2007

AI Loom: a generic development framework for multi-agent systems ideally suited for virtual worlds

Joshua Luke Brown
Iowa State University

AI Loom: a generic development framework for multi-agent systems ideally suited for virtual worlds

by

Joshua Luke Brown

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Mechanical Engineering (Human Computer Interaction)

Program of Study Committee:
James Oliver, Major Professor
Julie Dickerson
Dimitris Margaritis

Iowa State University

Ames, Iowa

2007

DEDICATION

I would like to dedicate this thesis to my loving wife Jennifer, who's support throughout this process has been amazing. Her selflessness and continued devotion to me have made this thesis possible. I could not have done it without you. I'd also like to thank my parents, who's steady guidance and support has always pushed me down the path to success and once again has helped me arive at another milestone. Also, to all of the professors and colleagues that have helped me on my journey. Namely, my committee memebers: Julie Dickerson, Dimitris Margaritis, and James Oliver, as well as my closest colleagues: Chad Austin, Allen Bierbaum, Carlina Cruz-Niera, Patrick Hartling, Andres Reinert, Adrien Sanier, David Sassen, and Ben Scott. Also thanks to all of the Virtual Reality Applications Center support staff and faculty. Finally, I'd like to thank God for allowing me to see all the opportunities and for opening the doors that allowed me to work on and complete this thesis.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

In books such as Snow Crash[1] and movies like The Matrix[2] and the Thirteenth Floor[3] science fiction authors have provided us with the vision of a ubiquitous constantly evolving, constantly growing, constantly available virtual environment that we can interact with. Since the mid-1980s with the beginnings and subsequent evolution of the Internet[4, 5] there has been growing public support for just such a world. The possibilities of the Internet and the visions of Hollywood may have brought the vision of these virtual environments to the masses, but it has long been a dream of computer scientists and engineer to have an interconnected persistent graphical world. In some niche areas there are even very successful examples of virtual environments tailored to specific interests. In the video game industry we have seen resounding successes with games such as Star Wars Galaxies[6] and recently and most successfully World of Warcraft[7]. Also, with a broad enough definition for what constitutes a virtual environment, we could argue that sites such as My Space[8] could be construed as a primitive form of a virtual environment with it's visual spaces representing individual users. The constant connectedness of the growing mobile phone and wireless Internet may one day evolve into a fully virtual interactive medium. There are recent attemps at more general virtual environments on the Internet with websites and applications such as There[9] and Second Life[10] that are just scratching the surface of the potential found in Snow Crash's metaverse.

This thesis covers base research in the construction of an open distributed virtual environment (DVE). For our purposes we define a DVE as having four distinct characteristics. The four characteristics of a DVE are:

1. It consists of computer constructed graphical representation of a world.

2. It has a world representation that is persistent.

3. It allows any user to connect to the world and be represented by his own customized avatar[11] to all other users.

4. It allows any user to add his own content to the world with it's own graphical representation and scriptable behavior that all other connected users can see.

With these four requirements for what constitutes a *distributed virtual environment* we can define more concisely that a DVE is a networked, graphical, persistent virtual space in which any number of users can be simultaneously interacting with the same world and all changes to the shared world's state are propogated in real-time for all other users to see.

The construction of even the simplest DVE presents an enormous engineering challenge. There are actually many challenges presented with this task in a plethora of research fields. We have identified four core and essential problems that must be overcome for the most basic of DVE to be constructed. First, we have the problem of *scale*. Millions, or even billions of simultaneous connections into the virtual environment presents a unique and challenging computer network problem. Secondly, we must address the problem of *generality*. The virtual environment must support nearly any behavior. Since we have no criterion for why users are participating in the virtual environment, then we have no criterion for knowing what actions users may need and therefore we mush allow nearly any action. Thirdly, we have the problem of *persistence* and potentially animated non user controlled objects. This means that in a DVE, we would be required to support non-user controlled objects; objects that can be interacted with by users and when interaction occurs be able to perform actions which may or may not change the global shared state of the DVE. These three challenges are not, by any stretch, the only challenges faced by one who would like to construct a distributed virtual environment. Other more specific problems may include the rendering of the virtual environment on any device, the physics representation and object interaction, network security and cheating as well as many non-engineering related challenges such as the economic feasibility[12] or the social dynamics[13] of a distributed virtual environment.

The first three main challenges we have considered are all formidable challenge, but the last of our major challenges is the most significant. The remaining challenge is the construction

of the world itself, by definition the virtual environment is an infinite space, and therefore filling it with content ourselves would be impossible. To give some concept of the time scale to build a large virtual environment we look to the video game industry and Blizzard. It took Blizzard Corporation© over 4 years with over 300 dedicated individuals to create the World of Warcraft[14] and it is a very specific online environment with very specific rules and it does not scale beyond a few tens of thousands of users simultaneously connected. Therefore if we are to populate an infinitely large world we must turn to another possibility. We are forced into constructing a framework capable of such a world and then rely on a community of users to form the world itself. We call this idea of user addition *open addition* and this can be equated to how the Internet works, anyone can add his own web page and google[15] now indexes over five trillion pages. Constructing a usable framework for anyone to add to is perhaps an even more troubling realization than any of the challenges we've considered so far. However, it is a necessary requirement that our world support an open ended mechanism for community addition in order for it to be at all plausible.

Taking these four problems into account: generality, scale, persistence, and openness we are presented with an interesting cross section of problems. At first glimpse they seem like a disjoint set with little similiarity, but we believe a single clever software architecture may be able to address all of these problems collectively. The rest of this text and our presentation of this research looks into the feasibility of constructing a Distributed Virtual Environment with a single software architecture which addresses these four concerns through a generic distributed multiagent system (MAS).

1.1 Statement of Purpose

The purpose of this research is to investigate the usefulness of a distributed multiagent system as a framework for a DVE. The long term goal of our research is to create a complete, fully usable distributed virtual environment. Our work up to this point addresses a subset of the problems associated with constructing a distributed virtual environment, namely the problems of network scalability, persistence, generality and open addition as described above.

Our research has yielded two substantial development projects which we introduce in this thesis. We first introduce *AI Loom* a framework that lends itself to the development of a distributed multiagent system in a simple yet highly extensible software architecture. We also introduce *Plane*, a prototype distributed virtual environment built on the AI Loom architecture which demonstrates some of the characteristics of a DVE.

We have hypothezised that a DVE could be constructed from an appropriately designed multiagent system. Because of the inherent dependency in our hypohtesis that a DVE must rely on a MAS we first researched and created our own multiagent system AI Loom, and tested and evaluated it's performance and usefulness in a variety of applications. We then built Plane on top of the AI Loom architecture. The purpose of this thesis is to present the details of our research and development of both AI Loom and Plane, and present our findings on the usefulness of using a multiagent system as a framework for a distributed virtual environment.

1.2 Scope of Research

This research is comprised of several stages which we outline below. As we already mentioned, the long term goal of our research is to create a complete DVE. In working toward this goal we broke our research down into four parts:

1. Researching multiagent systems, and potentially building a MAS architecture suitable to our needs.
2. Using the MAS architecture from part 1 to build a prototype DVE demonstrating it's feasibility.
3. Evaluating the prototype DVE to discover and define all of the remaining challenges.
4. Constructing a complete DVE based on evaluation in part 3 and a potential refactoring of the MAS from part 1 to accomidate the required features.

At the time of this writing we have conclluded the first two parts of our research. Therefore, this thesis is primarily about our work in constructing AI Loom, our own multiagent system, and then on the design and development of our prototype DVE, Plane.

The work we have completed on our research through the first two parts can be broken down into four phases that we passed through, which we enumerate below:

1. Perform requirements gathering on intelligent agents, distributed intelligent agents and of virtual environments.
2. Analyze existing technologies, tools, and techniques for developing distributed intelligent agents as well as for building virtual environments.
3. Implementation of AI Loom, our own MAS, based on requirements and past technologies.
4. Implementation of Plane, our own DVE, built on top of AI Loom.

These four stages are presented in this thesis in the following chapters:

- Chapter 2 covers background material and further definitions of AI, agents, distributed computing, and virtual environments.
- Chapter 3 satisfies stage 2 by presenting previous and related work in multiagent systems and in distributed virtual environments.
- Chapter 4 covers stage 1 by defining specific requirements and goals for AI Loom.
- Chapter 5 and 6 satisfies stage 3 by laying out the design and implementation details of AI Loom and our evaluation of AI Loom's design based on our goals established in chapter 4.
- Chapter 7 satisfies stage 4 of our research by presenting our design and implementation of Plane.
- Chapter 8 conclude our research by presenting our findings along with our analysis of the developed systems and discussing further research areas.

Chapter 2. Background

In this chapter we present some general background and history on artificial intelligence(AI), artificial intelligent agents, and distributed systems. Then we discuss more specifically the details and history of distributed intelligent agents and of virtual environments. We continue our background discussion into chapter 3 with the discussion and comparison of existing tools and software in these fields as well as covering related research and methodologies.

2.1 General Background

2.1.1 Artificial intelligence

2.1.1.1 Brief history

Artificial intelligence in the computing form is generally considered to have begun with the early computer pioneers. Alan Turing, Von-Neumann and Shannon were early experimentalists in AI. Many consider the beginning of AI as a field to have "arrived" with Turing's philosophical paper from 1950[16] where he proposed the famous touring test. This is not to suggest that Turing began the research, in fact many methods used in today's "state-of-the-art research" were first proposed in the mid 1940s.

From those first papers in the early 1940s through the 1970s there was great promise and great expectations for AI as a field. Most of the research through this era was devoted to symbolic solutions in which problems were split into knowledge representation and reasoning algorithms. This split forced researchers into a trade off, more information and slower algorithms, or less information with faster algorithms. This split between the knowledge and reasoning is still prevalent in the field however, as Millington points out in his book Artificial

Intelligence for Games[17], the understanding between the trade off has been lost by many researchers. As frustration grew over the slow progress of AI we find that in the mid 1980s there was a shift to using organic solutions for AI problems. This was a return to some of the earliest days in AI research where neural networks and decision trees were presented in primitive forms. These solutions have now been regurgitated and have produced some success, but progress is still slow.

2.1.1.2 What is AI

There is no well agreed to definition of what AI is. In the 1995 textbook Artificial Intelligence A Modern Approach, Russell and Norvig[18] create a taxonomy for categorizing AI into four distinct groups. The four categories are listed below with definitions of AI supplied from various sources for that categorization:

1. Systems that think like humans – "The exciting new effort to make computers think ... machines with minds, in the full and literal sense."[19]
2. Systems that act like humans – "The art of creating machines that perform functions that require intelligence when performed by people." [20]
3. Systems that think rationally – "The study of mental faculties through the use of computational models." [21]
4. Systems that act rationally – "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes." [22]

These four categories vary along two dimensions. Do the systems think or act and are they pursuing human behavior or rational behavior? Let's briefly explore each of these categories.

Acting human : the Turing test Perhaps the most academically famous artificial intelligence test is the Turing Test, developed by Alan Turing in 1950[16]. His test was designed to test if a machines qualified as intelligent. Turing defined intelligence by a computers ability to sufficiently fool an interrogator. Basically the Turing Test was to put a computer at one

end of a teletype and a human at the other. After 5 minutes if the human could not tell if he or she were communicating with a human or computer then the computer passed the test.

Through the years there have been other proposed tests such as a computer beating a human at chess[23]. Brian Schwab[24] points out that by the most broad definitions, even house thermostats exhibit intelligence because they are "acting human" when they decide to turn on the air conditioner to lower the temperature in the room. Of course this is far too broad of a definition to be useful. For our purposes the acting human category is fairly straight forward, although we might not have the specific test proposed by Turing we know if a system is, or is attempting to behave like a human.

Thinking humanly The field of cognitive science is what this category has evolved into. Cognitive Science attempts to bring together computer models of AI and experimental techniques from psychology to try to construct theories of how the human mind operates. This field has many theories as to how computer AI may one day mimic the mental capacity of humans in order to think as we do. The field came into it's own approximately 50 years ago and many of the proposed early theories are stilll in debate such as [25, 26, 27, 16], however these early works were all observational, experimentally, or theoretically based with little to no medical or biological backing of how humans think. Today we are progressing rapidly into discovering how our brains store information and how we process information which is leading to some exciting discoveries into the human thought process.

Thinking rationally The idea or concept of logic is the primary concern of agents that aim for thinking rationally. Rational thinking can be thought of as thinking correctly at all times or thinking perfectly, therefore logical connections between known entities can lead to new connections which can be seen in learning. For example, "Socrates is a man; all men are mortal; therefore Socrates is mortal."

Acting rationally In contrast to the acting humanly, this categorization is more concerned with rational actions, which is often considered to be acting "correct." Of course this

defining what is correct for actions is often difficult to define. This is an exciting category of AI and recent research has gone in multiple directions such as investigations into emotional agents[28, 29, 30, 31]. Some researches will argue that this goes beyond acting rationally however in this brief background we incorporate emotion as well as research into beliefs[32, 33] into this category.

Because of the general nature of the acting rationally category it has come to the forefront in most current research as the end goal for an intelligent agent. In video games and other media where users interact with intelligent objects this is certainly the case. Therefore, we determine just as Russell and Norvig[18] do that no matter what the specific goal for an artificially intelligent entity, the more general aim of the entity in one form or another is to act rationally. For the purposes of our research and the remainder of this thesis we consider AI to deal exclusively with this category of AI.

2.1.2 Intelligent agents

Up until now we have considered artificial intelligence as a nebulous field and we have classified it into areas of interest. However we have not given AI substance yet in the physical sense. This is where intelligent agents come in. For the purposes of our research we define an intelligent agent as any entity capable of observing an environment and effecting it's environment through actions. This is a derivative of the definition given by Wooldridge[34] and is explained best by presenting his diagram presented in figure 2.1. This definitions suffices for us, however the term intelligent agent is not well defined and there are nearly as many definitions as there are publications on intelligent agents, as demonstrated by the wide variety in [35, 36, 18, 37, 38, 17]. As we concluded in the last subsection we will limit our discussion to rational AI and that applies to intelligent agents as well; that is, for the purposes of this research we are only interested in intelligent agents that act rationally. As an example, if we were modeling a human with an intelligent agent we would incorporate ears and eyes as sensory mechanisms and we would give it the ability to walk and interact with objects via arms and legs.

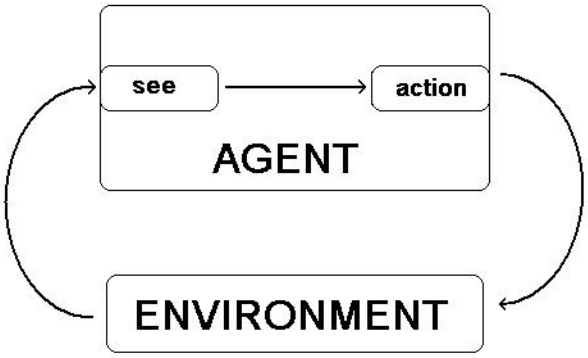

Figure 2.1 Wooldridge's simple agent representation

There are of course many variants on what an agent is and does. There are many examples of learning agents that have been proposed over the years and there is a wide range of applications and research using learning agents today including [39, 40, 41, 42, 43]. There are also planning and strategy agents[44, 45, 46, 47], distributed agent systems[48, 34, 37, 49] and mobile agent systems[50, 51, 52, 53]. All of these classes of agents are subclasses of the more generic intelligent agent field and are solutions to specific problems and they all potentially have different means of sensing and effecting their environments. In the rest of this subsection we discuss what constitutes an intelligent agent and how agent's can generally be broken down into components.

2.1.2.1 Minimal mandatory components

With a vast array of agents it may be hard to see what every agent could possibly need. Russell and Norvig provide one model for determining a breakdown of the characteristics that go into these various different agents. Woodridge in his book[34] also proposes a breakdown that shares the same minimal pieces an intelligent agent requires. As we stated above, all agents must be capable of sensing and effecting their environment and this helps us define an initial minimum set of components that all agents must possess:

Perception All agents in order to solve a problem need to be able to observe some environment. In the case of a software agent this may be entirely a digital representation of an

environment, none the less an observation of the state of the representation is still required. We will define this observation as being done through *sensors*. When an agent observes a sensor result it gains some state information about it's environment. The sensors themselves can be implemented by hardware such as sonar, radar, microphone. or thermostats. Or, purely in software with database queries or algorithmic analysis of an environment representation.

Action In order to solve a problem, or more generally, just to do anything an agent needs to be able to perform some actions on it's environment. This action can really be anything. In most applications of agents the actions an agent is capable of are strictly effects on itself, such as movement, however this isn't always the case. We define an action in an environment that is initiated by an agent to be an *effector*. Agents typically have a set of effectors from which the perform actions. In the simplest case there is one effector and the agent makes a binary decision to use the effector or not.

Decision In order for an agent to act rationally and use it's effectors it must make decisions. The decision making process, or selection, is encompassed in a component of all agents called a *performance element*. A performance element must make decisions on which effector that it would be best to use, this is almost always done based on the agent's perception of it's environment.

In Brief, an agent requires some means to observe it's environment, some set of actions that it can take and a decision making mechanism to select which action to perform based on the perceived environment. As a formal explanation for these minimal components we say that an agent can be in a finite set of states with in his environment $S = \{s_1, s_2, ...\}$. Also an agent has a finite set of actions $A = \{a_1, a_2, ...\}$. From these two sets we can define a mapping function:

$$f(env) : S \times A \rightarrow \delta(S)$$

This function maps the current state of the agent in the environment $s \in S$ and an agent's action $a \in A$ to a new set of environment states $env(s, a)$, that is, all states that could result from performing action a from state s in environment env.

