the simulator's ADVANCETIMESTEP() method.

If feedback is ignored, the simulator advances the time step by simply calling world's ADVANCETIMESTEP() method. One of the provided behaviors, SIMPLE-FIXEDTIMEFEEDBACKTIMESTEPPER, will wait a fixed amount of time for feedback to be received before proceeding. It is implemented by advancing the time step on a copy of the world and using that data to inform clients that feedback can be sent.

Let W be the current state of the world, and let W_P, a *projected state* of W, be such that $W_P = W$. Initially, the projected state is just a copy of the W's state, but we then invoke $W_P \leftarrow advanceTimestep(W_P)$ to advance the state of the projected world, leaving W untouched. At this point, $W_P \neq W$. The state of W_P is sent to all clients with a marker that indicates that this state is only projected and not the actual state that the world has moved into. Each client then uses this data to perform whatever actions it would normally take. For instance,

if some client C is a mobile robot and W_P contains data that indicates that C's virtual counterpart moved, C would then move in the physical world. Each client may then send feedback. The simulator allows feedback to be received and waits for some specified amount of time. After this interval has elapsed, the simulator processes any feedback that requires processing before the time step is advanced in W. $W \leftarrow advanceTimestep(W)$ is then called, which informs all clients of the actual state change. The simulator then processes any feedback that requires processing after the time step has advanced.

Feedback messages contain data that can be used to alter the state of the simulation. The methods that actually perform these alterations are called *feedback processors*. Feedback processing is divided into two categories: processing that is done before the time step is advanced and processing that is done after. An example of pre-advancement processing would be a processor that performed vector adjustment. Consider a world object WO and its physical counterpart, client C. WO has several force vectors acting on it that determine its movement. Let $\mathbb{V}$ be the set of these vectors and $\mathbf{V_R}$, the resultant vector, be the sum of all vectors $\mathbf{v} \in \mathbb{V}$. Let $S(t)$ be a function that gives WO's position at time t and $Q(t)$ be a function that gives C's physical position at time t. Assume $S(k) = \langle x, y, z \rangle$. The simulation advances by a time step delta Δt and informs C that $S(k + \Delta t) = \langle x + 10, y, z \rangle$. Assume that C moves awkwardly, resulting in $Q(k + \Delta t) = \langle x + 9, y, z \rangle$. C sends a feedback message containing $Q(k + \Delta t)$ to the simulator. The vector adjustment processor would then examine the difference between $S(k + \Delta t)$ and $Q(k + \Delta t)$ and would attempt to find a new set of vectors $\mathbb{V}'$ such that $\mathbf{V'_R} = \sum_{\mathbf{v} \in \mathbb{V}'} \mathbf{v}$ and $advancePosition(S(k), \mathbf{V'_R}) \approx Q(k + \Delta t)$. $\mathbb{V}'$ can then be used when advancing the time step of W.

An example of post-advancement processing would be a simple direct alteration processor. Let WO, C, $S(t)$, and $Q(t)$ be given as they were above. If $S(k + \Delta t) = \langle x + 10, y, z \rangle$ and $Q(k + \Delta t) = \langle x + 9, y, z \rangle$, then a direct alteration processor would set WO state as $S(k + \Delta t) \leftarrow Q(k + \Delta t)$.

Though the direct alteration processor ensures that the virtual object will have exactly the same position as its physical counterpart, it bypasses the world's collision detection routines since it runs after the world's time step advancement. That would mean that if the path to the physical position encountered a collision that the path to the virtual position did not encounter, the direct alteration processor would miss the collision. Since the vector adjustment processor runs before the world advances its time step, the corrections will be subject to the collision detection routines, avoiding this problem. Since the vector adjustment could be inaccurate, however, we cannot be certain that the virtual object will end up at the exact physical position.

The type of feedback processor used can be specified within each feedback message, although the simulator also has the ability to disregard this and use whatever processor it wishes. It should also be noted that the separation of processors into pre- and post-time step advancement categories can also be user-defined. That is, one could write a simulator that introduces other categories, or that even allows the feedback data to be incorporated in some other way unrelated to when the time step is advanced.

Chapter 8
RESULTS

We ran several experiments with the system to demonstrate its core abilities and evaluate how well it performed. The first of those described here tests the CFS by comparing both a simulation with and a simulation without a client sending feedback. In the second, we evaluate the runtime performance of the default algorithms by running a simulation with a large number of colliding rigid bodies. Finally, we make observations regarding the difference between simulations that run on one machine and those that are run in a distributed environment.

8.1 CFS Demonstration

The purpose of the Corrective Feedback System is to allow remote clients to send physical data to the simulator in order to correct the virtual world's notion of what the state of reality is. To provide a basic test of CFS, we created a simple client that acts like a physical client sending feedback messages. Note that for this test, an actual physical client was not used. From the simulator's perspective, all clients are alike, so using a software client that sends feedback data as a physical client would elicits the same behavior. After every few position updates it receives from the simulator, it alters the position slightly and sends this new position back to the simulator as feedback. This is meant to simulate a minor "hiccup" in the physical agent's movement.

To compare this scenario with what would happen in a situation in which no remote client was present, we ran two simulations. Simulation A created one world object which was controlled by the simulator itself. It followed a simple movement pattern that was easy to identify (explained below). Simulation B contained the

same world object, but added a remote client that sent feedback messages as described above.

The world object in both simulations would alternate between moving right and left on the Y-axis. When traveling to the right, if it moved past a certain point on the Y-axis, a force pushing it to the left would be applied until it returned to some position before that point. The opposite would happen on the other side: after moving too far to the left, a force pushing it to the right would be applied until it returned closer to the center. By observing the normal behavior in Simulator A, we saw that the object would go farther in both directions over time.

Simulation B performed as expected. The movement pattern of the world object was much the same as in Simulation A, but its movement was also jerky. Every few steps, it would jump a tiny increment forward or back due to the corrections that were being sent by the remote client.

We added a third simulation, Simulation C, to demonstrate what would happen if the remote client simply disregarded the simulator's updates and reported feedback based on its own model of its movement. The remote client keeps track of a position $\mathbf{S}$ independent of the position given by the simulator and a constant velocity v. Each time the simulator sends it an update, the client calculates $\mathbf{S} \leftarrow \mathbf{S} + v\Delta t$ and sends this value as feedback.

The result is that Simulation C's world object moves almost exactly as the remote client dictates. The only discrepancies are minor; there are times when the world object's movement stops for a small interval. This is most likely due the server and client being slightly out-of-synch in some situations.[1]

[1]The tests in this section run the server and all of the clients from the same machine, but from different threads. This achieves the same logical effect as if the clients were run on separate machines, but allows the user to run the test from one machine and one program. Only so many threads may be active at any given instant, however. Thus, thread context switching may be responsible for some of these inaccuracies.