2.1.2.2 Optional components

The above components define the minimum components required to define all agent. However it behooves us to break the decision making process down into more specific components and although this is strictly not a necessity, it makes dealing with an agent architecture significantly easier. Therefore we propose the following optional components for an agent:

evolution Many agents, in order to make their decision attempt to learn or evolve with their perceived environment. We call this learning mechanism the *learning element*. For over 60 years (and especially in the last 15) there has extensive research into learning and applying learning techniques to agent systems in various systems. Popular learning component used in many applications are neural networks[54, 55, 56, 57, 58, 59, 60, 24, 17, 18] and decision trees[18, 17, 61, 62]. It's important to note that most learning algorithms require their own internal representation of the agent's perceived world. It is important to note this because it implies that the agent translates it's sensor input into a secondary representation which can be stored and evaluated over time, thus requiring a data storage mechanism in addition to a processing mechanism for an agent.

monitoring If an agent is trying to become better at solving a task then occasionally it may want to know if it is doing better or worse than it was doing before. Although it may just be told what is better or worse, there are a number of use cases in which an agent can monitor itself and evaluate it's own performance. These agents can then make decisions to evolve or update in an effort to make even better decisions. This monitor of decision making performance is commonly called a *critic*.

exploration An agent that needs to explore or that desires to explore new solutions or paths would need another component. This is useful to agents that evolve that have reached a local maximum of decision making performance but still need to discover true maximums. This type of component is called a *problem generator*, and is responsible for suggesting actions that would lead to new experiences. In it's simplest form this can be achieved by randomly

selecting effects to perform but usually some additional logic is used to determine experimental effects.

2.1.2.3 Putting all the components together

The first three components, the minimal set required to define an agent are sensors, effectors and a performance element. In addition we defined a learning element, a critic and a problem generator which help facilitate various functionality within an agent. In total, these six components are used in various different combinations and in different ways, and can create almost all types of the "thinking rationally" agent type. However, one broad category of agents however that cannot be modeled with these 6 components is an agent that directly communicates with other agents or more specifically distributed agents. Before we can analyze the needs of a distributed agent we would be remiss if we did not look at the broader field of distributed system for a basic understanding of the problem domain distributed agents fall into.

2.1.3 Distributed Systems

2.1.3.1 What is a distributed system

In their introductory textbook Distributed Systems, Principles and Paradigms[63], Tanenbaum and Steen loosely define a distributed system as, "A collection of independent computers that appears to its users as a single coherent system." We agree with this definition and it is a generic enough definition to also agree with most texts on distributed systems. According to Tanenbaum and Steen's text, there are four general goals of distributed computing and they are:

Connect users to resources Connecting users and resources is really the primary goal of distributed computing. There are several aspects that need to be addressed when considering connections between users and resources, the main concern being security.

Hide the fact that resources are distributed Hiding the distribution of resources is called *transparency*. There are various forms of transparency which refer to different ways in which we hide information from users.

Openness To have an open distribution system means to have a system that has a standard way of interfacing with the system so that it becomes easier to exchange resources.

Scalability In distributed computing this is a key concept. We want it to be easy to add more computers to the distributed system without having to do anything so we can grow and shrink the system with our needs.

2.1.3.2 Characteristics of distributed systems

Distributed systems can be categorized using the four above properties. There are many design challenges and decisions to be made in developing a distributed system. We outline the major decisions here.

Client server vs. peer-to-peer The first and most significant determination to make is whether the system will be client-server based, a peer-to-peer system or some hybrid of the two. Subramanian[64] define the term *peer-to-peer* (P2P) to mean in a network of equal peers using appropriate information and communication systems, two or more individuals are able to spontaneously collaborate without necessarily needing central coordination. Traditionally, peer-to-peer systems were not feasible for a multitude of reasons, not enough or powerful enough computing resources, network bandwidth constraints, peer negotiation, peer detection and network security are all barriers to creating peer-to-peer systems. Therefore, traditionally distributed computing focused on a client-server methodology and it's interesting that even in modern texts on distributed systems, like from Tanenbaum and Steen, there is no mention of peer-to-peer solutions. However, in the past few years there has been a surge in research and corporate development of peer-to-peer based technologies.

The surge in P2P research has stemmed from the two rapidly emerging areas of mobile computing and peer-to-peer file sharing. With technologies like Kazza. Napster[65] and Bit Torrents[66] we have seen a surge in average computer users utilizing vast P2P networks and it has spurred some very interesting research in network topologies and in peer negotiation and discovery. Likewise, the mobile computing field has exploded with smart phones and PDAs and has produced some very interesting research of it's own.

At this point, we have a genuine decision to make that was not as easily available in the past and that is if a distributed system should be client server based or P2P. There are benefits and shortcomings of both system. The main benefit of a server based system is that one computer monitors and controls accesses and resources on all other machines, providing fault tolerance and dispute resolution, security and general stability of the topology of the network are major benefits that the P2P model struggles with. However, P2P systems provide nearly unlimited scalability where client servers are bound significantly by the bottleneck in communicating with the server. We are therefore presented with an interesting trade space to consider in the design of a distributed system between scalability and stability.

Communication models The next design decision we need to make is what communications model is appropriate. There are several to choose from including: Remote Procedure call (RPC), Remote Object Invocations, Message-Oriented Communications, and Stream-Oriented Communication. Each of these has it's own benefits and shortcomings. In RPC a process on machine A makes a call to a procedure that can run on machine B. The caller on machine A halts while machine B runs the procedure, with no message passing visible to the programmer.

RPC[67, 68] accomplishes this by automatically generating stub implementations and an empty interface for the programmer to fill in. When the programmer has completed his code the application works as follows: when the client makes a call to a remote procedure it actually makes a call to the stub implementation which sends the parameters to the stub at the server side and then blocks waiting for a response. The server implementation gets the parameters from it's stub, runs it's procedure, and returns the result to it's stub. The server stub then sends the result back to the client stub which passes the result on to the client. All of this

hidden from the programmer.

In an object oriented language we have the ability to create objects and distributed computing takes that encapsulation of objects and uses it in a distributed setting by placing objects on different machines and then hiding the calls from one object to the next. All objects have methods to operate on the object which are defined in interfaces. By having a separation between interface and method we can place the object and all it's methods on one machine and the interface to that object on other machines. When a client binds or connects to a distributed object that object's interface is passed to the client in what is called a proxy. A proxy is similar to an RPC stub, managing communication between the client and the distributed object. As in RPC there is a server side stub as well called a skeleton which handles all the communication at the server side. The calling of remote objects methods is generally called remote method invocation (RMI).

In addition to RPC and RMI there is message passing communication. There are two situations to consider when dealing with message oriented communication: when the receiver is running and when the receiver is not running. There are several message passing paradigms for when receivers are running and there are message queuing paradigms for when the receiver is not running. The Message Passing Interface (MPI)[69, 70] standard was developed for hardware independent communication which allows developers to easily send and receive messages between machines. This works well for transient communication where if the receiver of a message cannot be reached then the message is dropped. For persistent communication we would need a message queuing model where messages are queued and stored until they can be delivered to the recipient.

Finally Stream oriented communication gives us support for continuously connected media. This is particularly useful in distributed applications for real-time information such as graphics or sound which strive to update continuously. Most streaming communications conform to the token bucket model where an application sends data to a bucket and the bucket sends out a regularly timed stream of data.

The communication mechanism used for a distributed system dictates the entire archi-

tecture of the system and is nearly as important as the decision between peer-to-peer and client-server systems. These two decisions are the most important, the next most important property of a distributed system is how much transparency the system has.

Transparency There are eight forms of transparency in distributed computing[71], each dealing with how to hide distribution of resources from the user:

- Access - Hide how resources are accessed
- Location - Hide where a resource is located
- Migration - Hide the movement of resources
- Relocation - Hide the movement of resources while they are in use
- Replication - Hide the duplication of resources on multiple systems
- Concurrency - Hide the sharing of resources among processes
- Failure - Hide the removal and addition of resources
- Persistence - Hide whether a resource is on disk or in memory

Developing distributed systems requires that one balances the trade-off between performance and transparency. In general the more transparent a system is to the user the worse it performs. So we must be careful in determining which forms of transparency are important to our applications and what forms of transparency must be avoided to harness performance thus forcing the user to deal with directly.

Security measures Determining security levels is also an essential component to all distributed systems. Because distributed systems pass information and resources between machines there is an inherent risk associated with them. Security levels directly effect the openness that distributed systems try to accommodate. Therefore, a balance must be struck between a system's desire to expand and how easily it shares information. There are many

security aspects to consider and we couldn't hope to cover all the security mechanisms here but some of the more important concepts in security are: encryption, firewalls, access control lists, and secure keys. Of course this isn't a complete listing and an explanation of security measures inside or outside of distributed computing is outside the scope of this thesis. Although this thesis does take security measures into account briefly it is not a primary concern of our research.

2.2 Focused Background

Now that we have built up a basic understanding and set definitions for AI, agents and distributed systems we can use this basis to look at fields more specific to our research, namely: multiagent systems and virtual environments. We look closely at multiagent systems and virtual environments here to build a basic understanding of our research topic and in the next chapter we look at related research and technologies.

2.2.1 Distributed intelligent agents

In just the last few years distributed intelligent agents or multiagent systems (MAS) have become their own research topic. Jacques Ferber released his book: An introduction to Multi Agent Systems[37], which jump started the distributed agent realm of research and it has spread it's roots into many areas of research. We have seen many publications multiagent systems and nearly every one has it's own definition for the term multiagent system, as a sampling of the multitude of definitions see[38, 37, 48, 35, 34] In this section we look at what a distributed agent is and discuss how the distributed computing trade-offs we presented in the last section impact distributed agents.

2.2.1.1 What is a distributed agent

Because there is such disparity in the definition for distributed agents in current literature, we propose our own very course definition for a distributed agent as an extension of a rational intelligent agent such that it does everything a rational agent does by accepting input, selecting

an action and performing that action. But, a distributed agent has the added requirement that it can communicate with other agents. With this open-ended definition we are leaving the design decisions of what is communicated between agents and how communication is transferred between agents to the developer of the distributed system. By having a considerably broad definition of what a distributed agent is and what it does we leave ourselves with a wide range of possible applications for distributed agents. This is nice for general purposes, but at the same time this requires careful analysis of the problem domain that each distributed agent is being applied to before a system can be successfully constructed to meet it's design goals. Throughout this thesis we will refer to distributed agents and multiagent systems interchangeably. This is not the case in the all literature, as some texts say that a multiagent system does not need to be distributed and that such systems should be categorized as distributed multiagent system. However, for the purposes of our research we treat these terms synonymously unless otherwise noted.

2.2.1.2 Characteristics of distributed agents

Since distributed agent systems consist of specific implementations of distributed systems and specific implementations of rational agents we can easily conclude that the characteristics of distributed agents derive from these two categories. Therefore there are many implementation decision in the creation of distributed agents in general and regarding the context in which the multiagent system is used.

In order to implement a distributed agent system one must carefully analyze the goals to which the system is working to achieve. There are several key trade-offs to consider, as well as several more subtle yet still important aspects to closely inspect, all of which are derivations from our discussion in the above two sub-sections on distributed systems. Below we analyze these trade-offs and how they are related to the goals of distributed agents.

Peer-to-peer or client-server based This decision has many implications not only for how the system handles control of the system and communication, but also for many other components in the agents. A peer-to-peer based solution is ideally suited to problems of scale

and works well for a very large number of agents. There is significant research today regarding communication and communication propagation around a peer-to-peer network which show promise that many of the associated problems in peer-to-peer systems are solvable. In contrast to P2P systems, client-server based solutions do not scale as well due to communication bottlenecks, but they are much easier to develop because control for the entire system resides with the server not spread across the network.

Also this decision affects how communication is passed between agents and what information must be contained in these messages. Client-Server solutions generally have a more simple communication protocol because some management of messages can be handled by the server. Whereas peer-to-peer solutions have no means of managing communication other than in the message that is being passed.

Distributed objects Distributed objects are the natural decision for the communication system in a multiagent system. Because agents are objects themselves applying the distributed object methodology is well suited for agents. However this requires some form of communication agreement between all agents. So the decision then becomes how open does the system need to be. Will there be new agents added to the system regularly. If that is the case then special attention is needed to ensure that the interface standard for the distributed object is easy to use and well thought out, doing everything possible to not inhibit the developers of agents for the system. The Foundation for Intelligent Physical Agents (FIPA)[72] is attempting to create a standard protocol for agent communication which uses a distributed object framework.

Transparency The level of transparency in a distributed system is another key decision. Many agent architectures go to great lengths for distribution transparency, this is evident in the FIPA standard. In developing a distributed agent architecture one needs to evaluate the goals of the system and determine what levels of transparency are appropriate for both the developer of an agent and the end user of the agent system.

Are the agents going to have learning capabilities This is an essential determination for all distributed agent architectures. This will determine how complex the agent is. Generally, using non-learning agents, there is a significant reduction in communication. This is because agents don't change; they change their state or make decisions and they broadcast this information. Also in non-learning systems generally there is a reduction in processing in an agent. If the agent is not learning new things then it generally has less to compute in a given time interval and therefore can respond quicker. If the agents in a system learn then generally they take longer to process and they need to communicate more information, and at the very least they need to communicate what they are capable of learning. All of this is important for figuring out if the system needs a communication language between agents or whether a distributed agent system is necessary at all. This decision further affects what communication model the system uses and whether it's peer-to-peer or client-server based.

The list of above key decisions is far from complete and depending on the system that is being designed other considerations may become more important while these decisions become insignificant, but for the vast majority of scenarios these four decisions are an excellent starting point for deciding how a system should be implemented.

2.2.2 Virtual environments

In the introduction to this thesis we briefly defined what a distributed virtual environment was for the purposes of this thesis. In this section we look deeper into what a virtual environment is and then at distributed virtual environments and existing software and technologies.

2.2.2.1 Distributed virtual environments

In Richard A Bartle's book Designing virtual worlds[73], he defines a virtual environment (or world) as follows: "Virtual worlds are implemented by a computer (or network of computers) that simulates an environment. Some - but not all - the entities in the environment act under the direct control of individual people. Because several such people can affect the same environment simultaneously, the world is said to be shared or multi-user. The environment

continues to exist and develop internally (at least to some degree) even when there are no people interacting with it; this means it is persistent." This matches very closely the definition that we put forth for a distributed virtual world and is a fairly common definition. Allen Bierbaum's definition in his thesis on VR Juggler[74] is somewhat more ambiguous, he says: "[an] immersive virtual world [is] where the user makes use of natural interaction methods to control the application. By definition, this means that the application must be interactive, immersive, multi-sensory, and synthetic." And in their paper on avatar manipulation, Hartling [75] say virtual worlds are defined as, "[being] characterized by the use of three-dimensional graphics rendered by one or more cmputers to create interactive, immersive spaces." This definition is perhaps the most generic definition and leaves the most room for interpretation, and is also a common definition found in relavent literature.

All of the these definitions are similar and agreeable. For the broad definition of what a virtual environment is we prefer Harling's definition as it is the most generic. By his definition any three dimensional computer generated graphical space with user interaction can be considered a virtual environment. This leaves many opportunities open to being called virtual environments. Any 3D place that can be imagined and programmed can be a virtual environment and indeed many different virtual environments have been constructed. Just a few examples which demonstrate the vast array of virtual environemnts that are in existance include 3D video game like EA Sports' John Madden Football and Bethesda Softworks' Oblivion, virtual reality environments, software simulation software, flight simulators like Microsoft's Flight Simulator and even some websites that have Macromedia Flash©3D landscapes like. This broad definition of a virtual world is compelling in that it leaves the door open to so many possibilities.

Although we like Hartling's definition as it applies to the greater research field of virtual environments, we need a narrower scope for our research to help us better define what our goals are. In our introduction we defined a distributed virtual environemtn for the purposes of our research. We said that a DVE is a networked, graphical, persistent virtual world in which any number of users can be simultaneously connected experiencing the same world and be changing the shared world's state for all other users to see. Our definition is a significant

reduction in scope from what constitues a virtual environment compred to Hartling's. It adds the requirement that the world be persistent; meaning that the world's state continues to exist and potentially evolve even when user's are not connected. It also adds the requirement that the virtual world be networked and shared across a distributed network. By requiring that any number of users can connect over a network to the virtual world and that any number of users can experience the same environment simultaneously we have greatly reduced the scope of what constitues a DVE. We use this more specific definition for a DVE to help focus our research onto a much more specific problem set. In the next chapter we look at a wide variety of research and technologies that are relevant to distributed virtual worlds as we have defined them here.

Chapter 3. Related Work

The trend in today's high powered personal computer era is toward distributed computing. Significant amounts of research is going into distributed systems such as grid computing systems[76, 77, 78, 79] and there are examples of consumer applications too, such as the SETI@home[80] program. Perhaps someday we'll reach the point where all computer software and systems are built to use distributed computing inherently and we'll have reached the point of having truly ubiquitous computing but for now we must develop software specifically for distributed computing if we wish to take advantage of multiple computer systems. As we've indicated in the previous chapters there is a tremendous amount of research into distributed computing and more recently into multiagent systems.

In this chapter we identify several research projects and commercial products that are related to our research into using distributed agents for a distributed virtual environment. At this point we are unaware of anyone using a distributed agents system to construct a distributed virtual environment and therefore we must leverage technologies and research from similar fields. One area of research and commercial development that most closely relates to our research is the video game industry and specifically from massively multiplayer online (MMO) games. In addition to MMO development two areas of research that have leveraged MASs are mobile phone network technologies as well as P2P file sharing technologies. We will show later that these two fields have an important impact on our research because they deal with very large numbers of users. We start our exploration into related works by looking at some existing technologies developed specifically for multiagent systems.

3.1 Existing Multiagent Systems

In this section we present six existing multiagent systems currently available in a variety of application domains. We present these to help familiarize ourselves with the available technolgoies as well as to provide a sense of the breadth of applications that multiagent systems are applied to.

3.1.1 Multiagent System Engineering (MSE)

The Multiagent System Engineering (MaSE)[81] methodology developed at the Air Force Institute of Technology is a tool for developing distributed agents. It is bound tightly with an implementation of the methodology called AgentTool[82]. This tool gives us a good methodology to implementation mapping enabling us to make a good analysis of the methodology. The MaSE is structured in such a way to help the developer of a distributed agent system to visualize the design process of what he or she may need to do in order to create the agent. Therefore in AgentTool, which gives the user a graphical representation of agents, agent components and communication networks, the developer is given a set of components from which to work. And after assembling components the user has an agent. This is what the MaSE supports - the design process and not as much the design itself. A developer using AgentTool or any other tool on top of the MaSE infrastructure is still required to determine what agents are needed and how to construct them.

3.1.2 Open Agent Architecture (OAA)

As described in the last section there is a tool available called the Open Agent Architecture (OAA)[83] which seeks to develop a methodology for developing distributed agents and there have been several development toolkits that have implemented the OAA. One example of a tool that implements the OAA is the Agent Development Toolkit (ADT)[84]. The ADT provides a variety of mechanisms that support the specification and implementation of individual agents, as well as cooperating communities of agents.

The OAA model seeks to encapsulate six characteristics. These are:

- Open: agents can be created in multiple programming languages and interface with existing legacy systems.
- Extensible: agents can be added or replaced individually at runtime.
- Distributed: agents can be spread across any network-enabled computers.
- Parallel: agents can cooperate or compete on tasks in parallel.
- Mobile: lightweight user interfaces can run on handheld PDA's or in a web browser using Java or HTML and most applications can be run through a telephone-only interface.
- Multimodal: When communicating with agents, handwriting, speech, pen gestures and direct manipulation (GUIs) can be combined in a natural way.

The OAA has many of the same goals as we are striving to accomplish, and many people have acknowledged that the OAA has a good methodology for some distributed agent situations. However there are OAA methodology has choosen to implement a client-server system. As we've said before, this significantly limits the scalability of a system developed using the OAA. Also the system does not specify how an agent should be constructed. It leaves a significant amount of freedom to the developers as to how to develop their agents. Which in some circumstances is acceptable but a definitive agent model is necessary if all agents developed could be considered part of the same distributed system since they would all be able to determine what the others were doing.

3.1.3 Foundation for Intelligent Physical Agents (FIPA)

The Foundation for Intelligent Physical Agents (FIPA)[72] also hosts a potential standard for developing distributed agents. FIPA has a set of proposed standards and schemata for the development of distributed agents. FIPA itself is purely conceptual in nature and has no specific implementation. There are different standards for each component of the distributed system. By partitioning different components of agents into separate standards the entire set of standards which FIPA has developed and grown into a massive specification. There

are hundreds of pages detailing the various standards, and although there are some great general models in the FIPA standard, they must be abstracted out of the complex detail that is supplied. Parts of this model, such as the agent communication layer are used in several research projects such as [85], but as a general model for an entire system we feel FIPA is far too cumbersome.

3.1.4 PUMAS

PUMAS [85] is a framework based on ubiquitous agents for accessing web information systems through mobile devices. This system is an interesting cross-bread of MASs and web information systems (WIS). It's goal is to run on mobile devices such as cell phones where each device contains three agents that help filter content based on the device and the information available from search queries. It's an interesting case because agents are not communicating with each other necessarily, they are interfacing with the WIS directly and not affecting the content in any way. Because this is a primarily access only system is not well suited to our research but it is a nice ideas on how to manage content on an individual user basis.

3.1.5 FRAGme2004

The FRAGme2004[86] framework is very much similar to the system that we are considering at it's network level. It is comprised of a peer-to-peer system with built in tolerance and security systems. However, like PUMAS, it is built for mobile devices and web interfacing. It is also constructed with the Java programming language [87] (a popular choice amongst mobile device software developers) and it uses RMI for it's message passing interface. Java and RMI are interesting choices for mobile devices, but in the general case and certainly in the distributed virtual environment scenario these are not ideal candidates for performance reasons. The FRAGme2004 framework does have interest to us in it's object layer which abstracts away the details of the P2P underlying nodes and allows developers to focus on their applications. One specifically nice feature is the data management and redundancy technique employed by FRAGme2004 which allows for peers to drop out of the network and still maintain

complete data integrity.

3.1.6 Java Agent Development Framework (JADE)

The Java Agent Development framework or JADE[88] is an open source platform for peer-to-peer agent based applications. It uses the FIPA specification for communication and provides a set of graphical tools. The framework allows you to build agents using component-based programming and is reasonably well designed in so far as the API is matured and simple. It has a very generic agent structure which is very favorable. However it is written in Java and as we argued before this is not ideal for our work. It also has low network transparency, requiring developers to deal with communication between agents far more than we would like to see in a multiagent system.

3.1.7 Others

There are many other examples of P2P systems and even P2P agent based system. The frameworks OpenAI[89] and the Microsoft Robotics Studio[90], although not strictly agent frameworks are well known and used toolkits that have been used for creating agents. Also, the SMART agent framework is an excellent concept presented by Luck and d'Inverno as a theoretical framework[38]. And there are also commercially viable toolkits such as Anthill[91] and Skype[92], and mature research frameworks such as SeMPHoNIA by Patkos and Plexousakis[93] and chautauqua[94]. But, all of the systems we found and could evaluate suffered from one of three problems for our research:

1. The implementation was too specific to the field that the MAS was not portable enough to directly apply to a distributed virtual environment.

2. The programming language or the communication layer were not suited to be used by a distributed virtual environment because of it's performance or intrinsic limitations of the language.

3. The proposed framework was purely theoretical or for some reason not available for a thorough enough evaluation to conclude it's usefulness toward our research.

After an exhaustive search of potential research and commercial systems and not finding anything readily adaptable to our goals we began to research other technologies associated with our research for inspiration in our own design. In the next two sections we look at existing technologies in the realm of video games and specifically in MMO technologies but first we look at research and implementations of P2P file sharing technologies.

3.2 P2P File Sharing Technologies

There is no shortage of P2P file sharing technologies available for free to any computer user. Systems such as Kazaa, Napster, bittorrent, and eMule are just a few of the popular systems available. Obviously a raw P2P file sharing system does not map directly into a distributed virtual environment application. However, because of the multitude of examples and research on this topic it serves a useful purpose to review some of the technologies that are being applied in this area. We'll start by looking at the base technology behind file sharing applications and then look at some recent research related to file sharing techniques.

When talking about P2P file sharing technologies it's usually helpful to talk about pure networks and hybrid networks; and although this dichotomy is a very useful one in discussing P2P technologies it is usually only used in the context of file sharing. The dichotomy breaks down networks into two groups, in a pure network all the nodes are the same whereas in a hybrid network some nodes will be given special purpose. This break down is well defined by Schollmeier[95]. Gnutella[96, 97] is the classic example of a pure network, where as most P2P networks today are a hybrid. This dichotomy as it is discussed in file sharing is an important one to our distributed world research because it gives us yet another interesting trade-off to consider in our own design. Pure networks are much easier to develop and maintain than hybrids. However, they suffer from slow searching and unbalanced network structure which hybrid networks deal with nicely.

3.2.1 Existing technologies

In general the way P2P file sharing applications work is that users each have one client on their computer and they share some of the data on their computer with any other computer that also has the file sharing client. There are hundreds of networks and clients to chose from with Kazaa, eMule and Napster being some of the most famous. Most of these application clients have search capabilities built into make searching the network easy. Most of the time this search is done by looking a network index cache. Data is transferred from peer-to-peer and as new peers acquire data it theoretically becomes exponentially easier to find a peer that contains the data or resource that every other peer is searching for. This is the way P2P file sharing worked until 2001. File sharing was a pair matching process, once a client identified another peer that had the desired resource, the two were connected directly and the resource was transferred.

However, that all changed with the introduction of BitTorrent. With BitTorrent any part of a resource could come from any peer making it much easier to download resource. This made transferring resources much easier to anonymize which is very important for secure settings. It also made data redundancy much easier because peers could share the burden of storing data.

3.2.2 Novel P2P file sharing research

There has been some active research in this area and we point out two relevant publications related to our research. The first is Willmott, Pujol and Cortes' use of an agent abstraction in a file sharing application[98]. Interestingly they find that the criterion to use a MAS for file sharing is very similar to the criterion we identify for our distributed virtual environment. They conclude that peers are entirely autonomous, peers are bound to specific set of actions, rational behavior cannot be guaranteed, cooperative behavior cannot be assumed, and collusion is possible and likely. Since this criterion parallels our own research very closely we are content that it also points to using a P2P system.

The second paper that is closely related to our work is presented in a paper by Lee, Kwon, Kim and Hong [99] where they present a new approach to maintaining trustworthiness among

peers in a P2P file sharing system. This and other similar research is reassuring to developers of P2P systems because of the inherent security risks presented by P2P systems. Although for this thesis we are not directly concerned with the security aspects of constructing a commercially viable P2P system, it is something we must be aware of during our own design in order to avoid constructing a system incapable of reasonable security measures.

3.3 Video Game Technologies

The video game industry presents the closest parallel to our research presently available in a commercial product or research project. Some video games match our distributed virtual environment requirements very closely. Massively multiplayer online games (MMOs) along with DVE have the requirements of a graphical representation of an environment, are in real-time, have semi-persistent state of the world, have intelligent simulated objects and require network connectivity to the environment. As we investigated the video game industries we discovered two main distinguishing traits of the distributed virtual environment that games do not have. The first is the scale in number of simultaneous users connected and that games are much more specific in terms of what is an allowed action in the environment. Because of all the similarities we could not pass up the opportunity to look into the video game industry for related technologies.

What we found is that the agent model is a prevalent one in games and there are several massively multiplayer games and architectures available. In fact, there are a few applications in the game market that are nearly exactly the type of application we are attempting to develop such as There[9] and Second Life[10]. Right up front we discovered however, that all video game technology use a client sever paradigm. This is somewhat surprising, as games draw larger and larger audiences we would have expect that a P2P network would be a very attractive alternative. We suspect that the P2P model has not matured to a commercially viable point for the video game industry due primary to security and "cheating" concerns.

Although MMOs and other networked games currently use client-server architectures there are many game AI architectures and agent abstractions that we can draw insight from. In this

section we present several key points from literature on video game software design that will be relevant to our own architecture.

3.3.1 Game AI engine design

In nearly all of the literature on Game AI we saw the same themes arise where authors put emphasis on the same areas of a game's AI system. Mainly:

- All entities in the game are derived from a base agent object.
- Searching and route planning in the environment are critical.
- Debugging and testing are extremely important.
- Fast iteration time is critical.
- AI implementation is very specific and very unique to the game being developed.

Because of all the similarities between creating our distributed virtual environment and video game development we have drawn the conclusion that these points are also important to us. In the following chapters these same themes arise and we use them, in part, as rationale for some of the design decisions we choose.

3.3.2 Agent abstraction

The software representation that is chosen for the agent is also very important. According to Schwab[24] this can make or break the development of a game. He advocates a layered approach where different parts of each agent are responsible for different parts of the processing. Interestingly his breakdown of layers matches very closely with our component model of agents in previous chapters. He suggests a sensor layer, a decision layer and effects layers as well as long and short term planning layers. This fits nicely with our sensors, effectors, the performance element components along with the learning component and critic component. The approach suggested by Schwab is stunning similar to what we have already mapped out for requirements for our agents.

Likewise in AI Game Programming Wisdom, there are no fewer than 8 papers which discuss agents in an AI system that sense their environment, choose their action, and then perform the action. We can feel fairly confident that an abstraction consisting of these components will be ideally suited for our distributed virtual environment.

3.3.3 Commercial tools

There are a few commercial tools available for building intelligent agents such as AI Implant, the Quake Engine and Unreal engines. Also there is some level of support for agents in gamebryo. However these tools suffer from two major problems. Primarily they are expensive and so adopting them by a community of developers is highly unlikely and, other than AI Implant they are tightly coupled with the rest of the game engine making them very heavy weight objects to use.

After exhaustively revieweing all of the prior work and evaluating existing tools we feel we have not found a multiagent system that is suited well enough to the construction of a DVE. Therefore, in the next chapters we present our own multiagent system, AI Loom, which is still generically designed, but is ideally suited for the task of building a distributed virtual environment.

Chapter 4. Goals of AI Loom

After our extensive review of current tools and technology we determined that there did not exist a multiagent system that was suitable to creating a DVE readily available and therefore we decided to create our own multiagent system called AI Loom that was as generic as possible while still being tailored to our goal of constructing a DVE. Constructing a generic multiagent system is a complex and large task and covers many facets of software engineering and therefore there are many trade-offs that must be made throughout the design process. In this chapter we lay out very carefully what the major design goals for AI Loom are in order to guide our design and decisions during development.

The chapter is broken down into 2 parts. The first part deals with the technical goals for our system and is broken down into several sections each discussing one design goal. The second part of this chapter discuss the non technical goals. Primarily, we want to create an open, free and cross platform multiagent framework. We discuss how this is important and how it does impact some of the technical design of our system.

4.1 Technical goals

The technical goals we set for AI Loom are the result of key trade-offs that we identified in the previous chapters and those that will impact our design significantly. Although with a large and complex system many decisions will be required that do not fit squarely into any one of these goals, as a group they will provide the direction for the design from top to bottom. The goals we set are:

- A minimal and simple API

- Pluggable and dynamic components
- Maximum network transparency
- Maximum scalability
- High agent flexibility
- Fast computational performance

The rest of this section is broken down into subsections that discuss each of these goals in detail and how they impact our design.

4.1.1 Minimal and simple API

No matter what kind of development tool is being designed the minimal, most simple, and most intuitive API is important, but for an artificial intelligence development platform this is of critical importance. As developers contribute to the DVE as many of the details of the underlying system should be protected and hidden so developers can focus on their agent's internal workings and not worry about the system as a whole. Also, the smaller the API that the agent architecture provides the less work it should be to create a DVE on top of that architecture thus making it easier for us to test our hypothesis that creating a DVE from a MAS should be possible.

4.1.2 Pluggable and dynamic components

Because of the diverse range of agents that could be developed in a DVE, our system should be flexible enough to allow components of agents to be removed, added or replaced in real-time with ease. The diversity in agent types is a vital feature to our system. In addition to diversity of agent types we also need to support a pluggable architecture because we must also support rapid iteration. In chapter 2 we pointed out how vital this was in the video game industry for games to be successful. By having a pluggable architecture we allow ourselves the freedom of quickly prototyping agents and throwing them into the system and then returning

when development is further along to creating smart and specific agent behavior. This flexible dynamic architecture element is an attractive model for developers that are new to distributed agents as well because they can create simple agents easily and quickly progress to generating better agents as their skill set expands.

4.1.3 Maximum network transparency

As pointed out in section 1 a goal that we believe is necessary is to hide the details of the distributed system as much as possible but at the same time allow access to specific key components so not to restrict the developer to a single implementation. This goal is focused on balancing the system's flexibility versus it's transparency. We want the details of the distributed communication to be as hidden from the developers as possible. However we need to allow the developers the ability to specify certain elements if they are needed for their agents. We have determined that our primary goal for transparency is we should maximize the amount of transparency even if this limits some multiagent systems. We feel that this matches well with our goal for a simple API. Also, for our DVE, developers adding agents to the system should be focused on their agents behavior and not on networking issues and therefore a highly transparent system is better.

4.1.4 Maximum scalable

Scalability is a critical consideration in any distributed system and in a distributed agent system it is no different. We have to ask ourselves how many agents are we required to support simultaneously and also how quickly do agents enter and leave the system. For the developer of a MAS the issue can be addressed with two different approaches. the first approach is to leave the development framework open thus forcing the developer to make sure that her system will scale well ultimately giving more power and more of the development burden to the agent developer. The second approach is to build a framework that inherently supports an open and scalable system but ultimately restricts the agent developer from controlling certain aspects of how agents are distributed and how they communicate.

For a DVE we believe that in order to develop a simple and easy to use system that the second approach is the appropriate solution. Primarily we have the goal of allowing an unlimited number of agents to connect and therefore a peer-to-peer based system is far better suited to the task. Also, this is a similar trade-off to the transparency trade-off of allowing more control to the agent developer or more control to the framework itself and we have elected to ultimately restrict the agent developer in favor of having the multiagent system controlling the agent distribution.

4.1.5 High agent flexibility

With the goal of creating a DVE where agents can have virtually any behavior we must have the the goal of allowing as many types of agents as possible. Whether we are working specifically toward creating a DVE or if we were creating a generic Although a development framework that supported the creation of any agent is probably outside the realms of possibility we still set the goal to create such as system in hopes of supporting a maximum amount of agents. This goal also dictates that the agent developer should be given the maximum control over his agents and that our system should be as minimal in structure as possible while still allowing us to control the scalability and transparency.