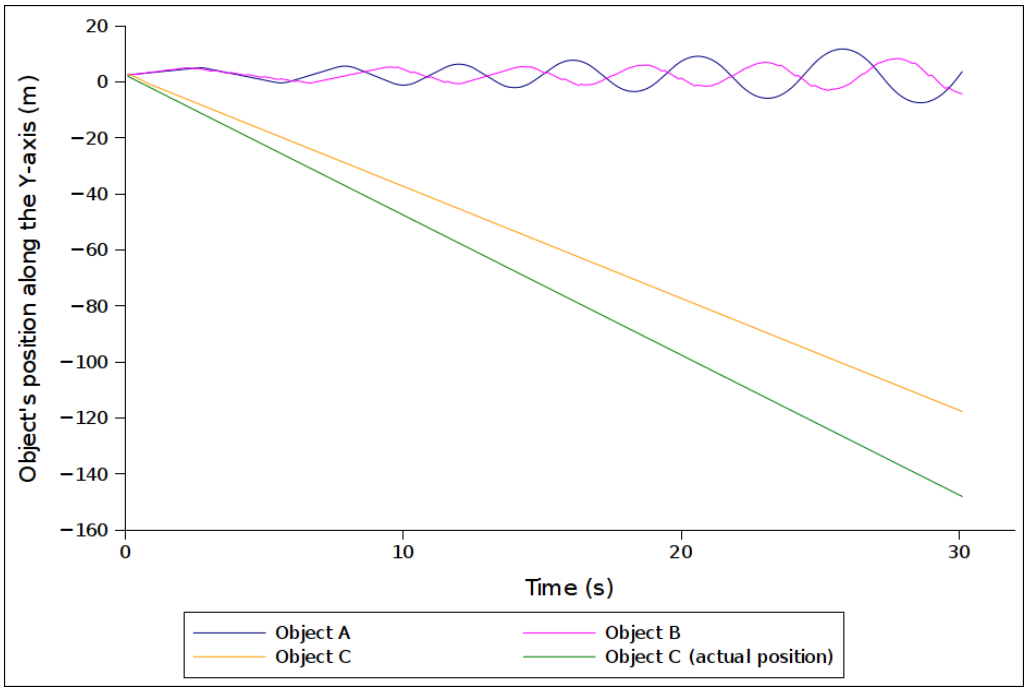

Figure 8.1. A comparison of the three objects' positions.

Figure 8.1 plots each object's position along the y-axis over time. Object A's position varies smoothly, while Object B's position is a distorted version of Object A's path. Object C's path shows the client taking over, and is plotted alongside Object C's actual position. Note that the virtual Object C progresses more slowly than the actual Object C. This is due to the fact that there is no guarantee of when the feedback will reach the simulator. A future improvement would be to extend CFS to allow tight synchronization with the client. This would entail waiting for each feedback message to be processed for the current snapshot before moving the simulator on to the next. Currently, the system processes feedback whenever it arrives.

This illustrates the flexibility of CFS, as client control can be minimal or extreme. The part of the system that handles how and when feedback messages are gathered is written to an interface, much like the time stepping and collision detection algorithms discussed in Chapter 5. Another implementation for this interface

could provide better synchronization between client and server, which would result in simulations like Simulation C progressing more smoothly.

8.2 Runtime Performance of Default Algorithms

Where the previous test removed consideration for collisions by creating only one world object, the next series of tests exercised the collision detection and adaptive time stepping algorithms exclusively. To analyze how well our default algorithms performed, we ran several simulations with varying numbers of objects and counted how many primitive operations each algorithm executed.

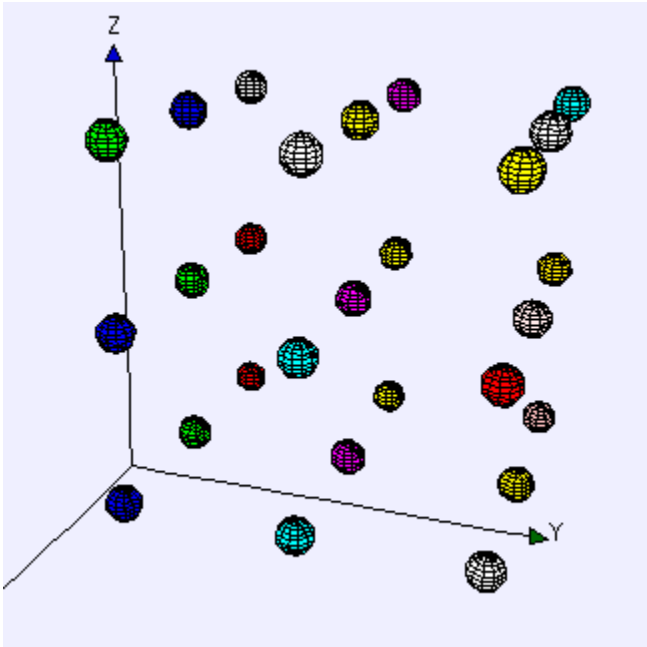

Figure 8.2. A $3 \times 3 \times 3$ cube formation with 27 world objects.

As described in Chapter 5, each time step advancement the world executes consists mainly of three steps: a preliminary shortening of the time step, partitioning the world into collision sets, and finding the time of the earliest collision. For these tests, each of those default implementations were instrumented to count the primitive operations that each executed during one cycle of the simulation.

This was done manually by adding a counter variable. Operations such as assignment and those that performed arithmetic would increment the counter by 1. Calls to functions that processed lists would increment the counter by n, where n was the size of the list. This counter data was then collected over the course of 100 time step advancements for each simulation.

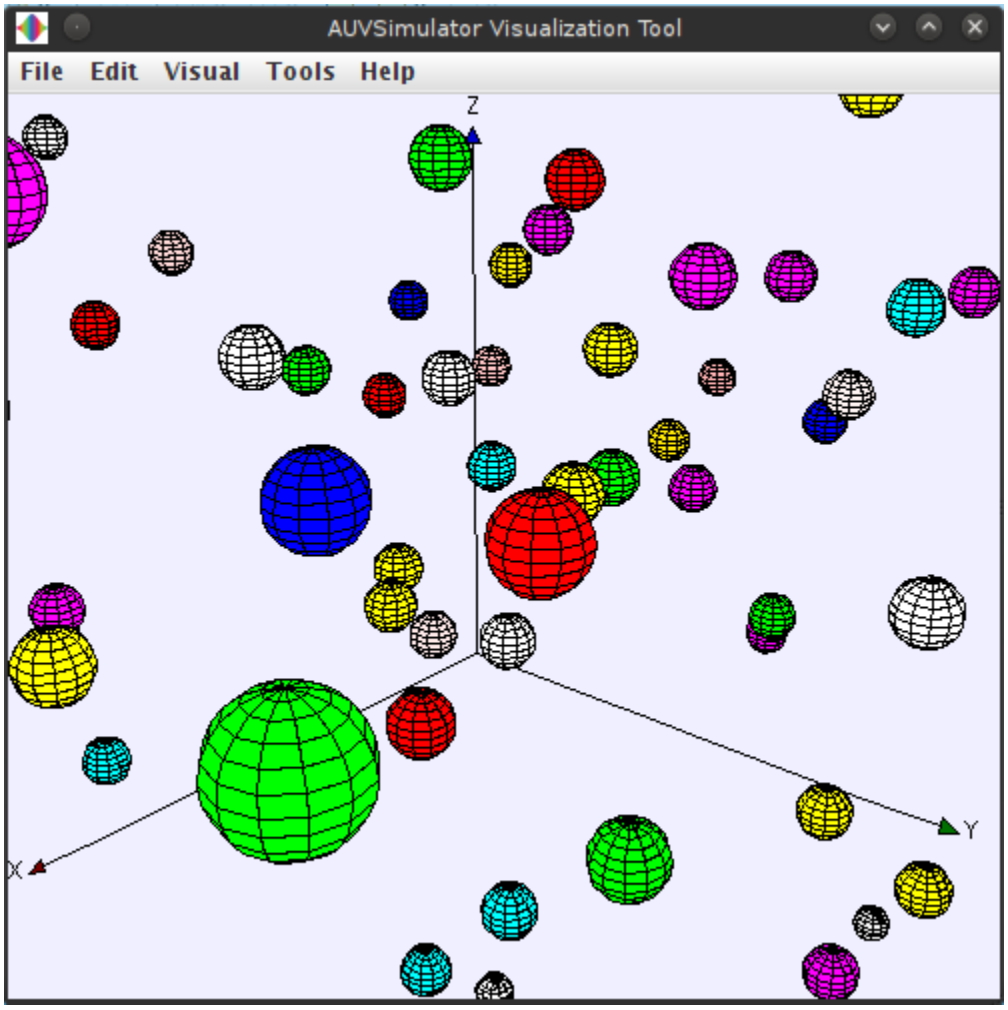

Figure 8.3. A simulation with 100 world objects (some are outside of the shown viewing volume).