4.1.6 Fast computational performance

The computation performance of our system is very important. Because our goals for creating a DVE require real-time performance computation time is critical and is also a goal for AI Loom. In most systems CPU performance is a trade-off between computational speed and memory consumption, accuracy, and complexity. In our case we feel that computation performance is more important than memory usage. And, we are willing to sacrifice complexity as well to gain performance. Having strong computation performance is important for many applications but we feel it is critical for AI Loom.

The decision to favor performance has a tremendous impact on our networking architecture. Network performance, although not directly related to CPU performance, has the same

disastrous effect on a real-time system of slowing the system down. Therefore, in our networking scheme we go to great lengths to reduce the communication overhead for the system and because we are also aiming for maximum transparency we are able to manage the amount of communication to some degree for the system. This increases the network and CPU performance.

4.2 Non Technical Goals

Beyond our technical goals for AI Loom there are two important considerations that have an impact on our design. These goals are to create an open and free software framework and we want to create a cross platform library. Although at face value these goals are not technical, they do impact our design on a technical level. In this section we discuss these two goals and how they impact our design.

4.2.1 Open and free software framework

With AI Loom we want to create a generic multiagent system software package, but more than that, we want to create a software framework that is useful in many applications and becomes a widely used package. We feel that the best way to gain a community of users is to make our software open to the public and freely available. Our hope is that AI Loom will be useful beyond our intended uses and that both academic and commercial applications can benefit from the software in theory as well as in practice.

In creating an open and freely available software package we have certain restrictions that non-open source software does not suffer from. Mainly we are restricted to using other software libraries that are also open and freely available and licensed under equally open licenses. For AI Loom we have decided to license the developed code under the LGPL[100] license and therefore all software that we include with AI Loom must also be at least as open as the LGPL. This restricts the set of software we are capable of using while building AI Loom. We feel that this restriction is greatly outweighed by the potential benefit to the general software community.

4.2.2 Cross-platform development

In addition to building a free and open software library, we also decided to set a goal that AI Loom be supported across operating systems. In an ideal world we would like to set the goal of building AI Loom for all platforms, however to be realistic we are only targeting Microsoft Windows and a handful of Linux variants. We have decided that as a cross-platform library , AI Loom will be potentially useful to more projects and open to the possibility of an embedded Linux system where AI Loom could be used in robotics or in other hardware platforms. Also by creating a cross-platform library we will have the added testing that inherently comes with working on software on multiple platforms. This helps create more stable and usable software.

Of course, the more platforms supported potentially means more complex code and it definitely means a more complex build system capable of working on a multitude of different configurations. Beyond our own software, we also have the added complication that all of the dependencies that AI Loom relies on must also be cross platform. This drastically reduces the set of software that we can leverage in the development of AI Loom and has impacted the design and implementation of our software significantly. Again, we feel the added gains of a cross-platform library far out weigh the additional development software complexities. With the ability for more developers to leverage AI Loom we will be able to see over time, if creating a generic MAS is useful in a range of fields.

Although we point out the implications of these goals in this thesis, a close inspection of the code would indicate just how impactful these goals are. It is surprising how many aspect's of AI Loom's implementation, beyond the high level points we describe here, are deeply impacted, if not entirely dictated by our choice of a dependent library chosen because it meets our open and free goal and our cross-platform goal. After we concluded our development, it was clear that having these two goals was important and that they impacted our design significantly. In the next chapter we look at the details of AI Loom's implementation and we point out in several instances how AI Loom's design and implementation were heavily affected by both a cross platform implementation as well as by keeping AI Loom open and free.

Chapter 5. AI Loom Architecture

The previous chapters have dealt with understanding the obstacles and design considerations that have gone into the development of a generic multiagent system, the goals that we feel are important to the system, and the major trade-offs in desiging the system. Having developed this foundation of knowledge and direction we proceed in this chapter with the architecture of AI Loom. In the last section, we developed six primary goals for the development of the AI Loom framework:

- simple API
- pluggable components
- high network transparency
- scalability
- flexibility
- performance

These goals as well as the foundation laid out in the previous chapters allow us to decompose the architecture of AI Loom into two convenient components. First we will look at the architecture we choose for individual agents and discuss how it meets our goals. After looking at agents as stand alone entities, we will continue our discussion of the architecture by looking at how AI Loom handles communication between agents and the distributed aspects of the framework.

5.1 High Level Design

At the highest level AI Loom is a C++ library. We choose to make AI Loom a C++ library for several reasons. First, as a language C++ gives us the CPU and memory performance we require. As we stated in the goals section, performance is a critical concern for our framework and by using C++ we are utilizing a language which will allow us to most effectively use the computing resources available. Also, C++ is an extremely common language which is very mature and it's merits and shortcomings are well understood which makes it attractive in that it has a very large user base. It is also the language of choice by game developers which indicates that it also can handle the vast array of usages which AI Loom may need to support. In addition to our language decision, we decided to make our framework a library as opposed to a simple code framework or prebuilt application to make it easier for developers to integrate into existing projects and to help us package and distribute the system more effectively, thus reaching more potential uses.

Beyond being a C++ library, the AI Loom architecture can be broken down into two logical modules. The first is a generic agent abstraction and the second is a communications layer. When we defined what constitutes an agent in chapter 2 we showed in figure 2.1 that the most basic agent must consist of sensors which sense an agent's environment and effectors which allow the agent to change it's state in the environment. In this chapter we expand on this model and present our own model for an agent which we have used to define an agent in AI Loom. Then we discuss how we enabled our agents to communicate through a clever classification of agent types. Because we have disassociated the agent architecture with the network layer this classification has no impact on the agent itself, but it is central concept in the network architecture. We discuss the agent classification scheme that we have developed at the begining of the network architecture section below, but first we discuss the agent architecture because this will help give us terminology used in the network section.

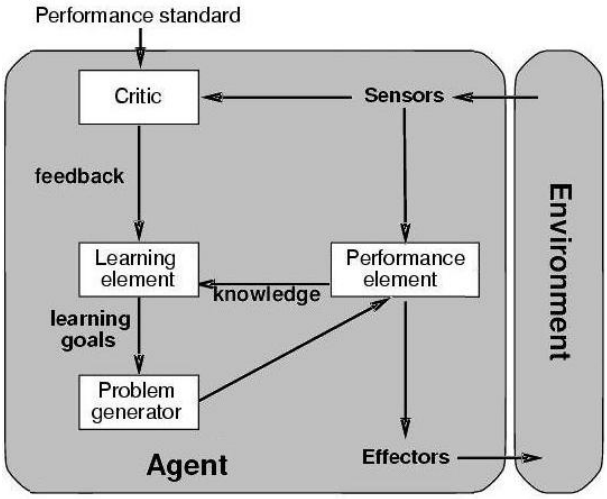

Figure 5.1 A generalized abstraction of how many agents can be modeled

5.2 Agent Architecture

The agent abstraction in any multiagent system is the heart of the system and AI Loom is no different. Because an agent can take on any number of designs it is important to have design goals for the agent and luckily we set our goals before beginning our design. After setting our goals we began our design process and our first task was to determine the representation of an agent. Because we have choose C++ as our language and because it is an object oriented language[101] it fit naturally to use a single abstract class object for an AI Loom agent which could be inherited from to create specific agents. The implementation details of our agent's class were guided by the goals we set out in our last chapter where we laid out six specific goals for AI Loom with four of those goals applying directly to our agent's design: simple API, pluggable components, flexibility, and performance.

5.2.1 Conceptual agent design

The design we present here meets these four goals and was originally constructed as an approximation of the agent model presented by Russell and Norvig[18] and repeated in figure 5.1.

A close inspection of figure 5.1 shows that by this design an agent is broken down into

six components. The effector and sensor components are fairly straight forward, they are the input and output the agent has with it's environment and an agent can have any number of each of these. In addition to the input/output for the agent there are four more components:

- The performance element is the final decision maker for the agent and chooses which effector the agent will use.
- The learning element can be used to track sensors and effectors and then use this knowledge to learn to make better decisions for the agent.
- The problem generator component is used to inject new or different behavior that the agent has not performed before.
- The critic component is a performance monitoring component which simply tracks the sensors input to dictate or record progress.

If we treat figure 5.1 as a directed graph, it is apparent that the shortest path an agent can utilize for an input/output cycle is: Sensor -¿ performance element -¿ effector. This shortest path dictates that the simplest agent must contain at least these three components and that the other components can be supplied for additional functionality but are optional. This design satisfies all of our stated goals. By requiring only 3 components for a basic agent we have a design that provides a simple and minimal API. By breaking our agent down into 6 total components, we have also formed the foundation for a pluggable architecture where different pieces can be pushed into place independent of the others. Also, by breaking the agent down into 3 required components and 3 optional components we make it easy for the developer to plug in any learning based system or algorithm such as a neural network or decision tree. This shows that our design is also very flexible in that nearly any type of agent could be constructed.

The Russel and Norvig model is conceptually very well organized with each component as a standalone piece of an agent. However, in order to create a usable implementation in a larger system we must deviate from it slightly. Rather than having one large agent object containing all of it's components we choose to make each component of the agent it's own base abstract

class object which developers can inherit from and create their own implementations. This deviation makes the agent itself very light weight and allows developers to only implement the parts of an agent that they need. This is nice from a flexibility standpoint because developers can add or remove components without having to change any other components or the agent itself. The decision to split the agent into independent classes for each component was also critical for our goal of a pluggable architecture; it allows us to load individual components into agents at run time using C++'s run time dll loading capability. It also allows us to much more easily create bindings to other languages to make scripting agents much easier. We feel that this design, although not conceptually as clean as might be possible does satisfy all of our goals we set for the agent architecture.

Although we have met all of our goals there are some drawbacks to our approach. Maintaining the execution flow from the graph in our agent model becomes much more burdensome and complicates the internal workings of an agent. Also, because each component is it's own class we have increased the API to some degree. To address the internal complexity we implemented the agent using a mediator pattern[102]. The mediator pattern gives the control of the agent to the performance element, making the performance element the heart of an agent and this conceptually makes sense because the performance element is the component that makes the decisions for the agent. Using a mediator was also essential for the overall framework, which will be explain below. To Address the increased API we feel that although the API is larger it is more straightforward. Rather than presenting one large class API for an agent, we have broken out the API into specific classes which developers may or may not be using. With our design completed we feel that although we may have increased the API and made the internal workings of an agent more complex, the agent developer is not in any significant way impaired by these decisions and therefore the users of AI Loom will benefit.

5.2.2 Concrete agent implementation

Our implementation of a standalone agent is diagrammed in figure 5.3.At the center of figure5.3 is the agent object, which is the mediator for the design from the mediator pattern.

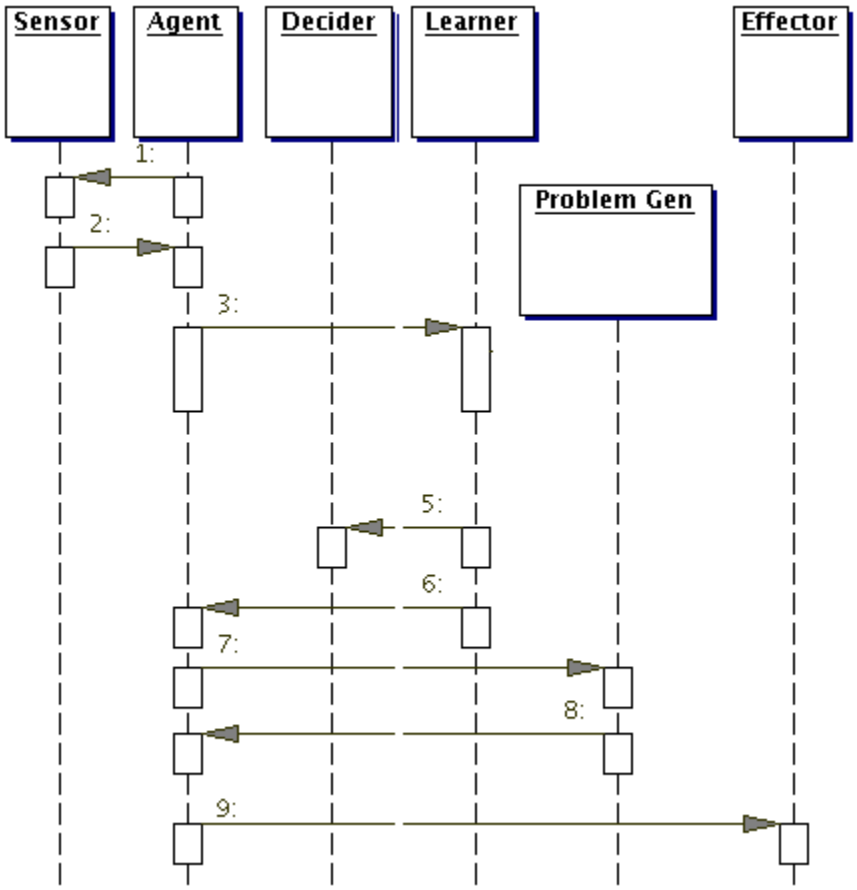

Figure 5.2 execution cycle for an agent, shown with all the standalone components.

Every cycle the Agent roughly approximates the flow of the directed graph in figure 5.1. It first checks it's sensors and passes the new sensor values to the learner, which is the learning element out of figure 5.1. The Decider object (which is what we call the performance element) is then given control to select an effector based on the sensors and input from the learner object. The problem generator is then queried to see if the agent should do anything irregular, trumping the decider's choosen effector. Finally the selected effector is executed. This model works well because the agent object maintains control of what the agent is doing at all times. The time-line in figure 5.2 shows us the control flow we just described for an agent.

Our model allows us to expose a very small portion of the agent to the end user through the mediator: the Agent object. The agent object then communicates with all it's components but

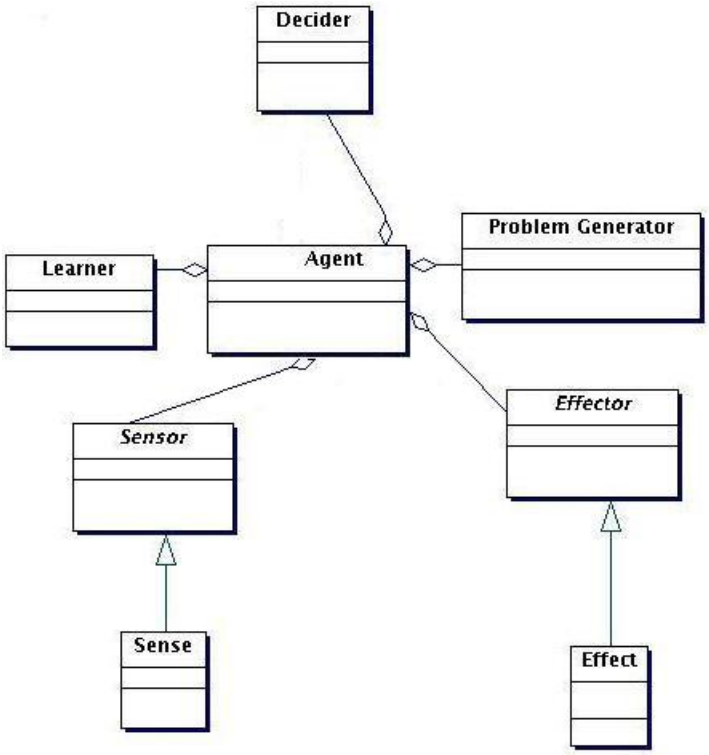

Figure 5.3 This is the basic agent architecture.

all the communication starts with the agent object. This design gives us the modular design we wanted, exposes a core API that only directly affects the agent, and a general model of an agent which allows for all types of agents. It satisfies all of our goals for an agent. Next, we discuss the rest of the AI Loom framework which builds on the agent model we just presented and adds support for multiple agents and communication between agents.

5.3 Network Architecture

The AI Loom network architecture is a pure peer-to-peer network. By using a pure network we do not complicate the system with multiple types of nodes keeping the API system and by using a P2P network we achieve our goals for scalability. In the last section we talked about how the agents architecture met four of our six primary goals for AI Loom. The remaining two goals, high network transparency and scalability apply to our network architecture. As we already indicated, by using a P2P system we have the means to supply a massively scaling network. We have also created a framework which hides nearly all of the networking details from the development of agents. In the rest of this section we present the network architecture in AI Loom and it's design implications. Our discussion starts with a classification mechanism deployed by AI Loom for how agent communication is structured in AI Loom. We then look at how the agent model is adapted to support the communication across the network and then we introduce the Loom Kernel component which is the central object in the network architecture for AI Loom.

5.3.1 Agent classification and communication overview

Our agent architecture, although composed of several components with various puposes, still maintains that the agents only have one means for gathering input from the enironment which we call sensors; and only one way of acting on it's environment which we call effectors. Therefore, no matter how complex an agent is internally, there is still only one input and one output mechanism. Our network architecture leverages this design detail by interjecting the network layer between the agent and it's sensors and effectors and only sending sensor input

and effector output over the network via messages. This has a number of benifits. As we'll show shortly this design gains us nearly complete network transparency for the agent developer. It also reduces our network bandwidth usage significantly which helps us with performance in the multiagent system. Finally it simplifies our overall design reducing the API. However, this approach is not free.

By proceeding with this design we have gained a more simple API, better performance, and greater network transparency but we have done it at the cost of a much less general framework because we have significantly limited how agents communicate. The limitation comes from only sending sensor input and effector output over the network and at first this may seem like a very restrictive communication mechansim, but a closer inpection reveals that it is actually quite powerful. If agents can only send sensors and effectors over the network then agents must determine if they will send and, or recieve other agent's sensors and effectors. This determination leads us to a classification mechansim for agents in AI Loom. But how do we classify agents to handle this communication?