10 simulations were run, with the first containing 10 world objects, the second containing 20, and so forth, with the last containing 100. The world objects were of uniform size and shape, and were placed into a cube formation at the beginning of

a simulation. For instance, if there were 27 objects, then they would be placed into a $3 \times 3 \times 3$ cube formation (see Figure 8.2). They were placed such that adjacent objects had a distance of $5d$ between them, where d is the diameter of the object. Over the course of the simulation, the world would constantly apply forces to each object that would direct it toward some target point. Each object had a different target, but they were all close enough together so that many collisions would end up occurring. Originally, objects were all directed at the same target, but this lead to a very predictable behavior of the entire cube collapsing into itself and expanding repeatedly. As Figure 8.3 shows, using different targets for each leads to more unpredictable behavior. By choosing targets that the objects will continually try to navigate to, we prevent them from spreading out infinitely after any initial collisions.

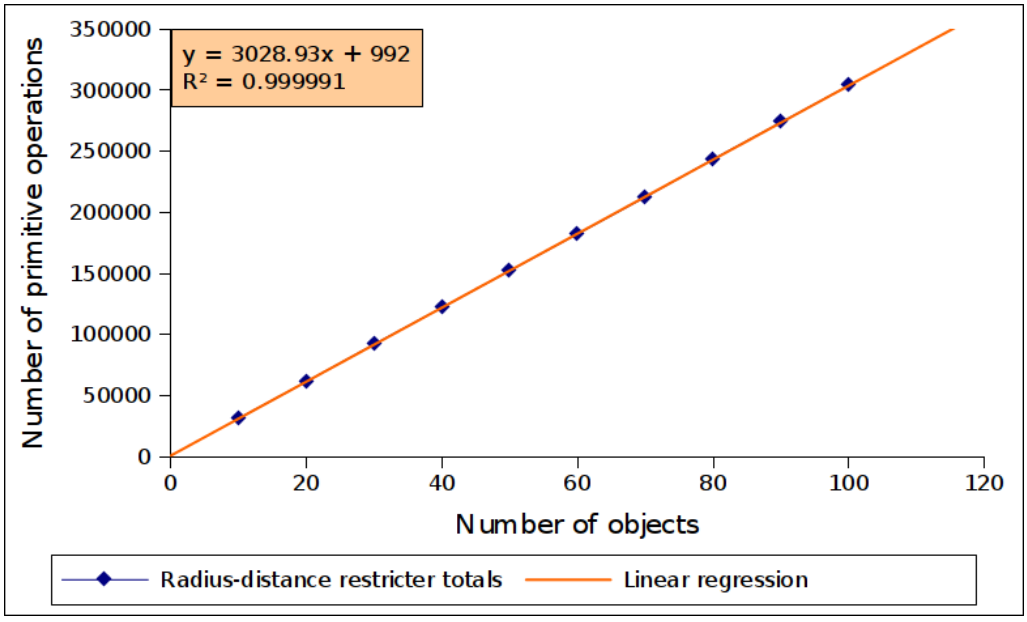

Figure 8.4. Actual runtime for the radius-distance restricter time stepping algorithm. Regression line data is shown in the orange-shaded box.

The runtime for the default preliminary adaptive time stepper, which ensures that no object moves more than its own radius per time step and is referred to as the radius-distance restricter, is plotted in Figure 8.4. The graph shows a clear linear

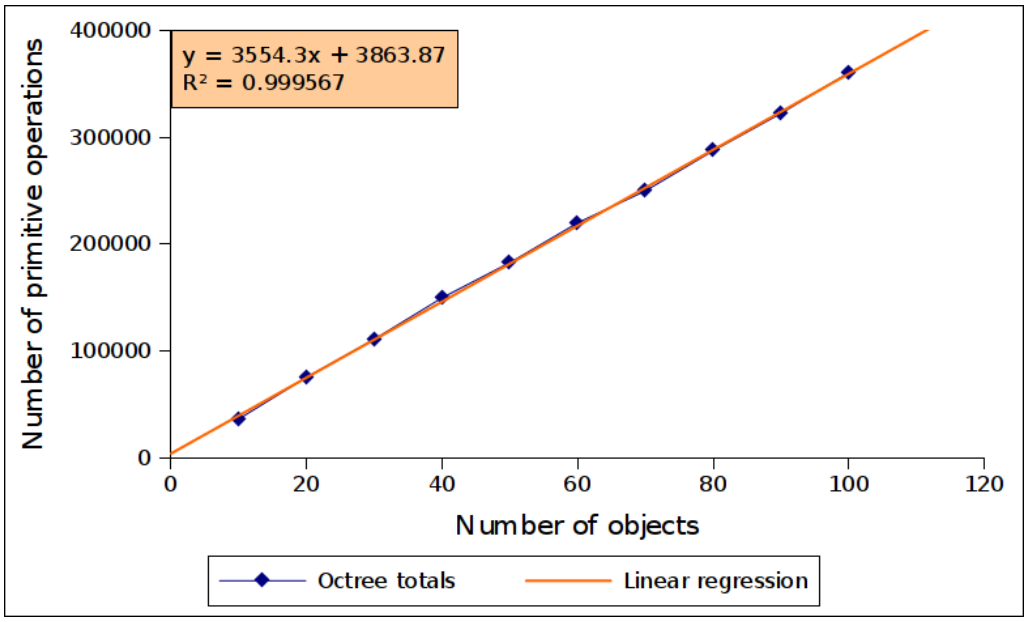

Figure 8.5. Actual runtime for the octree partitioning algorithm.

trend, which suggests that it runs in $O(n)$ time, where n is the number of objects. A regression line, shown in orange, fits the data with $R^2 = 0.999991$.

The runtime for the octree partitioner is shown in Figure 8.5. Here too, a linear trend is evident, suggesting a $O(n)$ runtime. A regression line fits the data with $R^2 = 0.999567$.

The runtime for the algorithm that finds the time of earliest collision is plotted in Figure 8.6. The trend appears somewhat polynomial, with a slight change between $n = 70$ and $n = 80$. From the graph, we hypothesized that this algorithm ran in $O(n^2)$ time. To verify this, we performed a curve fit analysis. A 2nd-order polynomial curve was fit to the data with $R^2 = 0.991847$.

It should be mentioned that this differs from the results of our theoretical analysis of this algorithm in Chapter 5, where we stated that the runtime is $\Theta(1)$ per invocation. While this is true, we must also consider how many invocations are performed per time step. Since this algorithm must be executed on each colliding pair (and some pairs that almost collide), the number of collisions that occur within the

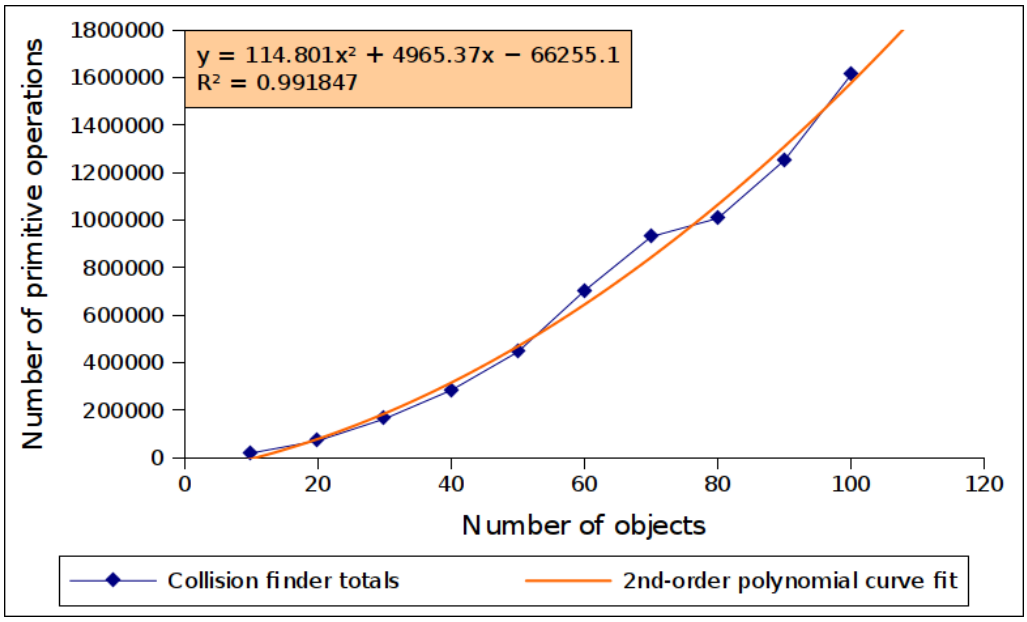

Figure 8.6. Actual runtime for the earliest collision finding algorithm.

given interval must be taken into account. Much more complex techniques would be necessary to pin down a reasonable theoretical bound, since the number of collisions per interval is difficult to estimate. Because such analyses are beyond the scope of this thesis, we will defer to the empirical results.

We also ran a test to see how these algorithms performed in real time. In particular, we asked "Do these algorithms cause the simulator's world time to elapse more slowly than real time?" We ran the same setup as before: ten simulations, the first with 10 objects, the second with 20, and so forth. Each simulation would run until two minutes of world time had elapsed. The real time that it took was then measured. The results are shown in Figure 8.7.

We can see that as the number of objects increase, the amount of real time it takes to run two minutes of simulated time also increases. The increase appears polynomial, so we applied a curve fit analysis. The 2nd-order polynomial appears to fit well, with the exception of the beginning of the curve. Because of this, we also applied a 3rd-order polynomial, which fits the data at the beginning well. Note,

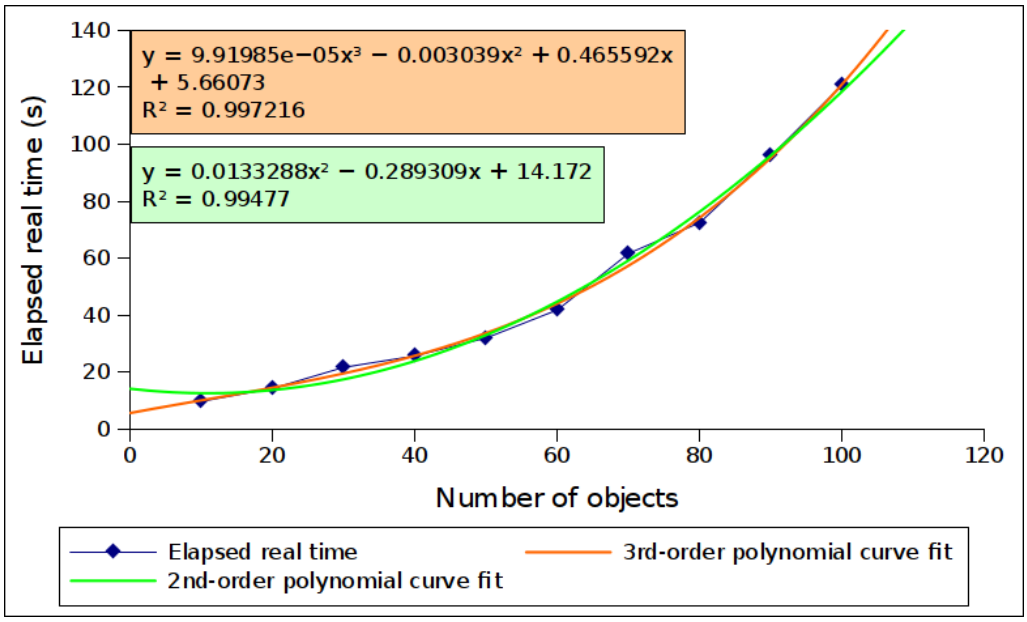

Figure 8.7. The real time it took to run each simulation.

however, that the x^3 coefficient for the curve, 9.91985×10^{-5}, is very small, suggesting that the 3rd-order term isn't significant.

This behavior is to be expected, since a simulation with more objects is more likely to encounter a greater number of collisions. The algorithm for finding the earliest collision was designed to slow the simulation down when a collision is imminent, thus yielding progressively slower runtimes as the number of objects and collisions increases. It is also important to note that although slowdown is experienced, it isn't until around 100 objects exist in the simulator that the elapsed real time is actually close to the elapsed world time. For less objects, the simulator is able to run faster than real time, despite the collision finder algorithm's intentional slowdown.

8.3 Comparison of Distributed and Non-distributed Simulators

After testing the default algorithms and the CFS, we tested one other major component to the system. The network interface allows remote clients to connect

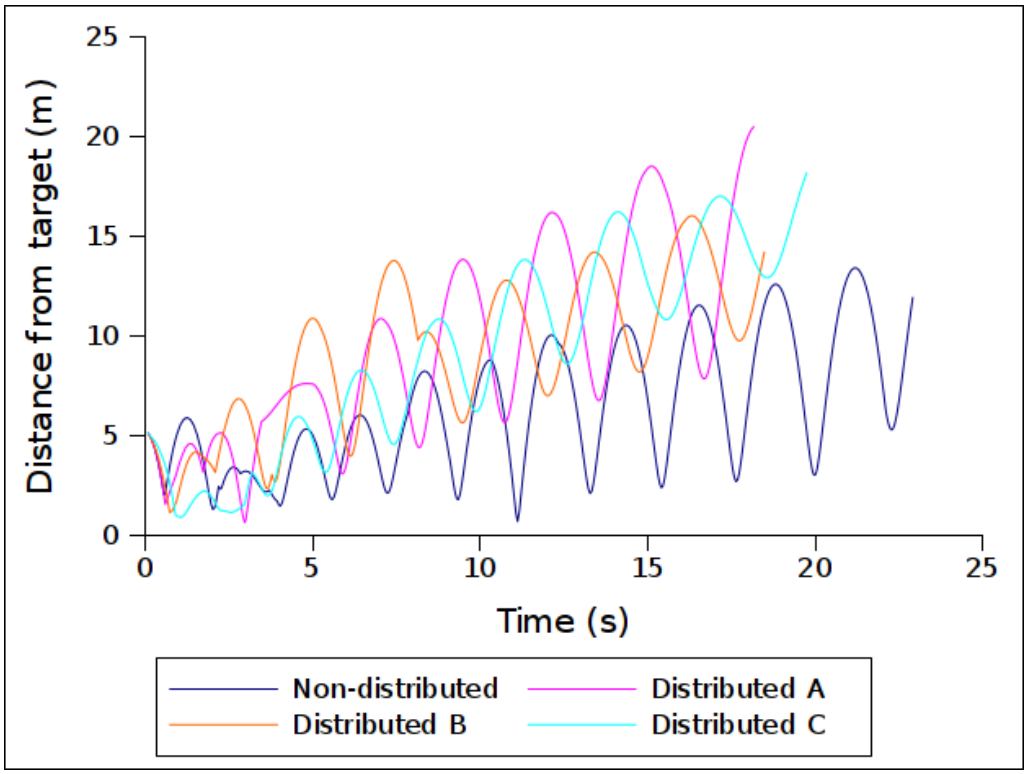

Figure 8.8. Distributed vs. non-distributed simulators. Each line tracks the distance from the target position for the same object for a different simulation.

and interact with the simulator by sending commands and receiving updates. Because our simulator does not yet provide solutions for tight synchronization with clients, simulators running in distributed mode will have different resulting states, even when the simulation is set up the same way for each. An example of tight synchronization would be to have the simulator and physical clients advance in lock-step with one another. For instance, the simulator would advance to time $t + \Delta t$, send movement data to the physical clients, and wait for their feedback before advancing the time step again. Currently, the simulator runs without waiting for feedback, processing it whenever it comes in from the network.