To classify agents based on how they handle sensor input and effector output we must look at how this information can be useful. For the purposes of this discussion let's assume we have two agents in a network, Agent A and B. Let's also focus our discussion on communication from agent A to agent B (in a two agent AI Loom network the communication is actually bi-directional between agents, but we simplify this case for our discussion here and note that the upcoming discussion expands to n-directional communication). In this two agent system let agent A have a set of sensors: $S_A = \{s_{a1}, s_{a2}, s_{a3}, ...\}$and a set of effectors $E_A = \{e_{a1}, e_{a2}, e_{a3}, ...\}$. Likewise we have stes of sensors and effectors for agent B defined as $S_B = \{s_{b1}, s_{b2}, s_{b3}, ...\}$and $E_B = \{e_{b1}, e_{b2}, 3_{b3}, ...\}$repectively. In order for an agent to make a decision on which effector to choose from it's set it evalutates it's set of sensors. Because the set of effectors is disjoint for every agent there is an implicit mapping of $S \Rightarrow E$ for every agent. Therefore we conclude that both agents A and B must interact with their environments through the supersets:

$$A_{SE} = \{s_{a1}, e_{a1}, s_{a2}, e_{a2}, ...\} \quad and \quad B_{SE} = \{s_{b1}, e_{b1}, s_{b2}, e_{b2}, ...\}$$

agent A sends to agent B	agent B uses
set $A_{SE}$and effect e_{a1}	ignores it's own sensors and uses the effect from A: e_{a1}
set A_{SE} and effect e_{a1}	ignores it's own sensors and uses the effect from A: e_{b1}
set B_{SE} and effect e_{b1}	using it's own sensor and effector set B_{SE} and uses effect from B: e_{b2}

Table 5.1 Agent communication breakdown

What this indicates is that in order for an agent to select an effect the set of effects and the set of sensors used must have come from the same agent. Furthemore, in order for two agents A and B to communicate then they must share a set of sensors and effectors that in the union of these two sets. We define this set P_{AB} of possible sensor and effectors as:

$$P_{AB} \in A_{SE} \bigcup B_{SE}$$

Returning to our case with all communication flowing from agent A to agent B, let's assume that agent A has available to it both of the supersets $A_{SE}$and B_{SE}. When agent A makes it's decision it sends the decision to agent B by telling B which effect it chose. We are left with only three possible scenarios for how agnet B uses this communication presented in table 5.1. This table shows that agent A can choose to use either it's own sensor and effector, set A_{SE}. Or it can choose to use the set B_{SE}. It then sends which set it used to agent B along with the effect it choose. Agent B recieves the sensor and effector information from agent A and then it chooses to either use it's own sensors and select it's own effector from set B_{SE} or to use the effector that it recieved from agent A.

We conclude from this discussion and from table 5.1 that there are only three possible communication options between two agents and therefore we have $3\,options \times 2\,agents = 6\,agent\,classes$. We actually define one additional class for an agent that does not communicate for a total of 7 agents classes. In AI Loom this agent classification is criticial to how agents use communication and we declare and define each class in the context of AI Loom in table 5.2. Conceptually and in implementation these are constructed in pairs with requester agents working with solver agents, parent agents controlling drone agents, and listener agents observing subject agents.

Agent class	description
Requester	An agent of this type is searching for an appropriate action to take. It broadcasts it's sensor's input values and it's possible actions (or effector as it is called in AI Loom) it can take over the network and waits for a reply which consists of what effector it should use.
Solver	An agent of this type is the counterpart to the requester agent. It listens on the network for broadcasts from requesters. When it receives a request it will attempt to analyze the sensor input and suggest an effector. If it can determine an effect then it replies to the requester the effect to choose.
Drone	Drone agents are told which effector to use at all times and they do not listen to their own sensors, it does this by constantly listening for broadcasts over the network for what effect to use. A drone is often used to mimic another agent somewhere else on the network however it does not have to.
Parent	The parent agent is the counterpart to the drone, it uses it's own senses and determines it's own effect and then broadcasts over the network what it's effect is and drones listening to this parent use the same effect as the parent.
Subject	An agent that is a subject also uses it's own sensors and effectors, however any time it uses an effect it broadcasts this information over the network for any listener agent that may use what this agent is doing to help themselves.
Listener	Listener Agents use their own sensors and effectors but also watch subject agents for what they are doing. By watching the actions of other agents directly agents can work together very efficiently. A network in which all agents are both subjects and listeners is a common usage because agents can act completely autonomous but group behavior can be modeled as agents observe eachother's actions.
Standalone	This is the default class for all agents. In this class the agent reads in it's own sensors and decides which effector it should use without any incomming or outgoing communication.

Table 5.2 Agent types in AI Loom

The clever reader at this point will have noticed that the only information that is required to be sent over the network is which effectors and which sensors are being used. This minimizes network communication and overhead and makes agent communication in the traditional sense of passing detailed messages unneeded. All an agent needs to communicate is what it is sensing and what it is effecting. Although there are a few additional messages passed throughout the network which we'll discuss later, we have developed a system which requires very little communication overhead. This was very important for our goal of scalability, in that we want to be able to support any number of connected agents and minimizing communication in a densely populated network is crucial when real-time performance is required.

In addition to the classification of agents, each agent goes through an initial state when first connecting to an AI Loom network. We discuss this initial state and what happens before the agent is fully connected to the network in the networking section below, however it is important to the high level design of the system to know that agents must first negotiate with the network what classification they are in before communicating with the network. This initial communication happens any time a new agent joins the network or when an agent needs to change it's classification. When that happens messages are broadcast to alert the network to the change. We talk in more detail about how this negotiation happens in section 3 of this chapter, but first we need to look at the architecture of a single agent and how it fits into the AI Loom framework as a whole.

5.3.2 Agent-Decider proxy

In order for agents to communicate between one another we needed some way of telling an agent to do two things that normally it would handle itself. We pointed out earlier in this chapter that the only required communication between agents in AI Loom is which effectors and sensors an agent should use. By restricting communication between agents to this paradigm we have significantly reduced the complexity of our network model. All our network model has to do is to tell each agent to either listen to its own sensors or use some other copy of them that will be supplied, and also to to tell each agent to use it's own chosen effect or to use a

different effect supplied from the network. We must update our agent model to support this functionality, but to update our agent model with the minimal amount of impact is a hard problem to overcome because the flow of execution through an agent is controlled directly by the mediator pattern. The solution we choose was to use a proxy object[102] between the agent and the decider.

As explained before an agent can be in any combination of the seven agent types. When an agent type is set the proxy determines where the sensors will be coming from and where the effector will be going to. When the agent is in Standalone mode the proxy passes the agent's own sensors in to the decider and when the decider returns, the proxy uses the decided on effector. However in a distributed mode the sensors may not be an agent's own and the chosen effect may need to be sent and applied to a different agent. As an example, if we have two agents A and B. Agent A may ask agent B to calculate an effect for him (Agent A is an AI Loom requester agent and agent B is an AI Loom solver agent). In this scenario agent A sends agent B a set of sensors. This is handled transparently by the proxy object sending agent A's sensor values to agent B instead of to agent A's own decider. When agent B's decider determines the effect to use agent B's proxy doesn't use the result directly, instead it passes the effect back to Agent A to use. This is a simple example to demonstrate the power of this communication model and how the proxy object is the vehicle for controlling communication at the agent level.

Using a proxy object is a great solution to help us minimize impact on the existing agent design as well as hide network details. The proxy solution allows an agent to handle the distributed communication that AI Loom requires without changing it's own structure or any components. In addition to the agent architecture being essentially unchanged the agent developer does not need to bother with any network or communication details. He simply writes his agent to use certain sensors and select from a set of effectors and he doesn't care where the effects or senses are coming from which provides a great deal of network transparency to the agent developer and mostly satisfies our transparency goal at the agent level.

The proxy object is the answer to how an agent's communication is chosen, but what how

do the sensors and effects actually get sent over the network? How do agent's join a network and share their sensors, effectors and abilities? And, how does the communication actually get routed over the network? In the next section we address all of these questions and more by presenting the heart of the AI Loom framework, the AI Look kernel.

5.3.3 AI Loom kernel

In the last section we talked about how by adding a proxy object to the agent we have added the ability to direct the sensor input and effector output to different agents. In this section we cover the AI Loom Kernel which is the module in AI Loom that manages communication between agents. When we decided to build our multiagent system as a peer-to-peer system we realized we had two options for what nodes in our network graph would contain. We could choose to make each agent it's own node or we could create a single network node which multiple agents belong to. We choose to allow for multiple agents at each node within a loom network because this is more flexible to the agent developer since he can still create one Kernel per he creates agent or to have multiple agents per Kernel. The implication of this decision is that we are required to create a platform at each node which handled the network communication for all the agents that reside on that node. We call this platform the AI Loom Kernel. In addition to managing communication for each agent, it also handles registration and initialization of agents as well as load balancing and supplies the main execution loop which agents run in.

5.3.3.1 Kernel initialization

During the initialization of an agent the Loom Kernel gives the agent a universally unique identifier (UUID)[103] which is used internally to AI Loom to identify each agent and because of the nature of UUIDs it is unique across the network as well. Before the kernel begins execution of the agent it must prepare the network for the agent depending on it's classification. If an agent is any of the non-standalone classes then it broadcasts all of it's known sensors and effectors to every other agent in the network. We'll discuss exactly how that is possible in the next section, but for now we glance over the how and say that after this initial broadcast, every

agent in the Loom network has knowledge of and potentially usefulness for the sensors and effectors that the newly joined agent has. After this initial network initialization the kernel spawns a thread for the agent and starts the agent executing in that thread. The agent then begins its execution cycle as shown in figure 5.2 until it is again interrupted by the kernel or until it is unregistered from the Loom kernel.

5.3.3.2 Kernel communication

The AI Loom kernel is also responsible for all communication between agents. When an agent's proxy object diverts it's sensor's values or it's effector result it sends that information to the kernel. The Kernel is then responsible for packaging the information from that agent into a packet that can be sent across the network. It is also responsible for unpacking received packets, determining if the message is for any of the agents connected to it and either delivering the message to the appropriate agent or passing the message along to any connected peers in the network.

Our system is now nearly complete and we present the complete model for the system in figure 5.4.

The astute reader will immediately notice that there are three components in this diagram that we have not yet discussed. The AI Loom kernel has a post-map object, a pre-map object and a network connection object. These three components are part of the AI Loom kernel and help with the handling of distributed communications. The handling of distributed communication is presented in the next subsection and is the last piece of AI Loom's architecture.

5.3.4 Network communication implementation details

For the communications layer we use a library called Plexus. Plexus is a library that manages communication across a peer-to-peer network using message passing. It supplies us with the abstract concept of routers; each node in the network must have a router which handles the routing of messages throughout the network. The router can be different at each node, but in AI Loom it is the same. Plexus uses the unreliable communication protocol

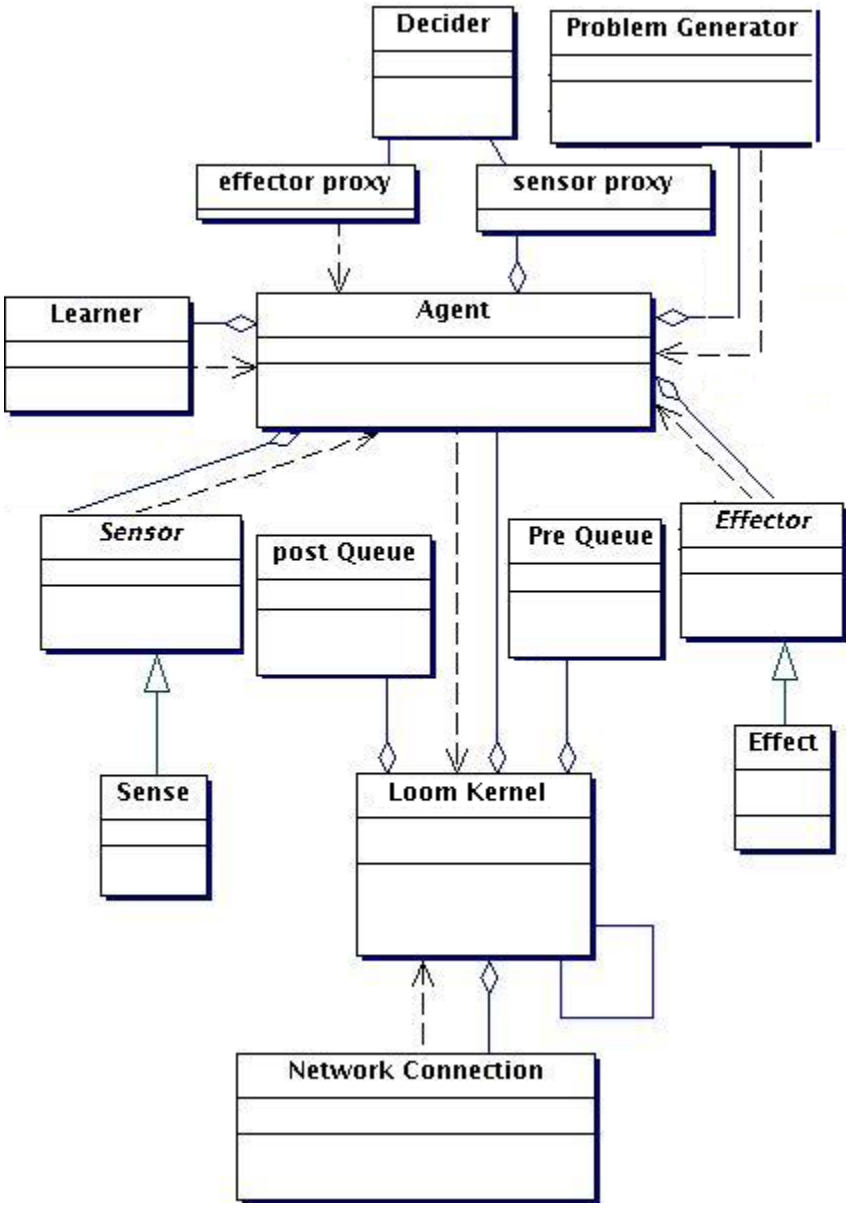

Figure 5.4 An agent and its relationship with other Loom components

UDP[104] because we do not require connections and because UDP is generally faster than TCP and handles all the routing of messages throughout the network for us, by usng Plexus we did not have to create our own socket and message passing functionality; we only had to write a message router for the AI Loom nodes and hook into Plexus to send our messages. To use the plexus networking functionality with as little exposed to the end agent developer we wrap the creation of the socket communication, connection, sending of messages, and receiving of messages into our own object which we call a Network Connection and can be seen at the bottom of figure 5.4. By wrapping all of this functionality into a single interface we have provided a convenient facade design pattern[102] to the AI Loom architecture for the network aspect of the design. The purpose for using Plexus is that it provides all the functionality we need for message communication on a peer-to-peer network and it's interface allowed us to create this singular facade.

The network connection facade supplied to the AI Loom kernel greatly increases the simplicity of the system in the internal design of AI Loom. Primarily the network connection object must encode and decode every message it receives from local agents and remote agents respectively. For every message that the network connection object receives, either from local or remote agents, it must negotiate with it's local Loom kernel and determine if the message should be delivered to any connected agents . In figure 5.5 we see that the relationship between the network connection object and the pre and post maps that when a message comes in to the Loom Kernel from the NetworkConnection it determines if the message is for any agent that it has. If it does have an agent that should receive the message then it puts it puts the message in one of the maps based on the type of message that is received. If the message is to perform an effector then it goes in the post-map if it is to use a specific sensor then the message goes in the pre-map. We use a map to store these messages for the agents because each agent runs in it's own thread and may take very long periods of time computing between execution loops. The maps allow us to immediately process the message and as soon as the agent is ready it has access to the map. We also made the decision to use two maps, one for sensors and one for effectors. This was not strictly necessary, however we felt that adding a

pre-decision map for sensors and a post-decision map for effectors helped clarify the system and made debugging easier.

After the messages have been pushed into the appropriate maps the Loom Kernel continues is finished with the message and it is left to the agent's sensor proxy object to check the maps and pull off any messages waiting for it. After the decider makes it's decision, it sends it's effector through the effector proxy which then performs the correct operation based on what classification the agent is. If the agent is of the appropriate class then the effector proxy generates a message to send over the network and sends it to the Loom Kernel who gives it to the NetworkConnection and then Plexus broadcasts the message out for us.

The model that we just presented is a bit complex but it maximizes run-time performance for the agents and allows us to hide the complexity from the user who simply creates his agent without care for the detailed network process happening internal to AI Loom.

5.4 Additional Features

Throughout this chapter we have explained the base design of AI Loom and provided supporting arguments for our design decisions. We started with a conceptual model for a generic agent and constructed a functional design for our agent and then added to the agent necessary pieces to facilitate communication in a multiagent system and finally in the previous section we covered the Loom Kernel and how communication is managed and how agents are initialized at a high level. In this final section of this chapter we take a look at the some additional features that round out the design of AI Loom and help us further meet our design goals. There are two specific features that we'll discuss here. First, we discuss how agents share new sensors and effectors with each other and second we talk about the routing of messages over the network.