For these tests, we set up the world as follows. For each of four simulations, five world objects are placed, their positions chosen arbitrarily ahead of time. The position chosen for each object is the same for each simulation. The first test runs

a non-distributed simulator; that is, the objects are controlled within the simulator itself, and no remote clients are connected. Each object has forces applied to it, much like with the objects in the previous section, that eventually move it toward a target point. This target point is chosen to be the same for all five objects, so collisions occur often.[2] For this simulation, each run produces the same results.

The remaining three simulators all have the same setup. Five remote clients connect, each one assuming control of one of the world objects. The commands to move toward the target point come from these clients, rather than from within the simulator. Due to the non-deterministic timings inherent in thread scheduling and network traffic, each simulation generates different results. The results of all four are plotted in Figure 8.8, with the y-axis representing the distance from the target point. The difference between each is clear.

Although it may not be an issue for some simulations, others may require the precision that a synchronized connection between server and client can provide. Fortunately, because of the extensible nature of the system, an implementation of the interface that controls network message flow could be easily extended to allow tight synchronization.

[2]Because the objects are not placed uniformly, the behavior of the objects over the course of the simulation seems unpredictable. Compare this with the setup of the previous section's tests, in which the objects were placed uniformly, but the targets were different for each object.

Chapter 9
FUTURE WORK & CONCLUSION

The highly-extensible and distributed nature of the simulator provides numerous possibilities for future work. By extending it, alternative algorithms can be used to create more efficient implementations of the simulator's core logic. Tools can be developed that connect to the simulator as clients, gathering data to generate visual and statistical analyses. Yet more can be done by modifying those parts of the simulator's software that are fixed. We discuss some improvements with the AUVSML library and the core algorithms used by the simulator, explore some interesting possible uses of the Corrective Feedback System, and end with a discussion on how reimplementing some parts of the simulator with other languages might provide an improvement in readability and flexibility over the current Java implementation.

9.1 An Improved AUVSML Library

AUVSML is handled cleanly by the library provided by the simulator. Each MESSAGE type has TOAUVSML() and FROMAUVSML() methods to easily translate between data objects and their corresponding AUVSML representations. Unfortunately, though each MESSAGE is easy to use, the process of defining a new MESSAGE type is somewhat tedious. This is due to the fact that the serialization code is done by hand; there is no automated tool that does it and the creator of the class must write it him/herself. Even for simple data structures, such as a list of values, this can produce about twenty lines of code for serialization and deserialization routines.

To mitigate the possibility of bugs in the manual serialization process, we use *unit tests*, which are programs that each test some small part of the system independent

of most or all other parts. We write a unit test for each MESSAGE type that creates a MESSAGE object, serializes it into an AUVSML string, and then deserializes that string back into an object. If the first and second objects are equal, then we can be reasonably certain that the serialization code works.[1]

Since we are defining each particular MESSAGE type ourselves, we must also define equality for each type. In Java, this is done by providing an implementation for the EQUALS() method. According to good Java programming practices, any class that defines EQUALS() should also define the HASHCODE() method (Bloch, 2008). The reasoning behind this has to do with the fact that HASHCODE() is used to test equality in some situations rather than EQUALS(), and so the two methods should always agree on what is equal and what isn't.

Ultimately, this means that for any MESSAGE subclass, anywhere from 30 to 250 lines of code might be required. For a device that is ultimately a language-agnostic remote procedure call, this is overweight. Future work could improve on this by abstracting away all boilerplate code, allowing the creator of the MESSAGE to focus only on what the MESSAGE's data and behaviors are.

There are several approaches that could achieve this objective. One would be to use an automated XML serialization tool. Several are available for Java, including Simple[2] and XStream.[3] During the course of the project, we evaluated these tools and found them unsuitable for the exact XML structuring style we wanted. Further research into these tools, however, might be beneficial, and certainly our style might evolve over time to accommodate these libraries better. Another approach would be to represent AUVSML messages not as classes and objects, but rather as simple

[1]For improved confidence in these tests, multiple instances of the type, each with different values, can be put through the test rather than just one.

[2]http://simple.sourceforge.net/

[3]http://xstream.codehaus.org/

associative arrays. For messages that define behaviors, however, this may not be tenable without resorting to reflection.[4]

9.2 Alternative Algorithms

The simulator is designed for extension, and many parts that perform critical computations, such as calculating the next time step delta, are written using interfaces. Thus, multiple implementations can be written that extend the interfaces, allowing the user to program his/her own solutions. Most users, however, may wish to run customized simulations without having to write custom code. The system already provides default implementations of all necessary interfaces, but other built-in implementations should be available.

Future work could see development of a variety of alternative algorithm implementations. For example, in Chapter 5, an analytical method for detecting collisions was discussed. Although one that took rotational effects into account might be untenable, a user might be interested in using it for simulations in which the rotational effects are not a concern. The current numerical approach ensures that world objects do not move a distance greater than their radii in a time step. This is largely to prevent objects going so fast that the time step delta misses a collision. In some simulations, however, the creator can know that objects won't go past a certain speed. In these situations, a less restrictive time step predictor could be implemented. By providing several alternative implementations, the user will be able to pick those that suit his/her simulation best, without having to write the code.

[4]Since there would be no class for methods which define the behaviors to live in.

9.3 Complementary Software

The simulator was designed to be used with other pieces of software in a language-agnostic way. Though the simulator is implemented in Java, programs written in any language should be able to communicate with it easily using AUVSML. Thus, many possibilities exist for software tools that complement the simulator. One avenue of development is writing visualization tools. A crude visualization tool was written in Java for the simulator project using the Open Source Physics library. Its purpose was to provide visual confirmation of how certain physics algorithms operated, so an advanced graphical display was unnecessary. Other, richer visualization options are possible, however.

A more graphically-advanced visualization program using the Unity game authoring tool[5] is being developed in MaineSAIL.[6] The software starts a relay host that serves as a translator between Unity's network interface and AUVSML. Unity has no Java component, making the project a successful proof-of-concept of language-independent interoperability with the simulator.

Another useful tool would be an administrator interface. Currently, a simulation is set up by configuring several world objects and adding them to the world. From that point on, when running as a server, the only way to interact with the objects is to send commands through the network interface. An administrative tool would connect to the server as a client and register itself as a listener for all events. It would provide the ability to not only manipulate objects, but to manipulate the world and simulator as well. For instance, it could allow the user to pause and resume the simulation, add or remove objects, and disconnect other clients from the server. Some of these commands are already available to connected clients, but the AUVSML messages have to be typed out by hand. A client that provides a

[5]http://unity3d.com/

[6]By Michael Brady Butler, an undergraduate student.

graphical user interface with an integrated visualization would enable users an easy and intuitive way to interact with their simulations.

Another option would be to use the CFS in tandem with another simulator to ensure algorithm correctness.[7] The simulator S_1 would run a server, to which some other, more mature simulation software, S_2, would connect as a client. S_2 would register itself as a listener for all events, creating a mirror image of S_1's environment within itself. At each snapshot, S_1 would calculate a set of new object positions $\mathbb{P}_1$ with its physics engine and send the results to S_2. S_2 could calculate a set of new object positions $\mathbb{P}_2$ with its own physics engine. S_2 would then send corrections to S_1 as feedback messages. S_1 could make use of this in any number of ways. For instance, it could simply gather statistics on how well it performs, or it could use artificial intelligence techniques to train its own physics engine on-the-fly from S_2's corrections.

9.4 A Simulator that Learns How Better to Simulate

The above application can be extended to use reality itself, rather than a better simulator, to improve its accuracy. Computer simulation can be a very useful tool in studying physical systems with the advantage of having very fine control over the virtual environment in ways that might not be possible to recreate in reality. Being a virtual environment, however, it cannot perfectly replace the state and workings of an actual physical system; it can only provide an approximation. Certainly, very good approximations can be incredibly useful. A primary focus of this project was to mitigate virtual inaccuracies by accepting physical data into the model. The methods so far described for using the framework that we have developed, however,

[7]Thanks to M. B. Butler for this idea.

have been very basic. The framework is very general, and so admits many powerful possibilities.

One such possibility is the development of a simulator that uses the physical environment as a performance measure to learn how to improve its models. Let us express the simulator's model of the environment as a state transition function $M(S_t^V, \Delta t)$ that maps a virtual state at time t S_t^V and a time step delta Δt to a new virtual state $M(S_t^V, \Delta t) \rightarrow S_{t+\Delta t}^V$. Let the physical environment, as perceived through sensors, be represented by a function $P(t)$ that maps a time t to a perceived state $P(t) \rightarrow S_t^P$. Let us also define an operation "$-$" on states as such: If S_i and S_j are states, then $S_i - S_j$ represents a quantitative "difference" between the states, where difference is loosely defined and dependent on the situation. In general, if $S_i - S_j = 0$, then S_i and S_j represent the same state. If $S_i - S_j$ is a small value, then the states are close approximations of one another. If $S_i - S_j$ is a large value, then the states are not alike. The objective will be to pick M such that $M(S_t^V, \Delta t) - P(t + \Delta t) < \epsilon$, where ϵ is some small value representing tolerance of inaccuracy.

Given this problem description, machine learning algorithms can be applied to find M. One interesting approach would be to use *genetic algorithms* for this task. Genetic algorithms begin with a set of n randomly-generated states, where each state is encoded as a string. Typically, strings of bits are used. Each state is then rated by a *fitness function*, which yields higher values for states that are closer to some goal state. States will be weighted according to their fitness and chosen randomly for *reproduction*. Each reproducing pair of strings will be spliced together to form a child string, and an optional *mutation* may be applied by flipping one of the child's bits. The states represented by the new strings are added to the set of states, replacing the states of lowest fitness. The intent is that the good qualities of highly-

ranked states will combine in the child states, and over time, future generations will yield better and better solutions. (Russell & Norvig, 2010, p. 126)

The simulator software could be easily modified to support learning using genetic algorithms. Using a few minor adjustments to the current architecture, the simulator could run n virtual worlds in parallel. Each world would have its own physics model M_i, which would have a binary string representation. We would start by producing n binary strings that encode the current physics model, with each string permuted slightly. We would run each model for one time step, producing $M_i(S_t^V, \Delta t)$. By taking the difference of this value and $P(t + \Delta t)$, we have a fitness function. Those M_i with the highest fitness values would be selected for reproduction, producing new states. The process is repeated until an M_i is found such that its value is less than ϵ, as defined above.

In this way, we can create a simulator that uses the physical environment to teach itself how to better simulate that particular environment. Over time, the models produced by the genetic algorithm may yield more realistic representations of state transitions than those produced by humans.

9.5 Alternative Language Implementation

Some of the difficulties that the improvements above seek to address stem from how those parts of the system were implemented in Java. The Java programming language was chosen for implementing the simulator for a number of reasons. One important reason was familiarity with the language and its capabilities; there was no need to invest time into learning and using Java. Many useful libraries are available for Java, including the standard library, which provided us with tools to write networking code without concerning ourselves with low-level details, and the Open Source Physics library, which was utilized for simple visualization tools and some

mathematics algorithms. Java's static and strong typing provided advantages with type checking, and its interface language construct was useful in cleanly describing the interfaces through which objects could interact.

There were several areas, however, that became somewhat awkward to implement using Java. In particular, we discuss how another language choice might benefit the implementation of the AUVSML serialization process and the entity/component architecture. Because Java was chosen, we have the option of implementing these pieces of the simulator in other languages that run on the Java Virtual Machine, leaving the remaining Java implementation intact.

9.5.1 AUVSML Deserialization

The issues encountered dealt mostly with Java's type system. One example of this is the AUVSML deserializer. Each AUVSML message can be stored in memory as a MESSAGE object, where the specific type of object is determined by the first part of the message's contents. For instance, a message that encodes a change in the state of an object's movement is represented as an object of type MOVEMENT-STATECHANGEMESSAGE. When incoming messages are processed, the system must figure out the appropriate type of object in which to store the data. As described in Chapter 7, this is a somewhat elaborate process in Java, and involves storing a hard-coded lookup table of MESSAGE types and using reflection (described next) to create a new object.

These difficulties stem from the fact that Java's reflection facilities are cumbersome to use. Reflection is the functionality of a language that allows programmatic access to runtime information (Oracle, 1995b). In the context of Java, type definitions, or classes, are composed of variables and functions (called methods) and are uniquely named. The name of the class and the names of the variables and methods are only available to the runtime through the use of reflection. In the context of our

problem, the class name of a particular MESSAGE type is required for deserialization. The process involved is riddled with boilerplate code that adds a lot of bulk. It should also be noted that in general, use of reflection can reduce performance (Oracle, 1995b).

This is not as large of an issue in other languages. In Python, for instance, reflection is handled in a more streamlined way. The names of a class, its variables, and its methods are typically available to every object by means of an easy associative array lookup. Calling a method reflectively, such as the one to create a new object of the given type, is done in the same way as one would do non-reflectively, with the exception that the method must first be retrieved from the associative array. This is due to the fact that methods and functions are first-class objects in Python, meaning that they can be stored in variables. In Java, this can only be achieved by first acquiring a METHOD object that represents the method to be used. The METHOD.INVOKE() method must then be called to actually call the represented method. This process requires close to 20 lines of code due to the fact that most reflective calls could each potentially throw several types of exceptions, each of which must be caught. A dynamic language like Python, where access to runtime information is the rule rather than the exception, would better suit a problem like this.

9.5.2 Entity/Component Architecture

The original entity/component architecture, which may be revisited in future work, had similar issues due to its dynamic nature. This discussion will first require a brief explanation of Java's inheritance system.

In Java, classes can form a hierarchy by extending, or subclassing, other classes, where the subclasses provide more specific behavior than the superclasses. For instance, an ANIMAL class may be subclassed by FISH, which adds the ability to

swim and the property of having fins. FISH can then be subclassed by CATFISH, which adds bottom-feeding behavior and the property of having long barbels. This concept is called *inheritance*. Java only allows single inheritance, which means that each class must have no more than one superclass. For instance, we cannot define a FLYINGFISH by having it subclass both FISH and a BIRD class.

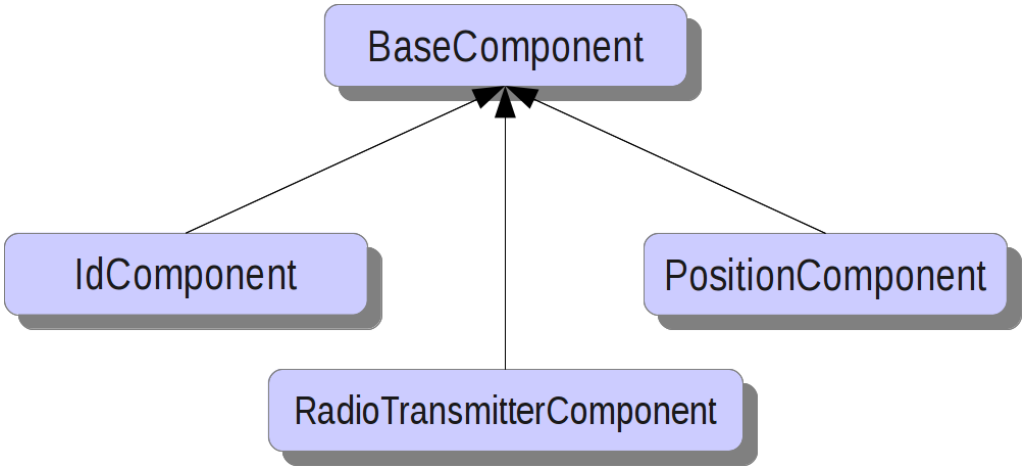

Figure 9.1. A visual representation of components that use inheritance. Each component type derives from the BASECOMPONENT superclass by extending it, preventing those components from extending other classes.