5.4.1 Sensor and effector sharing

As new agents connect to an AI Loom network they may bring new sensors and effectors to the network that other agents do not have access to. Because of this, when an agent is

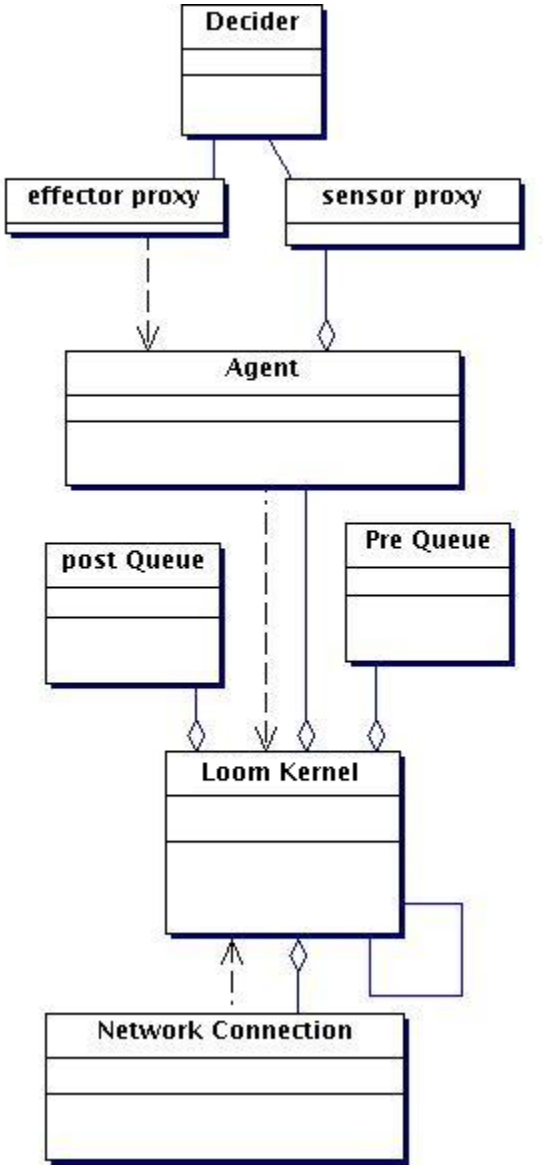

Figure 5.5 All the components involved in propagating a message in Loom.

added it goes through an initialization process with the Loom Kernel, at that point all of it's sensors and effectors are registered with the Kernel and a broadcast message is sent to the network connection object saying a new agent is joining the network. That message contains all of the new agent's sensors and effectors along with a hash key for them. The hash key is used by every other agent connected to the network to determine if it already has the sensor and effector. If it does not have any of the passed objects then it retains a copy of the sensor or effector. Conceptually this is a simple solution but in practice it is quite challenging. How do we send sensors and effectors over the network and how do agents load and use them once they are already running?

We developed two answers to this question. Our first solution was to develop sensors and effectors as shared object libraries. The library could be built and using C++'s run-time dynamic library loading the library itself could be sent over the network and loaded by agents at every node. This worked to a limited degree, it does complete the pluggable architecture that we wanted, however it is not portable; a library built with one compiler will not work with another on a different machine, thus breaking cross-platform functionality. Also, if error handling in the compiled library proved to be very challenging and caused the whole network to crash. Because of poor stability and the inability to use sensors and effectors cross-platform we developed a second mechanism to allow us to share sensors and effectors across a network.

Our second solution was to provide python bindings for the sensor and effector classes. This turned out to be a huge success on multiple levels. First, once an agent's sensors and effectors could be created in python and reloaded at run-time we had a much more pluggable architecture; and it allows for very rapid iterative development, something that we identified as important from the game industry. Also, it solved the sharing problem by allowing us to easy package up the python script file and sending it over the network to be used by every other agent. After seeing how successful this was at providing a truly pluggable component we dropped the dll sharing from the design entirely and now only have the python binding solution.

5.4.2 Message routing

AI Loom uses the Plexus library to manage the network communication. In Plexus messages are passed throughout the network as broadcast messages and at each node in the network sits a router. The router controls how messages are delivered to all the nodes peers. In the simplest implementation all messages are sent to all nodes. This is the default implementation that AI Loom uses. AI Loom does expose the router object to developers in the event that a specific networked MAS needs to override this default functionality. This is in fact a likely situation, for performance reasons since every effector and sensor result sent over the network would quickly use all available bandwidth in a densely populated network. In the next chapter when talking about our metaverse client implementation we take advantage of this feature in order to only send messages to/from agents in the same virtual vicinity to each other.

Chapter 6. AI Loom evaluation

In this chapter we evaluate the succssfulness of AI Loom by evaluating how well the system met the goals we set in chapter 4. We also evaluate the system's ability to create a wide range of applications by observing AI Loom in two different use cases and finally in the next chapter we present AI Loom's usefulness specifically in creating a distributed virtual environment.

6.1 Evaluation of Engineering Goals

In the previous chapter, we presented the design of AI Loom as it is currently implemented; during our discussion of the design we frequently referenced the goals for AI Loom as premise for design decisions we made. Therefore, it should come as no surprise that we feel that we have largely met our goals for a multiagent system. The rest of this section is broken up into the six goals we set for AI Loom. In each part we restate the goal and then explain concisely how AI Loom's design meets those goals and in places where we may have fallen short of goals.

6.1.1 Minimal and simple API

The completed design for AI Loom requires just a few lines of code to create a simple agent and connect the agent to an existing AI Loom network. We encourage the reader to reference the appendix to see an example program using AI Loom. In all, the API consists just 5 public classes which are either used or inherited from and has a total of just 33 exposed functions for the entire system, most of which are not needed in most user applications. Also, because just a few of the core AI Loom components are exposed through python bindings some usages of the system require only the most basic set up and all remaining work can be done through python scripts. For agent developers this can be extremely valuable because they only are required

to implement their functionality without learning any of the details of the system. Whether scripting agent behavior or setting up a C++ system from scratch AI Loom provides a very minimal API and short learning curve.

The counter-argument to a small API is that the system must be doing work for the user behind the interface and therefore may be making too many assumptions about the uses. We feel that in the case of AI Loom we have reached an excellent balance in this trade-space where enough functionality is exposed to give users the control they need while hiding the gory details. Even if this is not the case for some uses of AI Loom, we have decided to make the project open and freely available; therefore users are free to learn the internal workings and change or expose functionality that they need. However, we do not anticipate this happening frequently or even at all and the API we have presented is our best attempt at balancing a simple API with the exposing of maximum functionality.

6.1.2 Pluggable components

In previous chapters we went to great lengths to explain how AI Loom's design is a very component oriented design and agents are able to swap every component they use at run time. This is, by definition, a pluggable architecture and with the addition of allowing components to be written in python and swapped at run time without the need to recompile we feel we have really maximized the dynamic ability of AI Loom.

6.1.3 Maximum network transparency

The second half of the AI Loom architecture chapter was devoted to how networking works internally to AI Loom. We pointed out several times that none of the networking functionality is exposed to the end user of the system, that all that the system knows how to do is override sensor and effector input and output. Our goal was to maximize transparency, and we feel we achieved that, however this ends up somewhat restrictive to the agent developer. There is no way for instance for agents to directly pass knowledge between themselves. This, can be done of course by representing knowledge in a sensor and broadcasting the sensor's state over the

Loom network, however this is somewhat obscure. What we found after developing the system and presenting it to several developers for review is that we may have actually went too far. Most developers would prefer a more direct mechanism for passing information around the system, rather than pseudo-encoding it in a sensor which seems obscure. In the future work section in our conclusions we have noted that it may be worth while to revisit this goal and relax it somewhat.

6.1.4 Scalability

We developed AI Loom as a peer-to-peer system specifically to deal with the goal of stability and we have stress tested AI Loom with over 5000 simultaneously connected agents running on multiple machines and the system supported this with ease. Does AI Loom scale into the billions of connections? Well, we did not have the resources or time to answer that question, but we believe that AI Loom should scale to any size. With that many connections the Plexus library may require very specialized routers in order to intelligently manage network traffic. However at that point the problem domain changes, it is no longer specific to AI Loom's capability to sustain that many agents and rather becomes a network routing problem.

We also tested AI Loom's ability to manage multiple agents on a single machine and found that the overhead in AI Loom was very low and that well over 400 agents can co-exist on a single AI Loom kernel. We did not test more than 400 agents because of threading limitations on some platforms. In the future notes section we point out that a different threading model would allow us to more easily support a far greater number of agents on a single machine. At this point in the maturity of AI Loom we are happy with it's scalability both on a single machine and as a distributed platform for agents.

6.1.5 Flexibility

The flexibility of AI Loom is a standout feature of the system. In the course of our research and usages of AI Loom in other projects we have seen multiple different learning agents developed. We have seen AI Loom used in a non-distributed setting very effectively. In

the next section we show how AI Loom was used in a commercial quality video game and in the next chapter we show how AI Loom was used to create a distribute virtual environment. In total, AI Loom appears to be very capable of creating many different types of agents.

6.1.6 Performance

We would grade the memory and computational performance of AI Loom as moderately successful. Of our six design goals, performance is the only goal that we feel leaves significant room for improvement. A performance analysis of AI Loom indicates that the framework does have a fairly high overhead cost in computation. We also consume many CPU cycles with unnecessary network traffic handling. Also, when the python binding are used we see significant performance drops. These are still open areas of research and as AI Loom matures we expect that we will see enhancements to the performance of the system.

6.2 Application Development

In the previous section we reviewed the design goals for AI Loom and concluded that the system does meet the design goals very well. The proof of any software system though is in it's actual usage and AI Loom does not disappoint as a generic multiagent system. So far, AI Loom has been used in several non distributed applications, as a testbed for several AI learning projects, in a commercial game and most importantly in our own distributed virtual environment application test case. In this section we review AI Loom's usefulness in one existing application and point out a number of research projects that could benefit from using AI Loom for their development platform.

6.2.1 Current uses

AI Loom has been used in several research projects and at least one commercial venture. At the Virtual Reality Applications Center (VRAC)[105] two projects took advantage of AI Loom, one application to develop a virtual reality football simulation to help train athletes. In this project agents were created for each simulated football player. To our knowledge the agents

were not distributed, however the simulation was created so that this would be a possibility if performance limitations required it. The project is still in it's infancy, but the benefits of AI Loom's architecture are already evident. Very simple decision logic was first implemented for the agents in the simulation and as development continues more advanced intelligence will be developed and iterated on and swapped in and out of the simulation for testing purposes. The biggest advantage of AI Loom for this project is that as decision logic is tuned and updated the old decision logic can be swapped in and out at run-time allowing the designers to evaluate the performance of the AI in the simulation very quickly.

Also developed at the VRAC is a Hindu temple and marketplace simulation that utilizes AI Loom for all of the character AI. In this project characters were programmed with behaviors to help user's of the simulation to learn local customs and behavior. AI Loom was used in this project as a non-distributed multiagent system where each characters was an agent and decision logic was created to react to user interaction and cause behaviors that simulated real world reactions. AI Loom's simple interface was very useful for this project as this was the second iteration on the project and AI Loom was not used in the first iteration. Because of the simple API and because agent's in AI Loom are abstract objects, incorporating it into an existing application was fairly painless for the developers and the benefits of having rapid iteration were felt nearly immediately.

In addition to the projects at the VRAC, AI Loom was used in a commercial game development setting in a game called Treefort Wars[106]. This was a real-time strategy game that, to our knowledge was never published but did win awards at the independent games festival[107]. The game consisted of many characters which the user could give instructions to, such as build a structure or to attack an enemy, and after given instructions the character acted semi-autonomously to achieve the goal. AI Loom was used at for multiple purposes in Treefort Wars. The first usage was to create agents to simulate human AI opponents for the user. These agents were given decision logic to control characters in the game to accomplish tasks and to react to the user's actions. The second use was for each character in the game. Because the characters acted semi-autonomously based on goals, logic had to be created for

each goal that a character could be given. AI Loom proved very useful as a rapid prototyping tool in the development of the game and allowed the developers to rapidly iterate and tune the AI for the game.

In addition to these existing applications we have a number of examples and test applications that we developed as part of the AI Loom framework. Of course, because the over-arching goal of our research is to evaluate the feasibility of AI Loom as a framework for a distributed virtual environment, in the next chapter we look at specifically at the prototype DVE that we build using AI Loom called Plane. With Plane, the virtual reality football simulator, the Hindu marketplace and Treefort wars we feel that AI Loom has already been applicable to a wide variety of applications, but there are many additional areas that could also take advantage of AI Loom's design.

6.2.2 Possible additional applications

Because AI Loom does not specifically implement any mechanisms for the distributed virtual environment problem and maintains that it is only a distributed multiagent architecture we have been pleasantly surprised by AI Looms potential applicability to other research areas. Although we have not investigated with rigor AI Loom's ability to work in a wide variety of applications, based on our review of current literature that AI Loom would work favorably for most existing research into multi agent systems, the main category that it does not work well for is in research that deals directly with communication. We also believe that because AI Loom is a cross-platform, open, freely available software that it may find usefulness in a broad range of commercial applications.

In conclusion we feel that AI Loom has great prospects as a generic multiagent system, it meets all of our design goals and has already found usefulness in several projects. In the next chapter we look at the specific usage we intended for AI Loom and discuss our design and development of Plane, a distributed virtual environment built on AI Loom.

Chapter 7. Plane, a distributed virtual environment built with AI Loom

The long term goal for our research, beyond what we present in this thesis, is to create a viable peer-to-peer based distributed virtual environment. We felt that a multiagent system was an excellent candidate for creating the foundation for us to build a DVE. Our research took us down the process of designing a generic distributed multiagent system which yielded AI Loom. After evaluating AI Loom as it was used in several other projects we felt confident that we could build a prototype DVE with AI Loom as the foundation. In this last chapter we present Plane, the DVE we built and what we learned about multiagent systems as they pertain to DVEs.

7.1 Goals and Scope of Plane

The goal for Plane was not to create a full featured distributed virtual environment capable of being used in a commercial or even in a research setting. Instead, Plane is a prototype for part of a distributed virtual environment client application. Specifically what we wanted to create with Plane was a client application that addressed the problems of sharing user created objects and logic across a distributed peer-to-peer network in a three dimensional graphical world. At this point in our research we have created a very simple client application capable of demonstrating some of the key features necessary to create a distributed virtual environment and we have laid the groundwork for a potential solution to creating a full feature DVE. Before we discuss in detail the design and implementation of Plane, we present the specific goals we set for this phase of our research. Again, we emphasis that at this point we are not attempting to create a feature complete DVE but rather a prototype to evaluate the usefulness of AI Loom as a platform for a DVE, and specifically our prototype attempts to meet the following goals:

- Users should be able to connect remotely to a three dimensional virtual environment that they can navigate
- Each user should be represented by an avatar in the virtual environment to himself and to all other users
- Users should be able to create their own behavior for their avatar through scripts
- Users should be able to load any object with or without scripted behavior into the world as an autonomous agent
- As long as a single computer is connected to a location in the virtual environment all objects in that locale will stay in existence
- Any number of users and objects should be able to be added to the environment

The above six requirements for Plane, have left us with a wide scope for our project, however we feel that these are the minimum engineering requirements to prove AI Loom as a possible candidate for creating a peer-to-peer based DVE. We say a possible candidate because we have intentionally left out of our goals the problems of network security, economic and social impacts and several other smaller problems which must be solved for a truly viable DVE. In our concluding chapter we discuss security and other areas of further research that would be required if we wanted to create a full feature DVE.

7.2 Design of Plane

Plane is a client application that allows users to connect to the Plane network which is a peer-to-peer network with a shared state representation of a virtual environment built using AI Loom. The application itself was developed on Linux and uses the graphics engine Ogre[108, 109] for rendering. When the client starts up the user must supply a domain name and port for another machine that is connected to the Plane network and an avatar which will represent the user in the virtual environment. In the rest of this section we talk about

how Plane was constructed, how it uses AI Loom and how well AI Loom was suited to the construction of Plane.

7.2.1 High level design of Plane

The design of Plane is very simple, thanks in no small part to AI Loom's design. The client application for Plane is an Ogre graphics application. It uses Ogre for window management, rendering the virtual space and for input. We will discuss the choice of Ogre in a later subsection but the primary reason for using Ogre is that all entities rendered in Ogre are represented by C++ objects, which is also how AI Loom represents agents, making for a convenient and straight forward mapping between objects in the virtual environment and agents. In fact, this is accomplished in Plane with a single base abstract C++ class that we call a metaObject. The metaObject inherits from an AI Loom agent and contains a reference to a single Ogre SceneNode. This makes the metaObject an AI Loom agent which immediately registers it with the Loom Kernel and makes it available on the Loom network while at the same time registering the SceneNode with the Ogre rendering system so it renders in the virtual environment.

Because of this convenient mapping the entire design of Plane boils down to a single Ogre application object which manages rendering and input and a set of MetaObjects which represent all the objects in the virtual environment. We show a UML diagram of this simple system in figure 7.1. There are a few additional ancillary classes which are not shown that maintain timing information and input state but at it's core the design is very simple and only consists of MetaObjects, the Ogre application and AI Loom components. From the UML diagram we see that the metaObject is a Loom agent and that each MetaObject has a Loom::decider and Ogre::SceneNode. The UML diagram shows the simple mapping and the central design of the MetaObject.