Java does allow classes to implement multiple interfaces, however. In Java, the interface construct defines what types of methods a class can have without actually implementing them. Each class that uses the interface will provide its own implementation. So if FISH and BIRD were interfaces, rather than classes, FLYINGFISH could implement the methods declared by FISH and BIRD and would be considered not only of type FLYINGFISH, but also of type FISH and of type BIRD.

With this in mind, we consider the design of the entity/component architecture. Each component needed to be of some common type, COMPONENT, so that a collection of components could be stored easily. In the original design, COMPONENTS could contain other COMPONENTS. So each COMPONENT would need to implement logic for adding, removing, and retrieving child COMPONENTS. On top of this, spe-

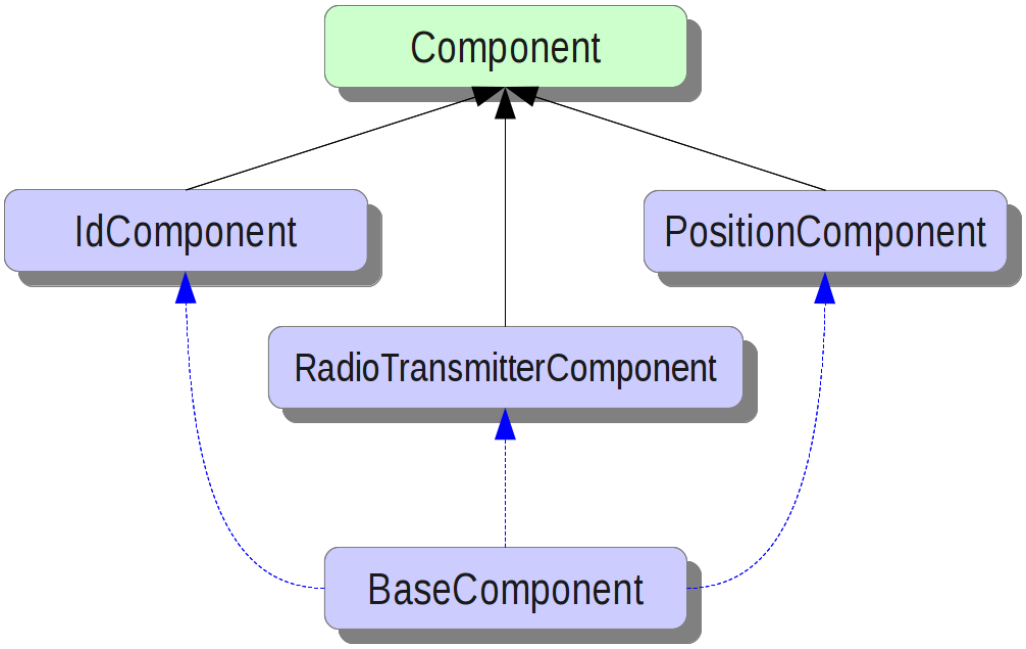

Figure 9.2. A visual representation of components that use composition. Each component type implements the COMPONENT interface, but assumes the default component behaviors by containing an instance of BASECOMPONENT (composition is represented here with a blue dashed line).

cific COMPONENT types, such as POSITIONCOMPONENT and RIGIDBODY, would provide their own specific behaviors.

We needed a way to make all COMPONENTS derive from some base type without duplicating all of the logic for managing the containers. Using inheritance would mean creating a base type that implemented all container logic and having each component subclass that type. This achieves our objective, but with a few caveats. For one, making deep copies of the data within the base type can be tricky. We also have the issue that components will no longer be able to inherit from any other class, since they will be inheriting from the base type already.

An alternative approach would be to use *composition*, which combines objects as building blocks to derive behaviors. A visual representation of both approaches is shown in Figures 9.1 and 9.2. Here, we would create a class BASECOMPONENT that provided the container logic implementation, and a COMPONENT interface that de-

scribed all of the methods that each component should have. Each component would implement Component and contain a BaseComponent object. All methods that dealt with the container would simply be relayed to the BaseComponent object. This achieves our objective in a way that is more flexible. Components are no longer prohibited from extending other classes, and copying is easier to accomplish.

The drawback of the composition approach is its verbosity. In order to forward methods to the BaseComponent object, each method must be defined in the component and an explicit call to BaseComponent must be made. This adds a lot of bulk to the code, as even very basic components will have to define around ten methods. Complicating this is the fact that due to some technical details of forwarding method calls, some of the arguments passed to the BaseComponent methods must be slightly altered first. This raises the possibility of human error when creating new components, as it is not always easy to spot the difference between a standard call and an altered call when reading through ten method definitions.

Before the original system was redesigned, we ultimately decided to use the inheritance approach. The trade-off came down to implementation flexibility versus user-friendliness for implementers of new Components. We ranked the latter as more important, and reasoned that there probably wouldn't be many situations in which a Component must inherit from some other class. In the case that it does, however, the Component isn't constrained to extending BaseComponent anyway, and could technically use a composition approach on a per-Component basis. That is, extending BaseComponent is a convention, not a requirement.

Better solutions exist. Other languages provide facilities that handle these problems in a cleaner way. The Lisp and Python languages, for instance, each support multiple inheritance. Another candidate is Scala, which supports *traits*. Traits act much like interfaces do in Java. Where interfaces only provide method declarations and are prohibited from containing any implemented methods, traits are allowed to

provide implementations. This construct would be the ideal way to mix in BaseComponent's functionality to each Component class. It is not the same mechanism as inheritance, and so does not prohibit the class from extending other classes, and it doesn't require a slew of delegate methods to forward method calls to a contained instance.

9.5.3 The Java Virtual Machine

Although it appears that other languages might be better suited to some problems than Java, the Java implementation would not necessarily have to be scrapped completely in future projects. Java programs are compiled to byte code that runs on the Java Virtual Machine (JVM) (Oracle, 1995a). Many other languages exist which also compile to JVM byte code, and can easily interoperate with Java programs. These include Scala, mentioned above, and Jython, an implementation of the Python language.

Because these languages can easily interoperate with Java and other JVM languages, parts of the simulator could be rewritten in whatever language makes the most sense for the problem being solved. For instance, Jython could be used to perform AUVSML deserialization, while Scala could be used to implement the entity/component architecture.

9.6 Conclusion

The simulator, originally conceived of to incorporate MaineSAIL's robots in simulations of underwater environments, has grown extensively over the course of the project. Over time, it grew to include a complete 3D rigid body dynamics physics engine, a clean network interface, and an interactive visualization tool.

The simulator is currently capable of not only incorporating robots, but clients of any type. Not just underwater environments, but any 3D environment can be

modeled. AUVs can be built from arbitrarily complex components within a system that allows unlimited opportunities for virtual object creation.

Though there is much room for improvement, the project has produced an extensible, well-documented, and capable simulator. The core architecture has gone through many changes, but the end result provides many advantages over previous stages of design. Future work can see not only future improvement, but future extensions to the software.

REFERENCES

Bloch, J. (2008). *Effective Java* (Second ed.). Addison-Wesley.

Bray, T. & Sperberg-McQueen, C. M. (1996). Extensible markup language (xml). http://www.w3.org/TR/WD-xml-961114.html.

Brutzman, D. P. (1994). *A Virtual World for an Autonomous Underwater Vehicle*. PhD thesis, Naval Postgraduate School, Monterey, California.