The Diagram shows that there is only one PlaneApp and only one Loom Kernel, and there can be one or more MetaObjects. What the diagram fails to show is that one MetaObject represents the user's avatar and that all other MetaObjects are either other objects or other users in the Plane network. The MetaObject that represents the user is classified as an AI

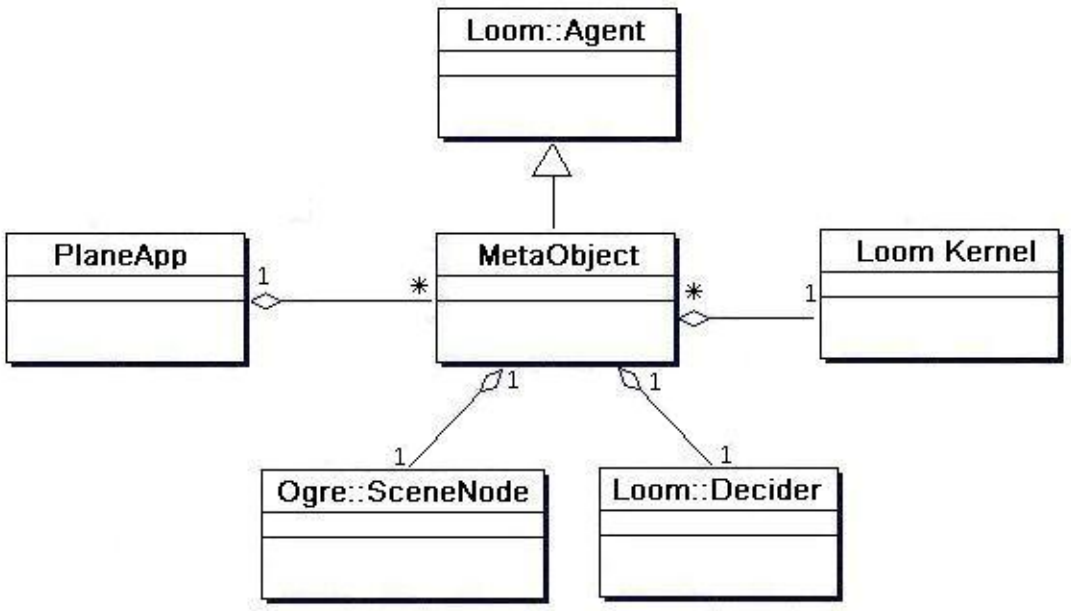

Figure 7.1 This is a near complete UML diagram of Plane

Loom Parent agent and all other MetaObjects are drones. Recalling from chapter 5 and our high level design of AI Loom, drone agents are mirrors of their parent node mimicking their every effect. This is the key concept that enables the Plane client to share the DVE's state with all connected clients. As each parent agent updates, it broadcasts it's AI Loom effector over the network, and at every other connected client a drone exists which duplicates the same effector, thus keeping every client in sync with every other client. In the next subsection of this chapter we talk about the initialization process for Plane and how a drone agent is created on every node when a new parent starts up.

7.2.2 Plane initialization

One of the primary reasons we built Plane on top of AI Loom was to utilize the initialization of AI Loom and it's agent initialization. When the Plane client starts up, it creates one MetaObject which is an AI Loom agent and then connects to the AI Loom network. When the initial connection to a peer happens AI Loom progresses through it's normal initialization process. As part of that process it determines what it needs to do at each peer with the new agent. In the case of Plane, we have configured the AI Loom kernel to duplicate any parent

nodes on all peers. Again, we designed AI Loom with this use case in mind and it has matched our expectations and been invaluable to creating a DVE.

After the newly connected agent has duplicated itself on it's peer and the new agent begins to propagate across the network, the newly connected client asks it's peer for a copy of all the agents it currently has. This is another feature of the AI Loom initialization process and was simply a configuration parameter supplied to AI Loom from Plane. Once a list of all agents is returned the local client duplicates every agent as a drone agent and our initialization process completes. At this point every node in the network has a copy of our new client agent, and the local client application has a copy of all the agents in the network and all of this was handled by AI Loom.

7.2.3 Adding behaviors to avatars

Plane is built with a small set of predefined behaviors for the client's avatar. These behaviors are all simple navigation commands: move forward, turn right, turn left, move backward, strafe right, strafe left. Each of these behaviors is actually a AI Loom effector and if a user choose to, she could write her own decider logic which took advantage of these and moved her avatar through the virtual environment, however she saw fit. Of course, if this were all that we could do with an avatar in Plane, it would be a very boring DVE. Luckily, AI Loom comes to the rescue again. Because of the pluggable component architecture and because effectors have exposed python bindings, anyone can write their own effector and add it to their client to make their avatar do whatever they want. This feature directly addresses our goal for Plane of allowing users to modify their avatar to behave in any manner.

Once a user's new effector has been added, AI Loom automatically populates it across the network and as soon as the client, parent agent chooses to use that effector it is used by all drone agents on every other client. Because AI Loom takes care of propagating changes to agents, users of Plane are free to focus on creating new content, effects and behaviors for their agents.

7.2.4 Adding non human agents to Plane

Another goal we set out for Plane was to allow users to add arbitrary content to the Plane virtual environment with or without associated AI behaviors. Once we had the ability for two Plane clients to connect and share the world space and see eachother's changes in real time, we modified our client to be able to start up without a graphics window and without input and just add an autonomous agent to the network. The propagation of the agent works exactly the same as for user controlled avatar agents, the only difference is that the agent has a decider component that chooses effectors for the agent. By default all agents have a "do nothing" effector, so if no decider is provided then the object that is associated with the agent will be loaded into the virtual environment and do nothing until that agent drops out of the network.

7.2.5 Rendering and input system: Ogre

We already mentioned that we choose Ogre for our rendering and input system because it mapped so nicely into AI Loom's design, but there are many reasons for choosing Ogre for our Plane. Ogre is well suited to rapid prototyping which we needed with the short time frame we had to implement Plane and with the research oriented work we were doing. In addition, Ogre is a mature software and heavily used graphics engine for video game development and therefore support for projects like ours is ample. Along these same lines, Ogre comes with tools to export three dimensional models from Autodesk's 3D Studio Max and Maya making it easy to drop in new content to Plane.

Besides the rendering system, we choose Ogre because of it's input system abstraction OIS. OIS gave us a complete input scheme ready for navigating a virtual environment. We used the input system along with some primitive rendering concepts to create a navigation mechanism which causes the client's avatar to perform terrain following while navigating the virtual environment. using this input system, along with Ogre as our base window and rendering system we were quickly able to create a client application that could render a test scene and load an avatar that could navigated around a virtual environment.

7.2.6 Persistence of state in Plane

Because every object and user avatar is duplicated on every machine, even when machines drop out of the network their objects stay visible in the virtual environment. If an agent rejoins the network it can pick up from where it left off or start over depending on how it is reinitialized. This gives us partial persistence. If all the nodes drop out and then one agent restarts he will no longer see any of the previously connected agents. So persistence is only maintained in so far as one client maintains connectivity.

7.2.7 Plane design evaluation

In the next chapter we discuss Plane's use of AI Loom as a framework for a distributed virtual environment and the many potential areas for improvement and additional research areas, but before we get into the details we would like to point out that Plane as it is implemented now, meets or exceeds our initial design goals we set at the beginning of this chapter. A user of Plane is able to connect to the peer-to-peer network and when she does, she is represented by an avatar that she can customize. Objects can be loaded beyond just the user's avatar and those objects can have any behavior that is scriptable. Objects in Plane and the user's avatar representation all have limited persistence in the world in so far as is possible without a significantly more complex scheme. Finally, we do not know the extent to the scalability but we have tested multiple simultaneous connected users, each adding several objects to the virtual environment and the network was capable of managing a small test case, it remains to be seen how far this system can scale, but with some additional work we feel that this system could easily scale to nearly any size. Although there are many areas for improvement, we are in general very pleased with the results of Plane as a prototype and discuss these results and shortcomings in detail in the next chapter.

Chapter 8. Conclussions

When we began our research we started with the long term goal of creating a distributed virtual environment that was graphical, persistent, infinitely scalable, purely peer-to-peer based and constructed entirely from user contribution. In this chapter we conclude this thesis with a review of the first two phases of our research toward this goal. The research we conducted was in evaluating the possible use of a multiagent system as the framework for a distributed virtual environment. We constructed our own generic multiagent system first as it's own research project which we call AI Loom. Once we had iterated on the implementation of AI Loom and were happy with it's stability and usability we progressed to building a prototype distributed virtual environment with AI Loom as the framework which we call Plane. In the first section of this chapter we specifically look at AI Loom, it's design goals, the implementation we choose and how well it met our goals. In the second section we review Plane as a prototype DVE, it's goals and how well the implementation met our goals. Finally, in the last section we discuss further research opportunities in this area.

8.1 Conclusions on AI Loom

8.1.1 Review of AI Loom

In the first phase of our research we hypothesized that we could use a multiagent system as a framework to build a distributed virtual environment. We proceeded to study existing multiagent systems and existing frameworks and determined that there were not any existing multiagent systems that could readily be applied to created a DVE. We concluded that we could create a generic multiagent system in C++ that's design, although still generic, was intended to support the development of a distributed virtual environment. We set six design

goals for AI Loom that we felt were important to a multiagent system that would facilitate the development of a DVE. Those goals were:

- simple API
- pluggable components
- high network transparency
- scalability
- flexibility
- performance

With these goals as our guideposts we designed AI Loom and proceeded through several iterations and finally arrived at a stable, useful implementation for the framework which was both a significant step forward in our own research as well as a contribution to the multiagent systems research field as a whole. AI Loom has now been used in multiple research and commercial projects as a multiagent system.

8.1.2 Evaluation of AI Loom

Our conclusion for AI Loom is that it has met most of our goals. The simple API keeps the learning curve to a minimum and makes integration into existing projects less burdensome than we expected. The pluggable architecture has made rapid prototyping and iteration easy and has been proven to support a wide range of agent types meeting our goal for flexibility. The network architecture is very transparent and being a pure peer-to-peer system is very scalable. The computational performance has proven to be light enough that AI Loom worked in a real time commercial video game and that is better than we had expected. In general our design has met our goals with success.

As AI Loom is used in more projects the design and goals themselves will continue to be tested. From the limited use cases thus far, it would appear that our goals were well targeted, however more usage across a broader spectrum of projects is required to confirm this assertion.

AI Loom is easily adoptable to most C++ object based projects and the network architecture, based on Plexus, is an excellent fit for a distributed MAS. Our main concern for the system at this point is that it has too much transparency to it and perhaps the API could be expanded to allow more user control over network message passing. Several times while working on Plane, we wished there had been better exposure of the networking layer. We resisted the urge to rework part of AI Loom's design at that point but have made it an area of future interest. We have concluded that beyond exposing more network functionality that our system's design has held up well in the limited settings it has been used in and we are anxious for more projects to test the limits of AI Loom.

8.2 Conclusions on Plane

8.2.1 Review of Plane

Our goals for Plane were to create a prototype that demonstrated the possibility of using a multiagent distributed system to create a distributed virtual environment. After completing work on AI Loom we used it to create Plane, a client application that used AI Loom to connect to a peer-to-peer network. The goals we set for Plane were very broad but still only contained a subset of problems associated with creating a complete DVE. Our goals for Plane were:

- Users should be able to connect remotely to a three dimensional virtual environment that they can navigate
- Each user should be represented by an avatar in the virtual environment to himself and to all other users
- Users should be able to create their own three dimensional representation and behavior for their avatar through custom scripts
- Users should be able to load any object with or without scripted behavior into the world as an autonomous agent

- As long as a single computer is connected to a location in the virtual environment all objects in that locale will stay in existence
- Any number of users and objects should be able to be added to the environment

We constructed Plane such that each peer in the network constituted a single agent that was either human controlled or autonomous and every agent had a graphical representation in a virtual environment. Agent's used AI Loom's initialization mechanisms and it's network infrastructure to communicate their initial and changing state as they updated and every connected client stayed in sync through this mechanism.

8.2.2 Evaluation of Plane

Plane was intended as a prototype to demonstrate that a MAS could work as a framework for a DVE and also as a learning tool for us in our research to help us design a feature complete and fully functional DVE. Our conclusion on Plane is that a distributed MAS is a very strong candidate for a framework to build a DVE. In our prototype we accomplished many of the features necessary for a successful DVE. A few of the key attributes that Plane demonstrates are:

- A shared, persistent world state across all clients
- A real-time updating graphical view into the virtual environment
- Extensible objects with very little bounds on what an object can be or what it can do
- simulated many users simultaneously connected

Of the major engineering challenges for building a DVE we have addressed almost all of them, except for security. In the next and final section we discuss security, how the Plane prototype is a very insecure application and what steps might be taken to create a secure distributed virtual environment. However, putting security aside, Plane is a solid step forward in creating a functional peer-to-peer based DVE and we are very happy with it's outcome.

8.3 Future Research

From the beginning of this thesis we have talked about how our long term goal for our research is to create a viable distributed virtual environment. Our research up to this point has focused on finding a solution to a subset of the challenges that are presented by creating a DVE. Naturally then, when talking about future research we will be focused on the remaining challenges that will be faced in proceeding in creating a DVE. Some of the challenges we anticipate are: network security, agent negotiation, space utilization and forcing determinism in scripts. We discuss each of these briefly in the remainder of this section.

8.3.1 Network security

Our research up to this point has disregarded network security primarily because it presents a set of challenges equal to all of the challenges we have faced up to this point. The primary security concern is that scripts are shared over a network and run on every computer in the network, if a user writes a malicious script that, for instance formats the hard drive of the computer it is run on, then we must provide a way to insulate against such attacks. Also, we currently do not provide any security at the network level, all messages are passed unencrypted and there is no validation of users when they connect to a network. All of these things and probably others will need to be dealt with for a complete DVE to be constructed.

8.3.2 Agent negotiation

In the Plane prototype we do not deal with collision detection or with how agents interact unless they are specifically designed to interact by the users who wrote their behavior. In a completed DVE at least some amount of negotiation will be necessary for all agents to inhabit the same virtual environment. We would say that the system would require at least some basic rules which govern all agents, and actually, in Plane there is one requirement that all agents are set on the ground. We suspect that additional basic rules will be required and this is an open area for research

8.3.3 Space utilization

Using up space in the virtual environment will be as much of an engineering problem as a social and economical problem. In second life space is sold to anyone who wants to own their own land, this is as much an economic decision as it is an engineering one. In second life, a client-server based application land is controlled through the server, however in a peer-to-peer system there is no authority who can dictate who owns land. The question remains how to best handle space reservation in a distributed virtual environment. It may be possible to utilize a registrar system, similar to the domain name registration service on the Internet that reserves space in the world to a specific client machine. Although this seems like a possible solution this is also definitely an open problem and will require future research to find an optimal solution.

8.3.4 Determinism

Up to this point we have glossed over the fact that in order for agents to stay in sync across the network, all of the agent effectors must be deterministic. If they are not then agents will perform differently on different nodes in the network and the entire system breaks down. Forcing determinism will require some retooling of how effectors are exposed to users and will potentially require some redesign of AI Loom in order to facilitate a solution to this problem.

Appendix. Simple Program demonstrating AI Loom

The following is a sample application that is distributed with the AI Loom library which demonstrates a straightforward usage of AI Loom.

```
#include <iostream>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

#include <boost/bind.hpp>
#include <boost/function.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/thread.hpp>

#include <lm/Agent.h>
#include <lm/Defs.h>
#include <lm/EffectorFactory.h>
#include <lm/LoomKernel.h>
#include <lm/Output.h>
#include <lm/SensorFactory.h>

using namespace lm;

/**
 * This sample application is a very basic sample app.  The basic idea is that
 * we are creating a senturyGaurd in a game that for now simply walks back and
 * forth.  Later we'll add logic for him to attack and run away but for now we
 * just have two functions he can perform, moveNorth and moveSouth and he
 * should just pace back and forth.  In AI Loom terms this translates into
 * 2 effectors, 1 sensor and a decider.
 * The 1 sensor returns 1 if the current time (in seconds) is even or 0 if it
 * is odd.  The decider then looks at the result of the sensor and decides
 * which effector to apply.  The decider simply says I'm sensing a one then
 * moveNorth, otherwise moveSouth.  And, for now the effect is simply to
 * output to the error stream if we are moving north or south.
 *
```

```
 * Clearly this isn't any sort of really intelligent agent, but it is just a
 * demonstration of how to use AI Loom and this hopefully show how loom works
 * with a minimal amount of work!
 */
```

Free function used as an AI Loom sensor

```
/**************************************************************************
 * SENSOR FUNCTIONS
 **************************************************************************/

int sence()
{
   time_t sec;
   sec = time(NULL);
   if(sec%2==0)
      return 1;
   else return 0;
}
```

Decider class

```
/**************************************************************************
 * DECIDER
 **************************************************************************/

/**
 * now we implement a decider by deriving from the base Decider class
 */
class MovementDecider : public Decider
{
public:
   /**
    * override the constructor but pass the base constructor the default
    * effector given at instantiation time
    */
   MovementDecider(Effector* e) : Decider(e)
   {
   }

   /**
    * We override the pure virtual function makeDecision with our own
    * implementation of how to make a decision
    */
   Effector* makeDecision()
   {
```

```
      /** iterate over all the sensors to see if one is called IsADown, if so
       * then return the first effector otherwise return the last effector
       */
      if(mSensorsResults["secondSence"]==1)
      {
         return mEffectors["moveNorthEffect"];
      }
      else
      {
         return mEffectors["moveSouthEffect"];
      }
   }
};
```

Agent Class

```
/*****************************************************************************
 * Agents
 *****************************************************************************/
class SenturyGuard : public lm::Agent
{
public:
   SenturyGuard(std::string name) : lm::Agent(name)
   {
   }

   void moveNorth()
   {
      debug::Output("ERROR", "moving North\n");
   }

   void moveSouth()
   {
      debug::Output("ERROR", "moving South\n");
   }
};
```

Application's main function

```
/*****************************************************************************
 * APPLICATION
 *****************************************************************************/
int main()
{
   /** first things first, initialize the Loom Kernel */
   LoomKernel* mKernel = new LoomKernel();
```

```
mKernel->init();

// if you are on a multi-processor machine and you don't want to use all the
// processors you can uncomment this line to force Loom to only use 1 thread
// on 1 processor.
mKernel->forceSingleThread(true);

/** next create an agent */
SenturyGuard* mAgent = new SenturyGuard(std::string("sampleAgent"));

/** create our sensor using the sensorFactory */
boost::function0<int> senceFunc = &sence;
Sensor* testSensor =
    SensorFactory::instance().generateSensor("secondSence",
                     "returns 1 if the seconds are even", senceFunc);

/// this uses an older form of boost function and boost bind to accomidate
// older compilers and hence is more complex than is absolutely required.

/** create the tick and tock Effectors using the EffectorFactory. */
boost::function0<void> moveNorthFunc = boost::bind(&SenturyGuard::moveNorth,
                                                   mAgent);
Effector* moveNorthEffect =
    EffectorFactory::instance().generateEffector("moveNorthEffect",
                                                 "moves gaurd north",
                                                 moveNorthFunc);

boost::function0<void> moveSouthFunc = boost::bind(&SenturyGuard::moveSouth,
                                                   mAgent);
Effector* moveSouthEffect =
    EffectorFactory::instance().generateEffector("moveSouthEffect",
                                                 "moves gaurd south",
                                                 moveSouthFunc);

/**
 * finally create a simple Decider we give it the tickEffect Effector as a
 * default effect.  This is a safegaurd that loom requires in case you
 * mis-implement your decider in a way that does not return an Effector.  In
 * that case Loom will use the Effector you supply to the constructor as the
 * default Effect.  In this case the default effector is the tickEffect.
 */
MovementDecider moveDecider(moveNorthEffect);

/** add the timeSensor, tickEffect, and tockEffect to the agent */
mAgent->add(testSensor);
mAgent->add(moveNorthEffect);
```

```
   mAgent->add(moveSouthEffect);

   /** register the agent with the kernel. */
   mAgent->registerAgent(mKernel);

   /** tell the agent to use the decider we created */
   mAgent->swapDecider(moveDecider);

   /** sit and spin */
   while(1)
   {
      /**
       * if you are using multi-threaded support then this call is not necessary
       * (Loom just stubs it out).  But if you are on a single processor machine
       * it is required for Loom.  Therefore in writing your own app you should
       * include this in your own control loop.
       */
      mKernel->update();
   }
}
```

Bibliography

[1] N. Stephenson, *Snow Crash*. 1745 Broadway, 3rd Floor, New York, NY 10019: Bantam Books, 1992.

[2] A. Wachowski and L. Wachowski, *The Matrix*. [Video] Warner Brother's Pictures, 1999. http://whatisthematrix.warnerbros.com.

[3] J. Rusnak, *The Thirteenth Floor*. [Video] Columbia Pictures, 1999. http://www.centropolis.com.

[4] J. R. Okin, *The Internet Revolution: The Not-for-Dummies Guide to the History, Technology, and Use of the Internet*. Ironbound Press, 2005.

[5] J. Abbate, *Inventing the Internet (Inside Technology)*. The MIT Press, 2000.

[6] 1UP, *Star Wars Nets 275,000 Users*. FindArticles.com, 2003. http://tinyurl.com/2sj82e.

[7] R. Fahey, "Chinese success pushes world of warcraft past 3.5 million users," 2005. http://tinyurl.com/387d82.

[8] MySpace.com, "Myspace," 2007. http://www.myspace.com.

[9] M. Technologies, "There - the online virtual world that is your everyday hangout," 2007. http://www.there.com.

[10] S. Life, "Second life: Your world. your imagination.," 2007. http://www.secondlife.com.

[11] J. M. Allbeck and N. I. Badler, "Avatars a la snow crash," in *CA*, pp. 19–24, 1998. citeseer.ist.psu.edu/allbeck98avatars.html.

[12] R. Schroeder, *The Social Life of Avatars*. Springer, 2002.

[13] J. Donham, "Managing and growing an mmog as a service," in *Massively Multiplayer Game Development 2*, pp. 425–448, Charles River Median, Inc., 2005.

[14] B. Entertainment, "World of warcraft.," 2006. http://www.wow-europe.com/de/.

[15] www.google.com, "Google," 2007. http://www.google.com.

[16] A. M. Turing, "Computing machinery and intelligence," in *Mind*, pp. 59:433–460, 1950.

[17] I. Millington, *Artificial Intelligence for Games*. Morgan Kaufmann publishers, 2006.

[18] S. Russell and P. Norvig, *Aritifical Intelligence - A modern Approach*. New Jersey, NJ: Prentice Hall Artificial Intelligence Series, 1995.

[19] J. Haugeland, ed., *Artificial Intelligence: The Very Idea*. Cambridge, MA: MIT Press, 1985.

[20] R. Kurzweil, *The Age of Intelligent Machines*. Cambridge, MA: MIT Press, 1990.

[21] E. Charniak and D. McDermott, *Inroduction to Artificial Intelligence*. Potomac, MA: Addison-Wesley publishing, 1987.

[22] R. J. Schalkoff, *Aritificial Intelligence: An Engineering Approach*. New York, NY: McGraw-Hill, 1990.

[23] F.-H. Hsu, *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002.

[24] B. Schwab, *AI Game Engine Programming*. Charles River Media, 2004.

[25] R. C. Atkinson and R. M. Shiffrin, "Human memory: A proposed system and its control processes," in *K. W. Spence and J. T. Spence (Eds.), The Psychology of learning and motivation: Advances in research and theory (vol. 2)*, New York: Academic Press, 1968.

[26] D. O. Hebb, *THe organization of behavior; a neuropsychological theory*. Wiley-Interscience, New York, 1949.

[27] G. Gillund and R. M. Shiffrin, "A retrieval model for both recognition and recall," in *Psychological Review,91*, pp. 1–67, 1984.

[28] M. Dastani and J.-J. C. Meyer, "Programming agents with emotions," in *ECAI 2006: 17th European Conference on Arificial Intelligence*, IOS Press, 2006.

[29] J. J. C. Meyer, "Reasoning about emotional agents," in *Proc.16th European Conf. on Artificial Intelligence (ECAI 2004)*, IOS Press.

[30] J. Tao, T. Tan, and R. W. Picard, *Affective Computing and Intelligent Interaction*. LNCS 3784, Springer, Berlin, 2005.

[31] M. E. Bratman, *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.

[32] D. C. Dennett, *The Intentional Stance*. MIT Press, Cambridge, Mass., 1987.

[33] A. Sloman, "What sort of architecture is required for a human-like agent?," tech. rep., School of Computer Science and Cognitive Science Research Centre, 1996.

[34] M. Wooldridge, *An Introduction to MultiAgent Systems*. John Wiley and Sons, LTD, 2002.

[35] G. Weiss, "Prologue, multiagent system and distributed artificial intelligence," in *Multi-agent Systems, A modern Approach to Distributed Artificial Intelligence*, pp. 1–23, MIT Press, 1999.

[36] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," in *The Knowledge Engineering Review*, pp. 10(2):115–152, 1995, 1995.

[37] J. Ferber, *Multi-Agent Systems, an Introduction to Distributed Artificial Intelligence*. Addison Wesley, 1999.

[38] M. d'Inverno and M. Luck, *Understanding Agent Systems*. Springer-Verlag Berlin Heidelberg, 2001,2004.

[39] C. Sammut, S. Hurst, D. Kedzier, and D. Michie, "Learning to fly," in *Proceedings of the Ninth International Conference on Machine Learning*, (Aberdeen), Morgan Kaufmann, 1992.

[40] D. Michie, "Current developments in expert systems," in *Proc. 2nd Australian Conference on Applicaoins of Expert Systems*, pp. 163–182, 1986.

[41] J. R. Quinlan and R. M. Cameron-Jones, "FOIL: A midterm report," in *Machine Learning: ECML-93, European Conference on Machine Learning, Proceedings*, vol. 667, pp. 3–20, Springer-Verlag, 1993.

[42] G. Holmes, A. Donkin, and I. Witten, "Weka: a machine learning workbench," in *Proceedings Second Australia and New Zealand Conference on Intelligent INformation Systems*, (Brisbane, Australias), pp. 357–361, 1994.

[43] S. Sen and G. Weiss, "Learning in multiagent systems," in *Multiagent Systems, A modern Approach to Distributed Artificial Intelligence*, pp. 259–298, MIT Press, 1999.

[44] A. Blum and M. Furst, "Fast planning through planning graph analysis," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pp. 1636–1642, 1995.

[45] R. E. Fikes and N. J. Nilsson, "Strips: a new approach to the application of theorem proving to problem solving," in *Artificial Intelligence*, pp. 3(4):251–288, 1971.

[46] H. Munoz-Avila and H. Hoang, "Coordinating teams of bots with hierarchical task network planning," in *AI Game Programming Wisdom 3*, pp. 417–427, 2006.

[47] J. Orkin, "Applying goal-oriented action planning to games," in *AI Game Programming Wisdom 2*, pp. 217–228, 2004.

[48] G. Weiss, *Multiagent Systems, A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.

[49] C. Lucena, A. Garcia, A. Romanovsky, J. Castro, and P. A. (Eds.), *Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications*. Springer-Verlag, 2004.

[50] D. Lubke and J. M. Gomez, "Designing a foundation for mobile agents in peer-to-peer networks," in *Architectural Design of Multi-Agent Systems Technologies and Techniques*, pp. 115–124, 2007.

[51] P. Marques and L. Silva, "Component agent systems: Building a mobil agent architecture that you can reuse," in *Architectural Design of Multi-Agent Systems Technologies and Techniques*, pp. 95–114, 2007.

[52] R. S. Gray, "Agent Tcl: A flexible and secure mobile-agent system," in *Fourth Annual Tcl/Tk Workshop (TCL 96)* (M. Diekhans and M. Roseman, eds.), (Monterey, CA), pp. 9–23, 1996.

[53] H. Peine and T. Stolpmann, "The architecture of the Ara platform for mobile agents," in *First International Workshop on Mobile Agents MA'97* (R. Popescu-Zeletin and K. Rothermel, eds.), vol. 1219 of *Lecture Notes in Computer Science*, (Berlin, Germany), pp. 50–61, Springer Verlag, Apr. 1997.

[54] J. Hertz, A. Kdogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991.

[55] D. M. Bourg and G. Seemann, *AI for Game Developers*. O'Reilly Media Inc, 2004.

[56] P. Sweetser, "How to build neural networks for games," in *AI Game Programming Wisdom 3*, pp. 615–625, Charles River Media, 2006.

[57] C. Baekkelund, "A brief comparison of machine learning methods," in *AI Game Programming Wisdom 2*, pp. 617–631, Charles River Media, 2004.

[58] M. Pfister, *Learning Algorithms for Feed-Foward Neural Networks - Design, Combination and Analysis*. Fortschrittberichte VDI-Verlag, Dusseldorf, 1996.

[59] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[60] A. J. Champandard, "The dark art of neural networks," in *AI Game Programming Wisdom*, pp. 640–651, Charles River Media, 2000.

[61] R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[62] D. Fu and R. Houlette, "Constructing a decision tree based on past experience," in *AI Game Programming Wisdom 3*, pp. 567–579, Charles River Media, 2006.

[63] A. S. Tannenbaum and M. van Steen, *Distributed Systems - Principles and Paradigms*. New Jersey, NJ: Prentice Hall, 2002.

[64] R. Subramanian and B. D. Goodman, *Peer to Peer Computing, The Evolution of a Distruptive Technology*. Idea Grup Publishing, 2005.

[65] K. T. Greenfeld, "Meet the napster.," *Time*, October 2, 2000.

[66] S. Gardner and K. Krug, *BitTorrent for Dummies*. Wiley Publishing, Inc, 2005.

[67] J. Bloomer, *Power Programming with RPC (Nutshell Handbooks)*. O'Reilly Media, Inc., 1992.

[68] S. S. Laurent, E. Dumbill, and J. Johnston, *Programming Web Services with XML-RPC (O'Reilly Internet Series)*. O'Reilly Media, Inc., 2007.

[69] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education (ISE Editions), 2003.

[70] J. Sloan, *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI (Nutshell Handbooks)*. O'Reilly Media, Inc, 2004.

[71] ISO, *Open Distributed Processing Reference Model*. International Standard ISO/IEC IS 10746, 1995.

[72] http://www.fipa.org, "Foundation for intelligent physical agents website," 2003.

[73] R. A. Bartle, *Designing virtual Worlds*. New Riders Publishing, 2003.

[74] A. Bierbaum, "Vr juggler: A virtual platform for virtual reality application development," Master's thesis, Iowa State University, 2000. http://www.vrjuggler.org/pub/Bierbaum-VRJuggler-MastersThesis-final.pdf.

[75] P. Hartling, "Avatar manipulation as a metaphor for virtual reality system simulator interaction," 2004.

[76] "Access grid," November 5th 2007. http://www.accessgrid.org.

[77] "Earth system grid."

[78] S. T. I. Foster, C. Kesselman, "The anatomy of the grid," in *International Journal of Supercomputer Applications*, p. 15, 2001.

[79] I. Foster and C. Kesselman, *The Grid: Blueprint for a new computing infrastructure*. Morgan Kaufmann Publisher, 2005.

[80] "Seti@home," November 5th 2007. http://setiathome.berkeley.edu/.

[81] M. F. W. Scott A. DeLoach and C. H. Sparkman, "Multi systems enginnering," *Internation Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 3, 2001.

[82] S. A. DeLoach and M. Wood, "Developing multiagent systems with agenttool," 2000.

[83] http://www.ai.sri.com/ oaa, "Open agent architecture website," 2003.

[84] D. L. Martin, A. Cheyer, and G. L. Lee, "Agent development tools for the open agent architecture," in *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, (London), pp. 387–404, The Practical Application Company Ltd., Apr 1996.

[85] A. C. Ramos, J. Gensel, M. V. Oliver, and H. Martin, "A peer ubiquitous multi-agent framework for providing nomadic users with adapted information," in *Agents and Peer-to-Peer Computing*, pp. 159–172, Springer Berlin, 2006.

[86] M. Wang, H. Wolf, M. Purvis, and M. Purvis, "An agent-based collaborative framework for mobile p2p applications," in *AP2PC 2005, LNAI 4118*, pp. 132–144, Springer-Verlag, 2006.

[87] D. Flanagan, *Java In a Nutshell, 5th Edition*. O'Reilly Media, Inc., 2005.

[88] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley and Sons Ltd, 2007.

[89] http://openai.sf.net, "openai website," 2003.

[90] "Microsoft robotics studio," November 5th, 2007. http://msdn2.microsoft.com/en-us/robotics/default.aspx.

[91] A. Montresor, "Anthill: a framework for the design and analysis of peer-to-peer systems," in *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems*, 2001. http://www.cs.unibo.it/projects/anthill/papers/anthill-01.pdf.

[92] *Skype for Dummies (For Dummies)*.

[93] T. Patkos and D. Plexousakis, "A semantic marketplace of negotiating agents," in *Agents and Peer-to-Peer Computing*, pp. 159–172, Springer Berlin, 2006.

[94] C. Ellis and J. Wainer, "Groupware and computer supported cooperative work," in *Multiagent Systems, A modern Approach to Distributed Artificial Intelligence*, pp. 425–458, MIT Press, 1999.

[95] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *International Conference on Peer-to-Peer Computing*, 2001.

[96] "The gnutella home page," 2001.

[97] "The gnutella2 developer network," November 5th, 2007. www.gnutella2.com.

[98] S. Willmott and U. C. Josep Pujol, "On exploiting agent technology in the design of peer-to-peer applications," in *Agents and peer-to-peer computing : (Third international workshop, AP2PC 2004)*, 2004.

[99] S. Y. Lee, O.-H. Kwon, J. Kim, and S. J. Hong, "A trust management scheem in structured p2p systems," in *Agents and peer-to-peer computing : (Fourth internationa workshop, AP2PC 2005)*, pp. 30–43, 2005.

[100] "Gnu lesser general public license website," June 29, 2007. http://www.gnu.org/licenses/lgpl.html.

[101] B. Stroustrup, *The C++ Programming Language*. Addison Wesley, 2000.

[102] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, New York, NY: Addison-Wesley Publishing Company, 1995.

[103] M. Mealling, P. J. Leach, and R. Salz, "A UUID URN namespace," tech. rep., Internet Engineering Task Force, 2002. http://www.ietf.org/internet-drafts/draft-mealling-uuid-urn-00.txt.

[104] C. Kozierok, *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. William Pollock, No Starch Press, Inc, 2005.

[105] "Virtual reality applications center website," Nov 5, 2007. http://www.vrac.iastate.edu.

[106] "Treefort wars website," Nov 5, 2007. http://www.treefortwars.com.

[107] "Independent games festival," Nov 5, 2007. http://www.igf.com.

[108] G. Junker, *Pro OGRE 3D Programming*. Springer-verlad, 2006.

[109] "Obre 3d: Open source graphics engine website," Nov 5, 2007. http://www.ogre3d.org.