Christian, W. (2007). *Open Source Physics: A User's Guide with Examples* (First ed.). Pearson / Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (First ed.). Addison-Wesley Professional.

Ganovelli, F., Dingliana, J., & O'Sullivan, C. (2000). Buckettree: Improving collision detection between deformable objects. Proceedings of SCCG2000: Spring Conference on Computer Graphics.

Gould, H., Tobochnik, J., & Christian, W. (2007). *An Introduction to Computer Simulation Methods: Applications to Physical Systems* (Third ed.). Pearson / Addison-Wesley.

Kirmse, A. (2004). *Game Programming Gems 4* (First ed.). Charles River Media.

Neumann, E. (2004). Myphysicslab - runge-kutta algorithm. http://www.myphysicslab.com/runge_kutta.html.

Oracle (1995a). About the java technology. http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html.

Oracle (1995b). Trail: The reflection api. http://docs.oracle.com/javase/tutorial/reflect/index.html.

Russell, S. & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (Third ed.). Prentice Hall.

Shoemake, K. (1985). Animating rotation with quaternion curves. SIGGRAPH '85 Proceedings of the 12th annual conference on Computer graphics and interactive techniques.

Witkin, A. & Baraff, D. (1997). Physically based modeling: Principles and practice.

APPENDIX
SUMMARY OF AUVSML MESSAGES

AUV Simulator Markup Language, or AUVSML, is used in all communication between server and client. We present a summary of all AUVSML messages currently used with the simulator. Note that when referencing a particular element type, we will refer to the counterpart Java data object's name. For instance, instead of referring to <PAUSE-SIMULATOR-COMMAND>, we will write PAUSESIMULATORCOMMAND. This is to maintain coherence with the rest of this thesis, which typically uses the data object names.

A.1 Simulator Commands

Clients register to listen to events generated by world objects by sending a LISTENERREGISTRATIONREQUESTCOMMAND to the simulator. It contains a list of which world objects to listen to and which events to listen to for each world object. To listen to all events, the type *ALL* is given. To unregister, the REGISTER field is set to 0.

```
<auvsml>
   <listener-registration-request-command
      register="1">
      <world-object id="52">
         <listener type="*ALL*" />
      </world-object>
      <world-object id="47">
         <listener type="*ALL*" />
      </world-object>
   </listener-registration-request-command>
</auvsml>
```

```
<auvsml>
   <pause-simulator-command />
</auvsml>

<auvsml>
   <resume-simulator-command />
</auvsml>

<auvsml>
   <shutdown-simulator-command />
</auvsml>
```

The simulator can be paused, resumed, and shut down remotely using PauseSimulatorCommand, ResumeSimulatorCommand, and ShutdownSimulatorCommand, respectively.

A.2 World Object Messages

World object commands operate directly on world objects. These include the ApplyDirectForceCommand and the RemoveForceCommand. ApplyDirectForceCommand applies a force onto an object such that no torque is induced. RemoveForceCommand removes a force by name.

```
<auvsml>
   <apply-direct-force-command
      world-object-id="47" name="Gravity" orientation="WORLD">
      <force-vector x="0.0" y="0.0" z="-9.8" />
   </apply-direct-force-command>
</auvsml>

<auvsml>
```

```
   <remove-force-command world-object-id="47"
      force-name="Engine Thrust" />
</auvsml>

<auvsml>
   <entity-added-message timestamp="1.2" entity-id="47" />
</auvsml>

<auvsml>
   <world-object-added-message timestamp="24.7"
      entity-id="47">
      <position x="14.0" y="-89.0" z="22.0" />
   </world-object-added-message>
</auvsml>
```

The server will send an EntityAddedMessage when an entity has been added to the system. If that entity is also a world object, it will send a WorldObjectAddedMessage, which includes the world object's position.

A.3 Movement-related Messages

The server notifies clients that a world object has moved with a MovementStateChangeMessage. It tells the client which object moved, when the movement occurred, and whether it was an actual state update or a projected update (see Chapter 7 for details). It also gives the object's position, its orientation as a matrix, its linear momentum and its angular momentum.

```
<auvsml>
   <movement-state-change-message
      timestamp="7.8" world-object-id="46" is-projected-state="0">
      <position x="1.0" y="2.0" z="3.0" />
      <orientation xx="1.0" xy="0.0" xz="0.0"
```

```
                    yx="0.0" yy="1.0" yz="0.0"
                    zx="0.0" zy="0.0" zz="1.0" />
      <linear-momentum x="3.0" y="4.0" z="3.0" />
      <angular-momentum x="0.0" y="0.0" z="0.0" />
   </movement-state-change-message>
</auvsml>

<auvsml>
   <timestep-advanced-notification
      timestamp="54.6" state-type="ACTUAL" />
</auvsml>

<auvsml>
   <different-position-feedback-message
      world-object-id="47" timestamp="5.6"
      handler-name="DirectPositionChangeFeedbackHandler">
      <virtual-position x="1.0" y="2.0" z="3.0" />
      <actual-position x="0.98" y="2.1" z="3.4" />
      <old-position x="1.0" y="2.0" z="2.0" />
   </different-position-feedback-message>
</auvsml>
```

The server sends a TimestepAdvancedNotification whenever it has finished processing a timestep. This is useful for visualization tools to determine when to draw the updated object positions that have been extracted from a series of MovementStateChangeMessages.

A client will send a DifferentPositionFeedbackMessage when it senses that its physical position differs from its counterpart world object's virtual position in the simulator. It provides the object's ID, the time at which the correction should occur, and the name of the feedback processor, or handler, that should handle the

message (although the simulator can override this). It also provides the virtual position, the actual position, and its position at the previous snapshot. This is provided in case the feedback processor needs to work with the difference in positions between the current and previous snapshots.

A.4 Queries and Responses

The server can be queried about the state of the world and its world objects. A client can send a GetWorldObjectListCommand to receive a list of all world objects currently residing in the world. The server will respond with a WorldObjectListResponse, which contains a list of the ID, type, and name of all world objects.[1]

```
<auvsml>
   <get-world-object-list-command />
</auvsml>


<auvsml>
   <world-object-list-response>
      <world-object id="47" type="WorldObject" name="Juggernaut" />
      <world-object id="88" type="WorldObject" name="Rock" />
      <world-object id="107" type="WorldObject" name="AUV-1701" />
   </world-object-list-response>
</auvsml>


<auvsml>
   <get-propulsor-list-request agent-id="47" />
</auvsml>
```

[1]Originally, the type field would contain either WorldObject or Agent, with Agents being the only type of object that a client could take control of. Later, the system was redesigned such that WorldObjects also had this functionality, and Agents were removed from the system. Currently, the type field only contains WorldObject, though there is no reason that different types may be added in the future.

```
<auvsml>
   <propulsor-list-response agent-id="47">
      <propulsor name="Rudder" />
      <propulsor name="Aft Thruster" />
      <propulsor name="Port Engine" />
      <propulsor name="Starboard Engine" />
   </propulsor-list-response>
</auvsml>
```

World objects may have any number of propulsors attached to them, which represent propulsion devices such as engines. The client can send a GetPropulsorListRequest to retrieve a list of propulsors for a given world object. The server will respond with a PropulsorListResponse, which contains a list of all of the propulsor names.[2]

[2]Note that agent-id is similarly a relic. Originally, only Agents could have propulsors.

BIOGRAPHY OF THE AUTHOR

James Isaac Pierce Brawn, Jr., was born in Bangor, Maine and graduated from Bucksport High School in June, 2005. He graduated with a Bachelor of Science in Computer Science from The University of Maine in May, 2009 and is a member of the Upsilon Pi Epsilon computing honor society and Golden Key International.

James Brawn, Jr. is a candidate for the Master of Science degree in Computer Science from The University of Maine in August 2012.